QoS-Aware Power Management via Scheduling and Governing Co-Optimization on Mobile Devices

Qianlong Sang, Jinqi Yan, Rui Xie, Chuang Hu, Kun Suo, Dazhao Cheng, *Senior Member, IEEE*

Abstract—Scheduling and governing are two key technologies to trade off the Quality of Service (QoS) against the power consumption on mobile devices with heterogeneous cores. However, there are still defects in the use of them, among which two of the decoupling issues are critical and need to be resolved. First, both the scheduling and governing decouple from QoS, one of the most important metrics of user experience on mobile platforms. Second, scheduling and governing also decouple from each other in mobile systems and they might weaken each other when being effective at the same time. To address the above issues, we propose Orthrus, a comprehensive QoS-aware power management approach that involves a governing approach based on deep reinforcement learning to adjust the frequency of heterogeneous cores, a scheduling algorithm based on finite state machine that assigns cores to QoS-related threads, and expert fuzzy control-based coordination mechanism between the two to manage the impact between scheduling and governing. Our proposed approach aims to minimize power consumption while guaranteeing the QoS. We implement Orthrus on Google Pixel 3 as the system service of Android and evaluate it using several widespread mobile applications. The performance evaluation demonstrates that Orthrus reduces the average power consumption by up to 35.7% compared to three state-of-the-art techniques while ensuring the QoS on mobile platforms.

Index Terms—Power management, Quality of Service, Scheduling, Governing, Mobile devices, Reinforcement learning.

1 Introduction

With the advancements on mobile devices, various high-performance applications such as web browser [1] [2], video [3] and game [4] have been deployed, presenting challenges in balancing the power consumption and application QoS. In this study, considering the typical characteristics of Android applications which usually have a target frame rate, the standard QoS metric is the *frame rate*. As the frame rate increases, the display appears smoother, enhancing the user experience [5]. However, high frame rates can significantly increase the system power consumption due to the rendering of frames [6] [7].

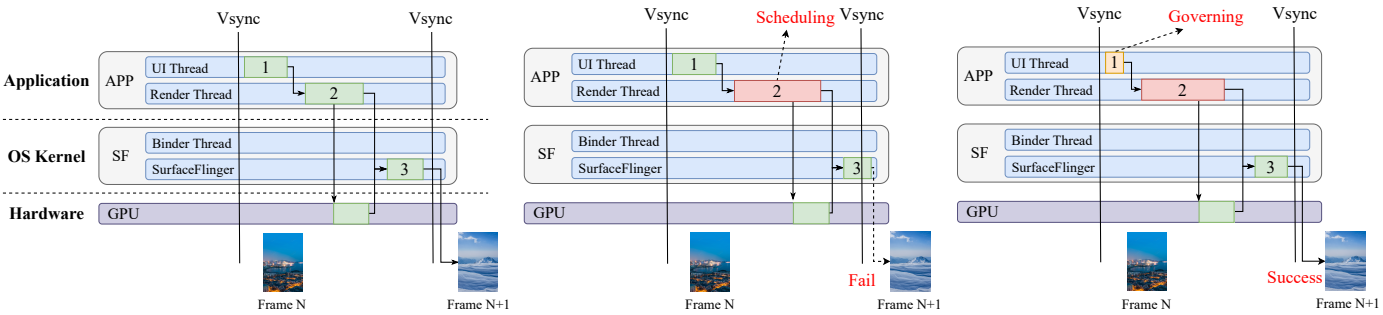
The trade-off between power consumption and QoS on mobile devices has always been a well-known research area. In hardware, heterogeneous CPUs are currently deployed on mobile phone System-on-Chip (SoC) designs, the most representative of which is Arm's big.LITTLE architecture [8]. "LITTLE" cluster is designed for maximum power efficiency while "big" cluster is designed to provide maximum computing performance. The LITTLE cluster is utilized to handle low-computation tasks to reduce power consumption, and big cluster is used to handle high-computation tasks to guarantee QoS. On top of such hardware, software solutions like scheduling and governing contribute significantly to managing QoS and power consumption on mobile devices [9]. Scheduling is responsible for assigning threads to CPU cores, while governing controls the frequencies of each CPU cluster. The position of QoS-related threads and the frequency of the core they are located in determine the execution time of frame generation, which affects QoS and system power consumption simultaneously. For time-sensitive tasks with strict time requirements, they have special scheduling and governing strategies, which are not considered in this paper.

In fact, lots of efforts have recently been made to optimize power management on mobile devices using these two techniques [10]–[16]. They have focused on mapping threads or adjusting frequency by monitoring the application status and the workload variations. However, there are still limitations in their current implementations, among which two issues caused by the decoupling design of the scheduling and governing are particularly critical and in need of resolution.

First, both scheduling and governing decouple from QoS on mobile devices. Specifically, current scheduling and governing decisions are based solely on the utilization of tasks, without considering the frame rate of upper-layer applications, resulting in performance degradation and unnecessary power consumption. For instance, scheduling will migrate QoS-related threads to the LITTLE cluster to reduce power consumption, yet the LITTLE cluster may not have enough computing capacity, thus leading to a decrease in QoS. Similarly, governing might increase the frequency of the cluster due to high utilization, yet if there are few QoS-related threads running on this cluster, power consumption is wasted.

Second, scheduling and governing decouple from each other. They not only fail to fulfill their respective roles but also risk mutual interference when simultaneously effective. For example, governing might reduce the frequency of one cluster when some QoS-related threads are migrated to it, or scheduling may migrate one thread from a cluster right after its frequency increases. Furthermore, the lack of joint

- *Qianlong Sang, Jinqi Yan, Chuang Hu, and Dazhao Cheng are with the School of Computer Science, Wuhan University, Hubei 430072, China. (E-mail: {qlsang,blues2431,handc,dcheng}@whu.edu.cn.) (Corresponding author: Chuang Hu, Dazhao Cheng.)*
- *Rui Xie is with the Guangdong Oppo Mobile Telecommunications Corp, Guangdong 510000, China.(E-mail: xierui@oppo.com)*
- *Kun Suo is with the Department of Computer Science, Kennesaw State University, GA 30144, USA.(E-mail: ksuo@kennesaw.edu)*



(a) *Process of Frame Rendering* (b) *Impact of Scheduling to Frame Rendering* (c) *Impact of Governing to Frame Rendering*

Fig. 1: Frame rendering and the impact of scheduling and governing on frame rendering.

optimization results in large delays in the responses to each other's actions. Many studies [17]–[19] have focused on the optimization of these two techniques separately, but few have studied their interaction and made them work efficiently together. To provide better QoS and lower power consumption on mobile platforms, it is essential to jointly optimize scheduling and governing with QoS awareness.

In this paper, we propose *Orthrus*[1], a QoS-aware scheduling and governing co-optimization approach for mobile devices. First, we design a Deep Reinforcement Learning (DRL)-based governor, which can sense QoS and system status to adjust the frequency of CPU cluster. Secondly, we design a new scheduler based on a Finite State Machine (FSM), which schedules QoS-related threads and maintains the system in a state that just satisfies the QoS. Lastly, we design a coordinator using Expert Fuzzy Control (EFC) to coordinate the interaction of the previous two modules, further satisfying the QoS and reducing power consumption. We implement Orthrus on real mobile devices and evaluate it with various scenarios. Through experiments, we verify that Orthrus achieves higher QoS at lower power consumption compared to state-of-the-art implementations. We develop a case study where we implement Orthrus to support users' daily use of mobile phones for 1 hour. This study shows the operations of Orthrus encountering various applications and its consistent high performance and strong practicality.

To summarize, this paper has made the following contributions:

- We investigate the impact of scheduling and governing on QoS, experimentally reveal wasted power caused by unawareness of QoS, and analyze their interaction (§2).
- We propose a DRL-based governing strategy to control the frequency of different clusters (§4.1), and a scheduler based on finite state machine to meet the QoS while minimizing power consumption (§4.2).
- We analyze the relationship between scheduling and governing, design the coordinator module to store frequency information from the governor, and use EFC to control the influence of the scheduler on the governor (§4.3).
- We implement Orthrus at the application layer of the Android platform on real mobile devices (§5) and evaluate Orthrus with three popular applications (§6). We present a case study (§7), showing the operation of Orthrus in daily use.

1. In Greek mythology, Orthrus was a two-headed dog who guarded Geryon's cattle. This paper is aimed to co-optimize scheduling and governing (two-headed) for QoS-aware power management (guard cattle) on mobile devices.

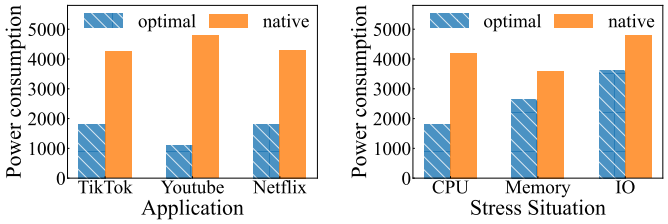
2 Background and Motivation

2.1 Background

2.1.1 Frame Rendering

Frame rendering is the act of generating a frame from applications and displaying it on the screen [20], which requires the joint efforts of software and hardware. Normally there are three main threads to work together for that: *UI Thread*, *Render Thread*, and *SurfaceFlinger*. The UI Thread and Render Thread belong to the foreground application process, while SurfaceFlinger is Android's system service. As shown in Fig. 1(a), there are three steps in the frame rendering process and each thread must execute in turn [21]. 1) UI Thread processes input events, creates a tree of drawing commands, and then passes them to the Render Thread. 2) Render Thread fetches the buffer from SurfaceFlinger and sends a rendering request to the GPU. Then, the Render Thread enqueues the buffer into the BufferQueue managed by SurfaceFlinger. 3) SurfaceFlinger composites buffers from different sources such as foreground applications and navigation bars. In order to coordinate the rendering and display of frames, the system generates a Vertical Synchronization (Vsync) signal at certain intervals to update the display and three threads must complete their work within the interval. In addition, Android uses the Binder mechanism for inter-process communication, which also helps to render frames. It is worth noting that although frame rendering must be completed within the Vsync interval, failure to do so does not result in severe consequences. This contrasts with conventional real-time tasks, where exceeding these constraints could lead to serious failures or system malfunctions. Therefore, such scenarios are beyond the scope of this paper.

The frame rate of applications on mobile devices, especially smartphones, is often a measure of QoS [22]–[26]. On mobile devices, the QoS requirements for different applications, or even the same application, can vary greatly. For example, chat software requires 10 frames per second (FPS), video playback needs 24 FPS, and games choose 60 or 120 FPS depending on the hardware capacity [27].



(a) *Governing: Optimal Vs Native* (b) *Scheduling: Optimal Vs Native*

Fig. 2: Power consumption between optimal and native in scheduling and governing when QoS is satisfied.

2.1.2 Scheduling and Governing on Mobile Devices

On mobile devices, scheduling is the action of allocating core to execute threads, taking into account the different computing requirements and system status. Governing is another technique that can automatically adjust the frequency of cores according to different situations, thus saving the system's energy consumption. Both scheduling and governing control the computing capacity of the system and have a critical impact on QoS. In terms of scheduling, as depicted in Fig. 1(b), the execution time of Render Thread will be too long if it is scheduled to run on the LITTLE cluster that cannot provide enough computing capacity. When the next Vsync comes, the frame is not rendered in time, resulting in a decrease in QoS. Governing also has a crucial impact on QoS, as illustrated in Fig. 1(c). Assuming that the governor increases the frequency of the cluster where the UI Thread is located, the time for UI Thread to run tasks will be shortened. In this case, the execution times of all three threads will be shorter than the Vsync interval, ensuring the generation of a perfect frame and thus providing a better user experience.

However, current scheduling and governing techniques are unaware of user-level QoS. The native scheduler used by mobile operating systems on heterogeneous platforms is *Energy Aware Scheduling (EAS)* [28], which schedules threads to the core with the lowest power consumption using the power model provided in advance by the hardware, ignoring the optimization of QoS. As a result, threads assigned to cores with inadequate computing capacity will suffer from increased execution time, ultimately degrading QoS. According to different purposes, there are many governors on mobile devices to choose from [29]. For example, the *performance* governor sets the frequency to the maximum for best performance, while the *powersave* governor sets the frequency to the lowest value to save power as much as possible. As Android's default governor, *schedutil* adjusts the frequency based on the computing capacity of the cluster and the maximum load within the cluster. Nevertheless, there exists a mismatch between QoS and CPU utilization. A thread that contributes to frames may have low utilization, while a thread that does not contribute to frames may have high utilization. In this case, *schedutil* will incorrectly set the frequency, resulting in loss of QoS and waste of power consumption. Essentially, current schedulers and governors make decisions based on CPU utilization, neglecting the upper-layer QoS. Furthermore, the work of the two modules is decoupled, resulting in suboptimal QoS and wasted energy consumption (we will detail it in Section 2.2).

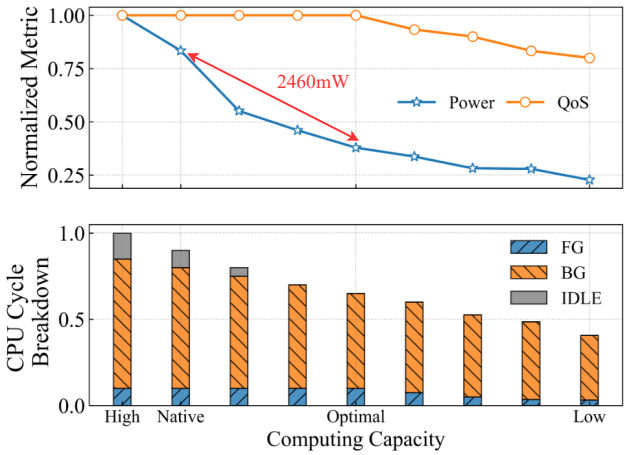


Fig. 3: Normalized power consumption, QoS, and breakdown of CPU resources for TikTok applications at different computing capacities. CPU cycles are broken down into foreground applications (FG), background applications (BG), and being idle.

2.2 Motivation

We conduct some measurements to demonstrate the potential enhancement of existing scheduling and governing approaches. The results demonstrate that the co-optimization of QoS-aware scheduling and governing has a great opportunity to reduce power consumption while achieving the target QoS.

We perform all measurements on Google Pixel 3, the same device used in the experimental section [2]. We use TikTok, Youtube, and Netflix applications as foreground applications with a target QoS of 60 FPS. In addition, we run three different loads of CPU-stress, Memory-stress and I/O-stress in the background to simulate the system load of the smartphone. The CPU-stress uses eight processes for the square root calculation, Memory-stress uses eight processes that malloc 256MB of memory each and write characters to dirty it, and the I/O-stress uses eight processes to execute the sync() function continuously. The source code and implementation details of the stress program are available online [30].

2.2.1 The Decouple Design of Governing and QoS Optimization

We use three different applications as foreground applications, and the CPU stress runs in the background. By traversing all the frequency points, we can find the frequency point that meets the user's QoS and consumes the least power. This method is called *optimal* governor. As shown in Fig. 2(a), with both the *optimal* and *native (schedutil)* meeting QoS requirements, the average power consumption of the *optimal* is 64.8% lower than that of the *native*.

As shown in Figure 3, we take TikTok as an example and reveal why there is such a large waste of power consumption by analyzing the changes in power consumption, QoS, and CPU cycle breakdown when the CPU frequency goes from high to low. We find that because *native* adjusts the frequency of clusters based solely on the CPU utilization,

2. In this paper, the unit of power consumption is milliwatts (mW).

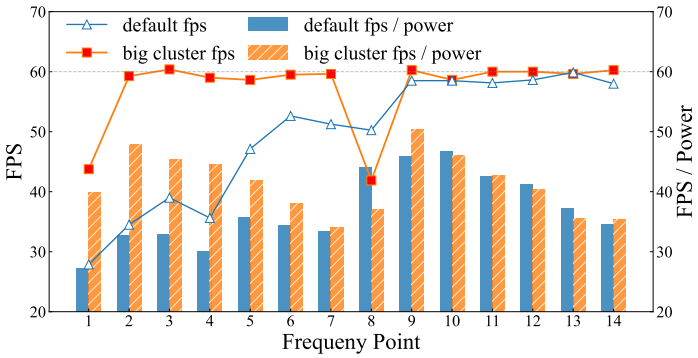


Fig. 4: FPS and FPS/Power at different frequency points when threads are aggregated on the big cluster or not.

when CPU stress causes an increase in CPU utilization, native will adjust the CPU to a very high frequency. At this time, the computing capacity is largely supplied to applications unrelated to QoS, resulting in a waste of power consumption. As the frequency decreases, the CPU idle time drops to negligible. Since foreground applications have higher priority than background applications and mobile device manufacturers will use some mechanisms to protect foreground applications, the CPU resources obtained by background applications are further reduced, while the CPU resources of foreground applications are not affected. At this time, there is no loss of QoS, and the power consumption is reduced by more than 2000 mW. In real-world scenarios, these background services such as GPS refresh have little impact on the QoS, but the existing governor *native* will introduce much power consumption waste.

Observation 1: There exists a clear gap between the performance of governing and the optimal QoS. Ideally, the governor should reduce the frequency as much as possible while ensuring the QoS, and slow down the operation of tasks that are unrelated to the QoS, thereby significantly saving power consumption.

2.2.2 The Decouple Design of Scheduling and QoS Optimization

We use TikTok as the foreground application and run three different stress programs in the background. By traversing all possible core assignments for threads that contribute to frame rendering, such as the UI Thread, Render Thread, SurfaceFlinger, and Binder Thread, we select a scheduling policy that satisfies QoS and minimizes power consumption. This method is named *optimal* scheduler. For a system with n threads and m CPU cores, the scheduling problem can be modeled as an n-dimensional vector, where each element has m possible values. Therefore, the time complexity of exhaustively searching for the optimal solution is $O(m^n)$, which is impractical. As a result, the optimal scheduler is employed solely to uncover inefficiencies in the *native (EAS)* scheduling strategy and to demonstrate the gap between the *native* strategy and an *optimal* strategy. As shown in Fig. 2(b), the average power consumption of the *optimal* is 36.1% lower than that of the *native*. This is because *native* schedules each thread on the core with the least power consumption and does not give special treatment to threads related to QoS. Even if there is Cgroup mechanism [31] that puts the foreground application thread on big cluster or increases the priority, it does not address the fundamental problem that the scheduler lacks upper-layer QoS awareness.

TABLE 1: Optimal frequencies under different scheduling strategies for UI Thread (U), Render Thread (R) and SurfaceFlinger (SF). b denotes the big cluster; L denotes the LITTLE cluster; n denotes the native choice.

U	R	SF	Power (mW)	LITTLE Frequency (GHz)	big Frequency (GHz)
b	*b*	*b*	1284	0.3	0.8
b	*b*	*L*	1354	0.6	0.8
b	*L*	*b*	1554	1.1	0.8
b	*L*	*L*	1488	1.1	1.3
L	*b*	*b*	1388	0.9	0.8
L	*b*	*L*	1398	0.9	0.8
L	*L*	*b*	1512	1.1	0.8
L	*L*	*L*	1508	1.1	0.3
n	*n*	*n*	1750	n	n

Observation 2: existing scheduling schemes do not work closely with QoS. The scheduler can operate more efficiently based on the information from the application layer, and can assign cores with different computing capabilities to QoS-related threads depending on whether the QoS is satisfied.

2.2.3 The Decouple Optimization of Governing and Scheduling on QoS

In the current state of research, scheduling and governing are often optimized separately. While some studies have focused on optimizing QoS, they have done little joint work for scheduling and governing, which can result in suboptimal performance and energy consumption. To demonstrate this deficiency, we conducted an experiment using TikTok as the foreground application and aggregating several threads that contribute to frame rendering on the big cluster. The results, shown in Fig. 4, indicate that the placement of QoS-related threads on the big cluster can lead to improved frame rate and energy efficiency. This highlights the significant impact that scheduling can have on governing and the importance of considering both components in the optimization process.

In order to gain a deeper understanding of the interaction between scheduling and governing, we place UI Thread, Render Thread, and SurfaceFlinger on different clusters, and traverse to obtain the frequency point that meets QoS and has minimum power consumption. The results, as shown in Table 1, reveal that different scheduling strategies result in varying optimal frequencies and power consumption. For example, as illustrated in the first two lines, when the threads are mainly scheduled in the big cluster, the frequency of the LITTLE cluster can be reduced to the minimum, while the frequency of the big cluster can be mainly increased to provide computing capacity. A similar pattern is observed when the threads are scheduled on the LITTLE clusters. If we can accurately characterize the impact of scheduling and governing, we can further reduce power consumption while meeting the QoS.

Observation 3: mutually decoupled scheduling and governing still have room to further optimize QoS and power consumption. The scheduling information of threads associated with the frame rendering can be provided to the

governor so that the governor can consider it when adjusting the frequency. Simultaneously, the frequency information of the cluster can also be shared with the scheduler to allocate the best cores to threads. This synergy between the two will lead to further power savings and performance enhancements.

3 Design Overview

3.1 Challenge and Key Design Choice

The decouple issues introduce modifications to the system's scheduler and governor modules while adding a coordinator module to coordinate behavior between the two. Therefore, we face three unique challenges.

Challenge 1: Unlike the conventional governors that only take into account utilization, our design must also consider QoS and system conditions such as CPU, memory, and IO. The system environment is intricate and challenging to model, making it difficult to determine the optimal frequency for a given situation.

Design 1: **A Proximal Policy Optimization (PPO) based governing approach** that trains an intelligent agent based on QoS and system conditions. The actor and critic neural networks take the system state, application characteristics, and QoS as inputs and adjust the cluster frequency to minimize power consumption while satisfying the QoS. The overhead of utilizing the PPO algorithm is relatively small compared to the benefits of reduced power consumption, making the PPO-based algorithm suitable for power management, which will be further discussed later.

Challenge 2: The QoS of upper-level applications is challenging to transmit to the scheduler in the kernel. The scheduler is unable to detect QoS dissatisfaction and wastage in power consumption.

Design 2: **A Finite State Machine based scheduling algorithm** which divides the system state into three parts using task-clock information and Instructions Per Second (IPS) to reflect the QoS of the upper layer. The scheduler then transfers states that can not meet the QoS and result in unnecessary power consumption into the state where the computing capacity precisely meets the desired QoS.

Challenge 3: The interconnections between scheduling and governing are intricate and dynamic. Once threads have been scheduled, any changes in the computing capacity needed by the cluster will impact the governing process. In turn, when the governor adjusts the cluster frequency, it will affect the states of the scheduler and cause state transitions. There is currently no accurate model for quantifying the interplay between these two processes due to the dynamic nature of thread placement, computing requirements, and system state.

Design 3: **An Expert Fuzzy Control based coordinating mechanism** aimed at coordinating the impact between scheduling and governing. The EFC method utilizes utilization and priority information provided by the scheduler as inputs, and outputs governing hints for the governor. Additionally, it records frequency information from the governor and passes it to the scheduler when invocated.

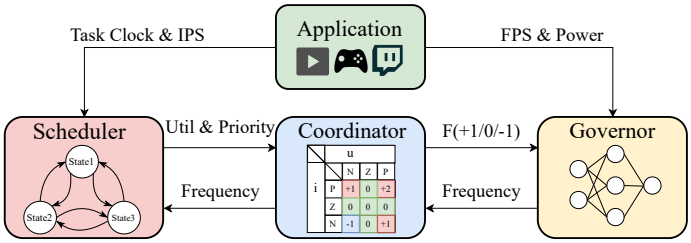


Fig. 5: Architecture of Orthrus.

3.2 Orthrus Architecture

We now present the design of Orthrus, a co-optimization approach that makes scheduling and governing aware of QoS and jointly minimizes power consumption while satisfying QoS. Fig. 5 illustrates *three* key components of Orthrus. The **Governor** module learns application characteristics and system workload fluctuation to control the frequency of clusters through deep reinforcement learning (§4.1). The **Scheduler** module assigns QoS-related threads priorities and schedules them onto different clusters based on their performance requirements and priorities through finite state machine (§4.2). This leads to a different distribution of threads, providing the potential to improve performance and reduce power consumption. To fully exploit this potential, a **Coordinator** module is introduced, which coordinates the scheduler and governor through expert fuzzy control (§4.3). The coordinator receives priority and utilization information on each cluster from the scheduler, and fine-scales the frequency through a set of rules based on a human expert's knowledge.

The workflow of Orthrus includes three steps: 1) At fixed time intervals, the governor monitors the system status and adjusts the frequency, and then transmits information about the frequency to the coordinator. 2) Triggered by the event where the system deviates from the moderate state, the scheduler runs the scheduling algorithm to migrate threads, and then updates the cluster utilization and priority to the coordinator. 3) The coordinator exchanges information when the governor or scheduler is running, passing frequency to the scheduler and governing hints to the governor.

4 Orthrus Design

4.1 A Proximal Policy Optimization Based Governing Approach

4.1.1 Problem Formulation

We define $Q(t)$ to represent the performance of the application as QoS at time t. In this context, t refers to a specific discrete time step or sampling instance in the operational process of the application. For video or gaming applications, $Q(t)$ is usually a function of the frame rate, which is used to ensure that the frame rate exceeds a certain level [12]. Let $P(t)$ denote the function of the instantaneous power consumption of the CPU at time t. $Q(t)$ and $P(t)$ represent performance and power consumption respectively, and are normalized values (we will discuss it in Section 4.1.3), so they can be used for comparison. For scalability, we assume that there are multiple clusters with different computing capabilities and energy efficiency on heterogeneous CPUs.

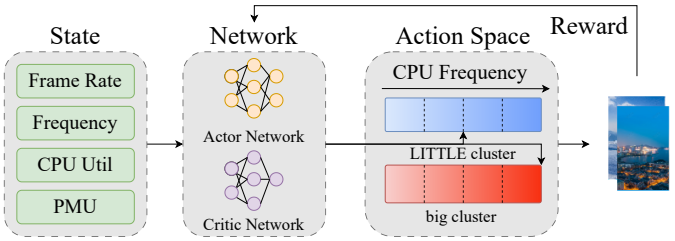


Fig. 6: The workflow of PPO-based governing approach.

Let C denote the set of CPU clusters. Let $f_c \in F_c$ denote the frequency of cluster c, where F_c denotes the available frequency points of cluster c [3]. Let π denote the set of available policies controlling the different CPU cluster frequency from $t = 1$ to $t = T$, $(f_c(1), ..., f_c(T))$, which is the key determinant. Then, the problem of maximizing $Q(t)$ and minimizing $P(t)$ on average for T can be formulated as **QoS-aware Governing Optimization (QGO) Problem**:

$$\max_{\pi} \frac{1}{T} \sum_{t=1}^{T} \{Q(t) - P(t)\} \tag{1}$$

$$s.t. f_c(t) \in F_c, \forall t \; \forall c \tag{2}$$

where $f_c(t)$ denotes the configured frequency of cluster c at time t. Constraint (2) is limited by the hardware. By subtracting $P(t)$ from $Q(t)$, we illustrate the objective of enhancing performance while reducing power consumption.

4.1.2 Problem Analysis

The complexity and variability of system conditions pose a challenge in solving the **QGO** problem. This results in the need for dynamic frequency adjustments to optimize the performance and power of the system. For instance, CPU, memory, and IO resources in the system will affect the QoS of foreground applications to varying degrees and interfere with each other. It is thus difficult to model the relationship between system conditions and cluster frequency.

We thus seek a learning-based approach. Our problem is intrinsically a control optimization problem, and the solution falls into the Reinforcement Learning (RL) algorithm. At a high level, RL is commonly used for a control problem where the control actions directly impact future states. Moreover, it interacts with the environment and learns the optimal decisions based on feedback without relying on an explicit model, which matches the governing problem.

When selecting an appropriate RL algorithm, it is critical to consider factors such as resource constraints of mobile devices, the complexity of the learning process, and the difficulty of parameter tuning. In this context, we adopt proximal policy optimization, a state-of-the-art actor-critic reinforcement learning algorithm [32] in Orthrus. The traditional actor-critic is a policy gradient method, and the principle is as follows:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log_{\pi_\theta}(\alpha_t|s_t) \hat{A}_t \right] \tag{3}$$

3. For example, the available frequency points for the LITTLE cluster of Google Pixel 3 are {300000 403200 480000 576000 652800 748800 825600 902400 979200 1056000 1132800 1228800 1324800 1420800 1516800 1612800 1689600 1766400}(the units are all in kHz).

Algorithm 1: PPO-based Governing, Actor-Critic Style

Input: Initial policy $\pi_{\theta_{old}}$
Output: Optimal policy $\pi_{\theta_{new}}$
for *interation=1,2,...* **do**
  for *actor=1,2,...,N* **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  Optimize surrogate $L_t(\theta)$, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
$\pi_{\theta_{new}} \leftarrow \pi_{\theta_{old}}$
return $\pi_{\theta_{new}}$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at time t, Here, the expectation $\hat{\mathbb{E}}_t [...]$ indicates the empirical average over a finite batch of samples [33]. The critic provides guidance to the actor by estimating the expected cumulative reward given a certain state and action. The actor then adjusts the policy to increase the probability of selecting actions that lead to higher reward estimates from the critic. In our approach to solving the **QGO** problem, the stochastic policy π_θ represented in Equation (3) is tailored to define the probability of selecting discrete frequency actions based on the current system state. The estimator $\hat{A}_t$ quantifies the benefit of choosing a particular action over others in terms of achieving the desired QoS while minimizing power consumption.

However, the high variance in estimating the gradient of the policy of this traditional policy gradient method is challenging to control. This can lead to slow convergence and instability in the training process. Meanwhile, it can get trapped in local optima and fail to converge to the optimal policy. Therefore, we used a better-performing method - PPO. The main objective of PPO is:

$$L^{clip}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right] \tag{4}$$

where ϵ is a hyperparameter, $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$ modifies the surrogate objective by clipping the probability ratio $r_t(\theta)$, which removes the incentive for moving $r_t(\theta)$ outside of the interval $[1-\epsilon, 1+\epsilon]$. Let $r_t(\theta)$ denote the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{5}$$

This modification is particularly crucial for the **QGO** problem as it ensures that the policy updates are not only conducive to learning effective governing strategies but also preserve the stability necessary for deployment on resource-constrained mobile devices. The probability ratio $r_t(\theta)$ in Equation (5) ensures that the updates to the policy are proportional to the change in the probability of taking the current action under the new policy compared to the old. Furthermore, PPO makes efficient use of the collected data by using multiple epochs of mini-batch updates, which ensures that the agent learns as much as possible from the experience it has gathered. We present more details in Algorithm 1.

Below, we will introduce the state, action, and reward we design and the ideas behind them.

TABLE 2: State, Action and Reward.

State $s(t)$	$f_c(t), u_c(t), q(t), p(t), cm(t), pf(t)$
Action $a(t)$	$f_c(t)$
Reward $r(t)$	$Q(t) - P(t)$

4.1.3 State, Action & Reward Design

As Fig. 6 shows, an agent (neural networks) reads the environment (mobile system) states and takes actions (CPU frequency) based on the probability distribution given by the policy $\pi_\theta(S_t, a_t)$. After executing each action, the environment will give us corresponding feedback, which will be used as observations of the environment for the agent to make the next decision. The agent's objective is to learn an optimal policy that maximizes the expected return, i.e., performance per watt, by taking optimal actions. Orthrus solves the **QGO** problem with PPO, which adapts to dynamic system workloads while ensuring performance requirements and minimizing power consumption. Here, we describe how we customize PPO for Orthrus by redefining state, action, and reward.

State. Our states are defined by a tuple containing six parameters, as shown in Table 2. In addition to monitoring power consumption $p(t)$ and frame rate $q(t)$ of foreground application at time t, we monitor CPU frequency $f_c(t)$ and CPU utilization $u_c(t)$ for each cluster c. Moreover, we monitor some Performance Monitoring Unit (PMU) events to reflect the characteristics of the application. After performing correlation analysis on multiple PMU events, we integrate PMU events $(cm(t), pf(t))$ to reflect the consumption of memory and I/O resources of the application. Specifically, we use **LLC_CACHE_MISS** $(cm(t))$ which counts the number of Last-Level-Cache (LLC) misses in memory, and **PAGE_FAULTS** $(pf(t))$ which counts the number of page faults for I/O resource.

Action. We define the combination of frequency points $f_c(t)$ of each cluster c as actions, which are performed using the clipping mechanism. The clipping mechanism limits the updates to the policy, ensuring that they remain within a reasonable range and preventing excessive exploitation. During the exploration phase, the agent takes uniform random actions across the frequency range to collect training data. In the exploit phase, the action $a(t)$ that maximizes the reward will be chosen to converge the model.

Reward. We propose a well-designed reward function as shown in Table 2. The reward function consists of $Q(t)$ and $P(t)$. QoS reward is designed with the following considerations. 1) No additional reward should be given for exceeding the target frame rate G. We use the min function to avoid the agent constantly increasing the frequency to get a higher reward. 2) Below a certain frame rate, there are severe penalties. We introduce Q_a to indicate the QoS the user can tolerate. When the QoS is less than this value, the reward becomes negative. $P(t)$ and $Q(t)$ are defined as follows.

$$Q(t) = \alpha \min\left(\frac{q(t) - G}{G - Q_a} + 1, 1\right) \tag{6}$$

$$P(t) = \beta \frac{p(t)}{M} \tag{7}$$

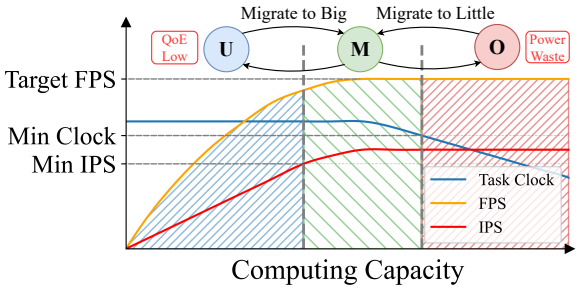


Fig. 7: The vary of IPS and Task Clock when computing capacity growing.

where M denotes the maximum power consumption of the mobile device, which is used to normalize the reward of power consumption, as the scale of $p(t)$ is in the hundreds or thousands. α and β are trade-off weights to balance performance and power consumption according to user preferences, with value $0 \leq \alpha, \beta \leq 1$.

4.2 A Finite State Machine Based Scheduling Algorithm

There are three states in the system: 1) The computing capacity is **Underprovisioned (U)** to satisfy QoS. 2) The computing capacity is **Overprovisioned (O)**, which exceeds the requirements for QoS but wastes power consumption. 3) The computing capacity is **Moderate (M)** for both QoS and power consumption. Fig. 7 illustrates the variation of *task-clock*, *IPS*, and *FPS* of the foreground application thread. Task-clock reports a clock count specific to the task that is running, which can be regarded as the execution time of the thread. Computing capacity indicates the amount of computing power that the hardware can provide for tasks. We control changes in computing capacity by adjusting frequency and thread migration. As the computing capacity increases, the IPS of the thread gradually increases, and the assigned task-clock remains unchanged due to CPU resource competition. At this time, the IPS is less than *Min IPS* and the computing capacity is deemed **underprovisioned** for the QoS requirements. When IPS is not less than *Min IPS*, the instructions to be executed gradually saturate, thus the task-clock allocated to threads decreases with more computing capacity. When the task-clock is below *Min clock*, the computing capacity is considered **overprovisioned**. Therefore, the scheduler can distinguish the three states through the *IPS* and *task-clock* of the foreground thread. To meet QoS and save power consumption, the scheduler should transfer from either of the first two states to the **moderate** state through migrating threads.

To this end, we develop a **Finite State Machine based Scheduling Algorithm** as shown in Algorithm 2. First, we calculate the priority i_k of each thread among n threads (Line 1-3) based on Eq. (8).

$$i_k = tc_k / \sum_{k=1}^{n} (tc_k) \tag{8}$$

If the thread's task-clock accounts for more, it means that its contribution and criticality to the frame are higher, which should be given high priority. Then we arrange cluster C in positive order by computing capacity (Line 4). After getting the index c of cluster where the thread is located (Line

Algorithm 2: FSM-based Scheduling

Constant: $MIPS_k$: Min IPS , MTC_k: Min Task Clock
Input: C: Cluster set, K: Threads number,
tc_k: Threads Task Clock, ips_k: Threads IPS
Output: $u_c(t+1)$: Utilization of cluster c at time $t+1$
$i_c(t+1)$: Priority of cluster c at time $t+1$
$f_c(t+1)$: Frequency of cluster c at time $t+1$

/* Calculate threads priority based on tc_k */
1 **for** $k \leftarrow 1$ **to** K **do**
2 $i_k = tc_k \ / \ \sum_{k=1}^{K}(tc_k)$
3 Sort threads based on priority i_k with reverse order
4 Sort C based on computing capacity with positive order
/* Scheduling based on tc_k and ips_k */
5 **for** $k \leftarrow 1$ **to** K **do**
6 c $\leftarrow get\_cluster\_of\_thread(k, C)$
7 **if** $ips_k \leq MIPS_k$ **then**
/* Computing Capacity Underprovision */
8 **if** *not exist stronger cluster* **then**
9 $f_c(t+1) \leftarrow min(f_c(t)^{max}, MIPS_k/ips_k * f_c(t))$
10 **else**
11 **if** *exists freq* f *s.t.* $f/f_c(t) > MIPS_k/ips_k$ **then**
12 $f_c(t+1) \leftarrow f$
13 **else**
14 Migrate thread to cluster $c+1$
15 Update
$u_c(t+1), u_{c+1}(t+1), i_c(t+1), i_{c+1}(t+1)$
16 **else if** $tc_k \leq MTC_k$ **then**
/* Computing Capacity Overprovision */
17 **if** *not exist weaker cluster* **then**
18 $f_c(t+1) \leftarrow max(f_c(t)^{min}, tc_k/MTC_k * f_c(t))$
19 **else**
20 **if** *exists freq* f *s.t.* $f/f_c(t) < tc_k/MTC_k$ **then**
21 $f_c(t+1) \leftarrow f$
22 **else**
23 Migrate thread to cluster $c-1$
24 Update
$u_c(t+1), u_{c-1}(t+1), i_c(t+1), i_{c-1}(t+1)$
25 return $u_c(t+1), i_c(t+1), f_c(t+1)$ of all clusters

6), we detect the **U** and **O** states based on task-clock and IPS (Line 7-24). If the thread IPS is less than *Min IPS*, we satisfy the QoS by scheduling threads to a stronger cluster, which means a cluster with greater computing capacity, or scaling up the frequency (Line 7-15). Conversely, if the IPS of threads is sufficient and the task-clock drops below the *Min Clock*, we schedule threads to a weaker cluster, implying a cluster with lesser computing capacity, or scale down the frequency to save power consumption (Line 16-24). We prioritize the behavior of governing to satisfy QoS. This is because the scheduling overhead (∼2ms) tends to be higher than the governing (∼500µs) due to cache affinity, etc [11] [34]. Lastly, we return the recalculated cluster priority and utilization information needed by the coordinator.

4.3 An Expert Fuzzy Control Based Coordinating Mechanism

The interplay between governing and scheduling is substantial. The actions taken by the governing algorithm can impact the state transition of the scheduling algorithm, while scheduling can affect the perception of the system conditions by the governing algorithm. To address this, we propose a coordinating mechanism that utilizes expert fuzzy control.

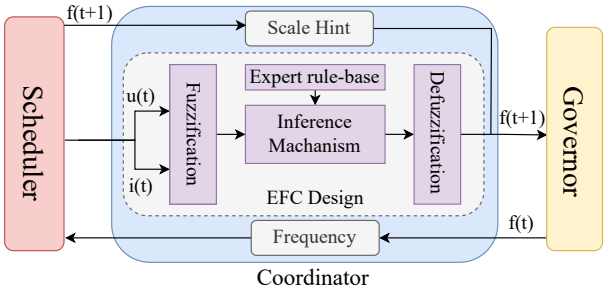


Fig. 8: The coordinator design.

The coordinating mechanism is responsible for passing governing hints when the scheduling algorithm changes state so that the governor can respond to system state changes in a timely manner, and transmitting the frequency information when the governing algorithm changes the frequency to make the scheduling algorithm perform better. As Fig. 8 depicts, the coordinator consists of three modules: EFC Design and Scale Hint for the governor and Frequency module for the scheduler.

Scheduling Impact on Governing. When threads are scheduled, the loads on clusters change dynamically. For instance, when a thread is moved from cluster A to cluster B, the load on A is reduced, while the load on B is increased. In this situation, if cluster A continues to maintain its previous frequency, it may result in power wastage, and if cluster B is left unchanged, the QoS may be compromised. The RL Governor will take several cycles to detect changes in the load. Therefore, it is essential for the scheduler to provide the governor with direct information to enable a prompt and accurate response. Since it is difficult to model this dynamism when we need to consider both utilization and priority changes of running threads, we use EFC to coordinate the interaction of scheduling and governing. EFC can exploit model-independent fuzzy control techniques to address the issue of lacking accurate performance-power models due to high workload dynamics [35].

EFC aims to translate expert knowledge into control rules, which are defined using linguistic variables corresponding to the two inputs, $u(t)$ and $i(t)$. The fuzzification process converts the numeric inputs into linguistic values such as NL (negative large), NS (negative small), ZE (zero), PS (positive small) and PL (positive large). The rules are in the form of If-Then statements. For example, if $u(t)$ is PL and $i(t)$ is PL, the adjustment in CPU frequency is $f(+1)$. The rationale behind this rule is that cluster frequency should increase since the utilization and priority of threads running on this cluster grow. Similarly, various rules are designed to determine the CPU frequency based on the utilization and priority in the cluster. The Scale Hint aims to make a balance between fuzzy control and formula calculation by Algorithm 2. It compares two frequencies and selects a reasonable one. When the frequency is scaled down, the maximum of the two is selected. When the frequency is scaled up, the minimum of the two is selected.

Governing Impact on Scheduling. Adjusting the cluster frequency by the governor may cause a change in computing capacity, which will result in a state transition from Moderate to other states and impact the scheduling

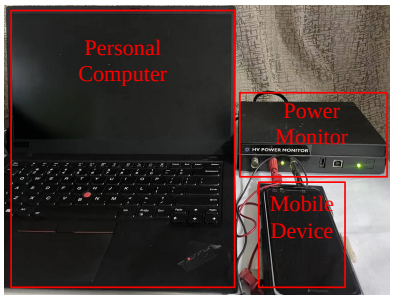


Fig. 9: Experimental equipment of Orthrus.

process. During state transitions, governing takes precedence over scheduling for the scheduler. Therefore, accessing the current frequency and adjustable frequency range of the cluster is vital to finding a frequency that meets the QoS requirements and ensures energy efficiency. Since the governor and scheduler have different running times, the governor should provide frequency-related information to the Frequency module at runtime via shared memory. The module then uses this information to assist the scheduler in making optimized decisions.

5 Implementation

We implement Orthrus with Python 3.7 and Java 11.0.13 on Google Pixel 3. The implementation of Orthrus includes four steps: state collection, neural network deployment, frequency setting, and thread scheduling. We develop the Orthrus in GitHub, and we plan to release the codes as open sources.

State Collection. All needed system information can be obtained by *sysfs* [36] provided by Linux or some commands provided by the Android kernel. We read the specified file from *sysfs* to get CPU frequency and call *linux_perf_event* syscall [37] to get the PMU event. Frame rates can be collected by *dumpsys SurfaceFlinger* command and power consumption can be measured with Monsoon Power Monitor [38].

Neural Network Deployment. The neural network can be implemented by TensorFlow Lite [39], an open-source library for machine learning on mobile devices. As described in Algorithm 1, the PPO model interacts with the environment by controlling the frequencies and then collects data (status, actions, and rewards) to update the parameters of the network. This process is repeated until the preset number of iterations is completed. We first train our PPO Model on the computer, then convert it to a TensorFlow Lite format file and deploy it on an Android device.

Frequency Setting. Implementing our governing policy at the user level is allowed by Android. Android allows privileged users to set CPU frequency after setting governor as *userspace* (writing *userspace* to */sys/devices/system/cpu/cpufreq/policyX/scaling_governor* file). Then, we can control the frequency of the CPU by setting the file interface (*/sys/devices/system/cpu/cpufreq/policyX/scaling_setspeed*).

Thread Scheduling. We use the *taskset* command to control which core a thread runs on. The pid of SurfaceFlinger remains constant without reboot and can be gotten by running *ps* command once. The tids of foreground application change every time the application is restarted, which can be gotten dynamically by *dumpsys windows* and *ps* command.

Finally, we implement Orthrus as a self-start service in the Android system. When the foreground application starts, it will automatically obtain relevant information such as pid, frame rate, etc., and perform scheduling and governing.

TABLE 3: Baselines specifications.

Baseline	Module		
	Scheduler	Governor	Coordinator
Android def	EAS	Schedutil	×
SmartBalance	Anneal	×	×
zTT	×	DQN	×
AdaMD	Adaptive Mapping	Bin Classfication	×
Orthrus	FSM	PPO	EFC

TABLE 4: Applications and background workload.

Foreground App	Target QoS	Description	Stress
TikTok	60 FPS	Video rendering	S1: No application
Genshin	30 FPS	Mobile game	S2: Download files & Load Image
Instagram	60 FPS	Social Media	S3: Zip/Unzip files

6 Experiment

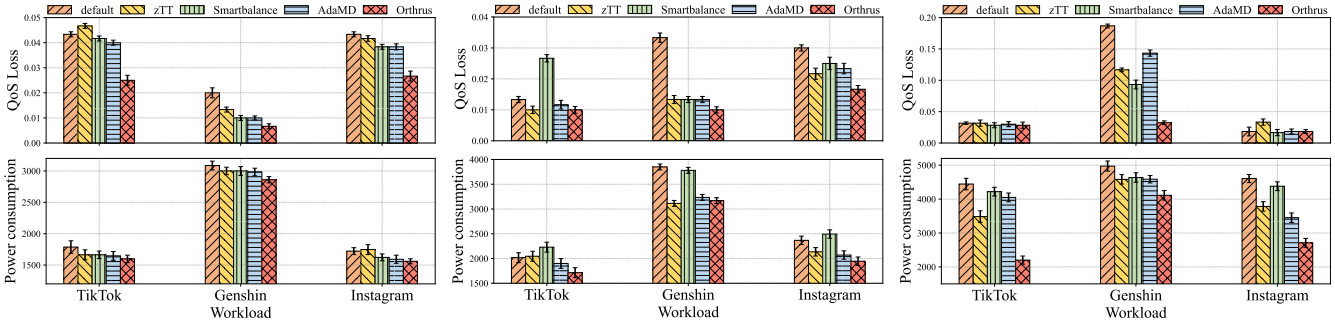
6.1 Methodology

Testbed. As illustrated in Fig. 9, our experimental equipment consists of mobile devices running the Orthrus program, a Monsoon Power Monitor measuring power consumption, and a PC machine running the monitoring program. We evaluate Orthrus on a Google Pixel 3 which is equipped with Snapdragon 845 SoC, 4GB memory, and runs Android 11. The SoC has 4 Kryo 385 Gold cores (big cluster) and 4 Kryo 385 Silver cores (LITTLE cluster).

Workloads. We use three popular applications as the foreground application, and three common applications as the system load, including file downloading (IO-Stress), image loading (Memory-Stress) and file compressing (CPU-Stress), to comprehensively test Orthrus performance. The details of experimented applications are summarized in Table 4.

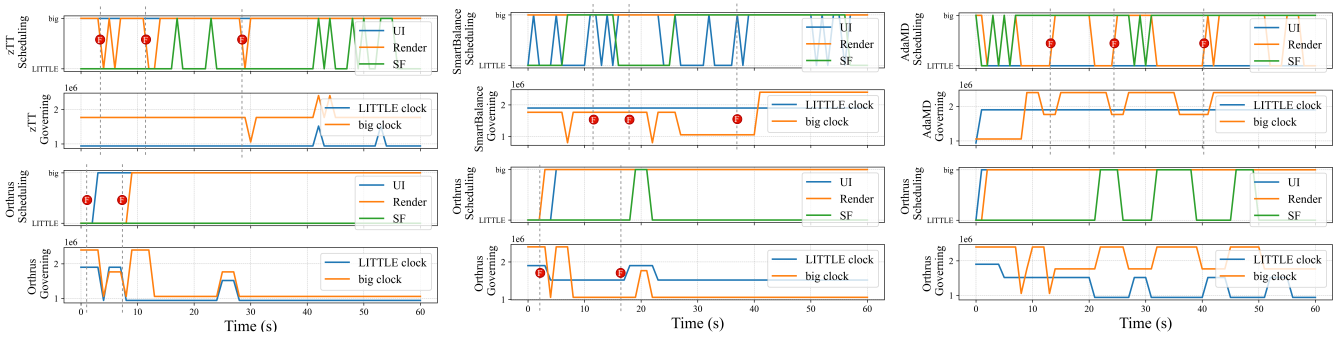
To comprehensively evaluate the effectiveness of Orthrus, we performed assessments under three distinct operational scenarios:

- No Background Application Running (**Light Workload**): This scenario reflects a minimal system load with no additional background applications running. It simulates the condition where the primary application operates in isolation, facing minimal competition for system resources.
- Running with Background Privileges (**Medium Workload**): For a more moderate level of workload, we introduce specific background applications performing tasks such as file downloading and image loading. This scenario mimics a common real-world usage pattern where the system concurrently handles secondary tasks.
- Running with Foreground Privileges (**Heavy Workload**): To simulate a high-demand environment, we



(a) *light workload* (b) *medium workload* (c) *heavy workload*

Fig. 10: Performance loss and power consumption of our work and four baselines in three situations.



(a) *Orthrus Vs zTT* (b) *Orthrus Vs Smartbalance* (c) *Orthrus Vs AdaMD*

Fig. 11: Analysis of scheduling and governing of Orthrus and baselines, where the red circles are the moments of QoS loss.

use CPU-intensive tasks like file compression and decompression running in the foreground. This setup is designed to emulate a realistic user scenario where multiple applications, including those requiring significant CPU resources, are active simultaneously.

These three scenarios are crafted to encapsulate a comprehensive spectrum of system conditions, mirroring the diverse usage patterns encountered in typical mobile device operations.

Baselines. We compare Orthrus with four state-of-the-art works. Table 3 shows the specification of these works.

- *Android default* uses *EAS* scheduling algorithm and *schedutil* governing scheme to improve performance and optimize power consumption.
- *SmartBalance* [10] uses the scheduling strategy with the best energy efficiency according to the IPS and power of different clusters of tasks.
- *zTT* [12] adopts a reinforcement learning algorithm to optimize governing, which aims to meet QoS and reduce power consumption.
- *AdaMD* [11] designs an adaptive thread-to-core mapping for each performance-constrained application and a DVFS algorithm to handle workload variations.

Unless otherwise specified, the scope of the Orthrus scheduling algorithm includes UI Thread, Render Thread, Surfaceflinger, and Binder Thread. The time interval of the governing algorithm is 1 second (we will discuss it in Section 6.4), and the load used is light workload. The power consumption on mobile devices is experimentally measured to be hardly more than 5000 mW, so M is preset to 5000. α and β are both set to 1 to indicate that QoS and power are equally valued, which is also the demand of most users.

Metrics. We use two metrics to evaluate the performance of Orthrus and the baselines:1) *QoS Loss* quantified the satisfaction degree of service quality. Since different applications have different target frame rates, QoS Loss is used as an indicator. QoS loss is computed as $\sum_{t=1}^{T} \frac{G-q(t)}{GT}$; and 2) *Power Consumption* shows the effect of Orthrus on energy saving.

6.2 Overall Performance

6.2.1 Results of Light Workload

Fig. 10(a) shows the QoS loss and power consumption of *default*, *zTT*, *Smartbalance*, *AdaMD* and Orthrus. We observe that Orthrus suffers the least QoS loss compared to the baselines, since it provides strict QoS guarantees while saving energy. Under three applications, the average power consumption is reduced by 8%, 5%, 3.5%, and 2.5% compared to *default*, *zTT*, *Smartbalance* and *AdaMD*, respectively. When no background applications are running, there is less room to reduce power consumption.

During experiments, we found that under light load, the clusters under each method run at a low frequency and the scheduling plays an important role in overall performance. As shown in Fig. 11(a), we compared the scheduling and governing operations of Orthrus and *zTT*. *zTT* adopts default scheduling policy *EAS*, which always seeks the core with the least power consumption for threads without considering QoS. Therefore, the three red circles in *zTT* all indicate that QoS-related threads are frequently migrated to the LITTLE cluster due to lower power consumption. However, the frequency of the LITTLE cluster is low and

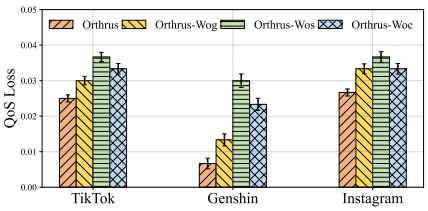


Fig. 12: QoS loss for component analysis.

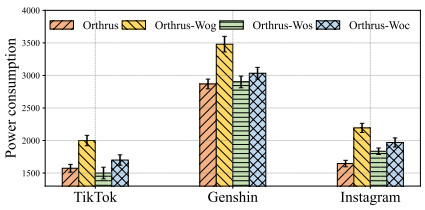


Fig. 13: Power consumption for component analysis.

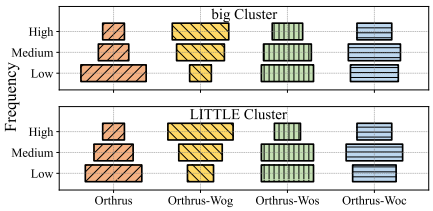


Fig. 14: Frequency distribution of different methods.

cannot provide sufficient computing capacity, resulting in a QoS decline. In Orthrus, the scheduler detects from the beginning based on *IPS* and *task-clock* that the LITTLE cluster cannot meet the computing requirements of two threads, and migrates them onto the big cluster. Subsequently, the scheduler observed that the system was always in the moderate state, and did not migrate threads frequently, ensuring that the target QoS could always be met.

6.2.2 Results of Medium Workload

As shown in Fig. 10(b), Orthrus is more effective when running download files and load images as background applications. With the smallest QoS loss, Orthrus reduces the power consumption of 17%, 6.4%, 20.7% and 5.3% on average in the three applications compared to *default*, *zTT*, *Smartbalance* and *AdaMD*. *Smartbalance* consumes more power and has more loss in QoS. This is because *Smartbalance* adopts the default governing policy *schedutil*, which adjusts the cluster frequency based on utilization without considering QoS. As shown in Fig. 11(b), background application runs on the LITTLE cluster most of the time, thus the LITTLE cluster of *Smartbalance* always stays at the highest frequency due to high utilization, which contributes little to QoS while wasting much power. The three red circles in Fig. 11(b) indicate that in *Smartbalance*, the cluster frequency does not respond when QoS-related threads are scheduled. In Orthrus, the first red circle indicates that after the thread migration, the governor adjusted the LITTLE cluster to a moderate frequency, thus saving a lot of power consumption. The second red circle shows that the thread migrates to the big cluster when the system fluctuates. At this time, the governor quickly adjusts the frequency according to the system status and scheduler information, and returns to the previous frequency after the thread migrates back to the LITTLE cluster in time.

6.2.3 Results of Heavy Workload

Fig. 10(c) shows the QoS loss and power consumption when running heavy workload. Compared with the baseline, Orthrus still achieves better QoS and lower power consumption by 35.7%, 23.8%, 31.8% and 25.3% compared to *default*, *zTT*, *Smartbalance* and *AdaMD*, respectively. In this case, we find that *AdaMD* is not as effective as before, with more wasted power consumption and QoS degradation. So we analyze the scheduling and governing behavior of *AdaMD* and Orthrus, as shown in Fig. 11(c). The results indicate that *AdaMD* lacks a collaborative optimization of scheduling and governing, resulting in a delayed governor response after each scheduling event. The three red circles are QoS drops caused when the frequency of the big cluster is not increased after the thread is scheduled onto it. Additionally, the frequency of the LITTLE cluster is not adjusted in accordance with the scheduling, leading to a waste of power consumption. In Orthrus, every time a QoS-related thread is scheduled, the coordinator will convert the scheduling information into the frequency of the cluster and pass it to the governor to ensure timely governing. At the same time, the frequency on the cluster will also be fed back to the scheduler to ensure reasonable scheduling. Therefore, as Fig. 11(c) depicts, the scheduling and governing of Orthrus are co-optimized to ensure that the two can manage power jointly while perceiving QoS.

6.3 Component Analysis

In this section, we explore the components of Orthrus to better understand their contribution to system performance. We implement three subdivided versions of Orthrus to study the contribution of each component: 1) *Orthrus-Wog* has the FSM-based scheduler but does not enable the PPO-based governor, 2) *Orthrus-Wos* has the PPO-based governor but does not enable the FSM-based scheduler, and 3) *Orthrus-Woc* has the FSM-based scheduler and PPO-based governor, but does not enable the coordinator to exchange information between them. Fig. 12-14 shows the comparison results on three applications.

For the governor, *Orthrus-Wog* has a higher QoS penalty than Orthrus, and consumes 20.8% more power than Orthrus on average. This is because when governing is unaware of QoS, the cluster will often erroneously run at a higher frequency due to the single indicator of CPU utilization. As shown in Fig. 14, both clusters of *Orthrus-Wog* run at the highest frequency most of the time. Power consumption is wasted when the cluster is mostly running threads unrelated to QoS. For the scheduler, the QoS loss of *Orthrus-Wos* is much higher than Orthrus, but the power consumption is similar. This is because the scheduler prefers to use governing rather than scheduling to transfer state, and the governing hints given to the governor are usually more aggressive than the DRL-agent. Thus, Fig. 14 shows that *Orthrus-Wos* is more in the middle and low frequency than *Orthrus-Wog*. However, if there is no perception of QoS, the scheduler will not transfer to the moderate state that satisfies QoS, but strives for the lowest power consumption, which will cause serious QoS degradation. For the coordinator, the frequency of *Orthrus-Woc* is only controlled by the governor, which is roughly consistent with the distribution of *Orthrus-Wos*. However, its QoS and power consumption are inferior to Orthrus due to the lack of synergy between the two. These results indicate the importance of three components. Orthrus is able to combine the advantages to achieve better performance.

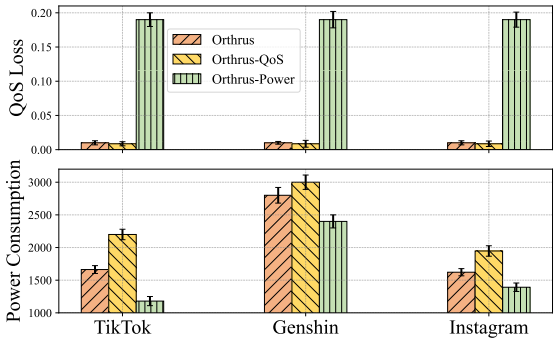


Fig. 15: QoS loss and power consumption of Orthrus, Orthrus-QoS and Orthrus-Power on various applications

TABLE 5: The impact of time intervals of governing.

Interval(s)	FPS	Power (mW)
0.1	58.6	1711
0.5	58.9	1632
1	**59.1**	**1597**
2	58.5	1560
3	58.1	1540

6.4 Sensitivity Analysis

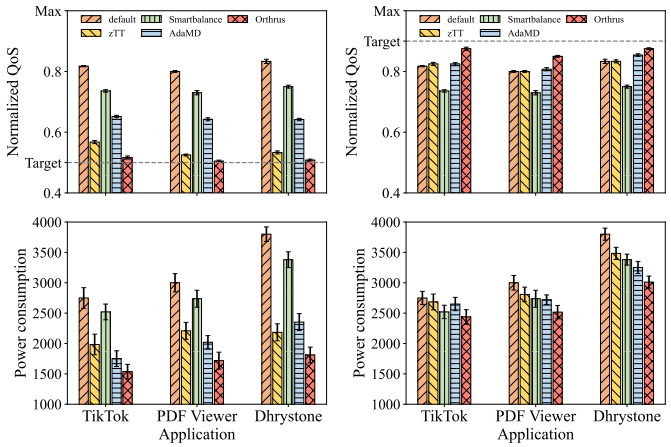
In this section, we investigate the effect of system parameters on Orthrus performance and power gains.

α and β in reward function of governing approach. As mentioned in Section 4.1, α and β can be used as trade-off weights to balance performance and power consumption. When the value of α increases and β decreases, the RL agent prioritizes ensuring user experience. Conversely, when α decreases and β increases, the RL agent prioritizes reducing power consumption. We have named the former *Orthrus-QoS* ($r(t) = Q(t), \alpha = 1, \beta = 0$), and the latter *Orthrus-Power* ($r(t) = P(t), \alpha = 0, \beta = 1$). As shown in Fig. 15, we find that under light workload conditions, Orthrus and *Orthrus-QoS* have similar QoS, but power consumption is much lower for Orthrus. This is because each application has a target frame rate, and *Orthrus-QoS* tries to meet QoS by using high frequencies as much as possible, ignoring the fact that the low frequency can also meet QoS with lower power consumption. The power consumption of Orthrus is higher than that of *Orthrus-Power*, but its QoS is much higher too. This is because *Orthrus-Power* tries to achieve low power consumption by using low frequencies as much as possible, but these frequencies cannot satisfy QoS.

Time interval of governing approach. The time interval of the governing algorithm plays a crucial role in its performance. As demonstrated in Table 5, if the interval is too long, the algorithm may be slow in responding to changes in system conditions. On the other hand, if the interval is too short, it can result in excessive delay and power consumption overhead. Therefore, we opted to use a time interval of 1 second for the governing algorithm.

6.5 Runtime Overhead

We analyze the overhead of Orthrus from three aspects: time, memory usage, and power consumption. In terms of time, the delay of Orthrus has the following components: 1) collecting system and application information, 2) scheduling algorithms and governing agent inference, and 3) performing scheduling and governing actions. These delays are all affected by the computing capacity, ranging from 35 ms to 174 ms. Since the interval used by the algorithm is 1 s, the latency is acceptable. In addition, the delay mainly comes from the third step of running the command. If it is subsequently implemented in the kernel, the latency will be further reduced. For memory usage, the service requires 42 MB of memory, which is 1% of all memory (4 GB for Google Pixel 3). If using a more advanced mobile phone, the proportion usage of memory will be smaller, which will not affect the user experience. For power consumption, running Orthrus consumes an average of 180 mW. Since the power consumption of the whole system is between 1500 mW and 5000 mW, Orthrus can help reduce the power consumption by more than 180 mW compared to *default* even under the lightest load. Therefore, the power consumption overhead of Orthrus is also negligible.



(a) *Target QoS = 0.5 Max QoS.* (b) *Target QoS = 0.9 Max QoS.*

Fig. 16: QoS and power consumption of Orthrus with different QoS metrics and different target QoS.

6.6 QoS Adaptability

In this section, we aim to assess Orthrus's ability to learn applications and adapt to diverse performance requirements by selecting various QoS metrics, including frame rate, latency, and IPS, along with different target QoS values. First, we define *Max QoS* as the QoS provided by the maximum computing capacity (all threads of the application run at the highest frequency of the big cluster) of the system under light workload. Then, we run applications with different QoS metrics in three scenarios. We incorporate two additional applications of different types, namely *Adobe PDF Viewer* and *Dhrystone* [40]. For *PDF Viewer*, the latency in processing time is a crucial factor influencing user experience [41], hence chosen as the QoS metric. As for the *Dhrystone* benchmark, we observe its IPS as the metric for its QoS. Finally, we adjusted the target QoS to 0.5 *Max QoS* (easily reachable) and 0.9 *Max QoS* (not reachable) to observe the performance of Orthrus.

As depicted in Figure 16(a), when targeting a QoS of 0.5 *Max QoS*, Orthrus effectively ensures QoS while achieving superior power savings compared to other baselines. This is attributed to the generality of our approach, allowing

TABLE 6: Device specifications. *l* denotes LITTLE CPU; *b* denotes Big CPU; *s* denotes Super Big CPU.

Device	Google Pixel 3	Google Pixel 6	OPPO Find X3 Pro
SoC	SDM845	Google Tensor	SM8350
LITTLE CPU	4x1.6 GHz Kryo 385 Silver	4x1.80 GHz Cortex-A55	4x1.80 GHz Cortex-A55
Big CPU	4x2.5 GHz Kryo 385 Gold	2x2.25 GHz Cortex-A76	3x2.42 GHz Cortex-A78
Super Big CPU	×	2x2.80 GHz Cortex-X1	1x2.84 GHz Cortex-X1
# Frequency (l/b/s)	18/22/×	11/14/17	16/16/19
OS	Android 11	Android 13	Android 13

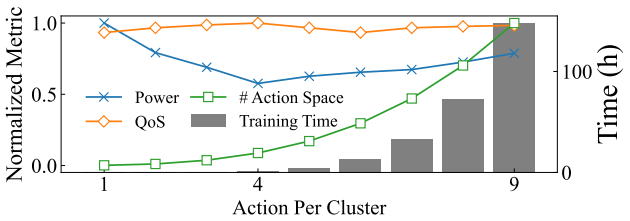
Orthrus to sense and guarantee any QoS metric with a target that can be quantified. Additionally, the reward function does not grant rewards beyond the target QoS, leading to substantial power consumption savings.

When the application's performance demands increase, with a target QoS of 0.9 *Max QoS*, as depicted in Figure 16(b), both *default* and *Smartbalance* exhibit minimal changes in QoS and power consumption. This is because they are unable to sense QoS changes, resulting in the failure to meet the target QoS. Although the remaining two baselines are QoS-aware, Orthrus, capable of jointly employing scheduling and governing technology under QoS awareness, demonstrates superior QoS and greater power savings compared to *AdaMD* and *zTT*. Therefore, we can see that Orthrus has good adaptability to different QoS metrics and target QoS, striving to ensure QoS while saving power consumption as much as possible.

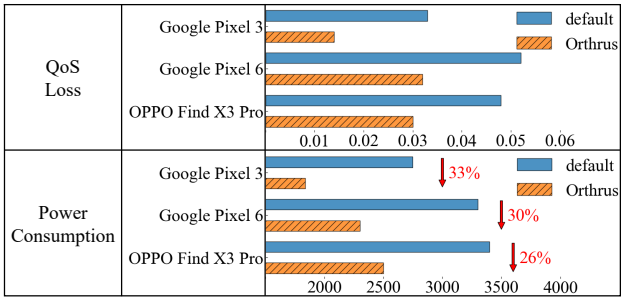
6.7 Scalability Analysis

To evaluate the scalability of Orthrus, we utilize TikTok as the foreground application on mobile devices with varying CPU architectures, as outlined in Table 6. We observe the QoS and power consumption of Orthrus in three scenarios. Changes in CPU architecture resulted in variations in the number of CPU frequency points and CPU clusters. From a scheduling perspective, the method of controlling state transition in the algorithm has evolved from migrating to another cluster to migrating to two other clusters, with no significant increase in complexity. For governing, this means an increase in action space. Taking Google Pixel 6 as an example, if all frequency points are used as actions of the RL agent, the output layer of the neural network would comprise 2618 nodes, significantly impacting training overhead and model effectiveness. Therefore, we explore the effect of the action space size on Orthrus.

As shown in Figure 17(a), we control the number of actions sampled in each cluster on Google Pixel 6 and observe the changes in QoS, power consumption, action space size, and RL agent's training time. The governing action space grows cubically as the number of actions per cluster increases. At this point, power consumption and QoS show certain improvements. However, when the actions per cluster exceed 4, the time for model training to reach convergence increases significantly due to the enlarged action space. Additionally, excessively fine-grained control of CPU frequency leads to less noticeable data differences, so power consumption and QoS do not exhibit the same levels



(a) *QoS, power, size of action space and training time of Orthrus under different actions per cluster sampled on Google Pixel 6.*



(b) *Average power consumption and QoS loss of Orthrus and default on different mobile devices.*

Fig. 17: Scalability analysis.

of improvement. Therefore, on hardware with dozens of frequency points in each cluster, only a certain number of frequency points need to be sampled on average.

While the CPU architecture has an influence on scheduling and governing, it is important to note that this impact is relatively minor, as mentioned earlier. As depicted in Figure 17(b), there is a noticeable decline in Orthrus' performance as the complexity of the CPU architecture increases. Nevertheless, it is noteworthy that Orthrus still achieves significant power savings compared to *default* approach. This underscores the scalability of Orthrus, demonstrating its ability to ensure QoS while reducing power consumption across diverse CPU architectures.

7 A Case Study

In this section, we demonstrate the application of Orthrus in the daily use of smartphones by users, which includes different scenarios such as video streaming, text messaging, gaming, etc. In preparation, we trained several models and stored them in the phone's memory. As the phone starts up, Orthrus runs automatically as a system service. When users use different applications, Orthrus promptly selects and loads an appropriate model to schedule the foreground application while adjusting the hardware frequency. We tested Orthrus on the Oneplus 9 Pro, which includes three different types of cores and can adjust 4 frequencies for each cluster. To demonstrate the performance of Orthrus clearly, we suggest a slight constraint where users should use Skype, TikTok, and PUBG in a specific sequence. This way, we can clearly see how the frequency and position of threads have changed. The user's total time using the phone lasted for one hour.

Fig. 18 shows the performance of Orthrus when users use various applications. The two subfigures on the top depict the frame rate and power consumption of a phone over a period of time, while the bottom two subfigures show

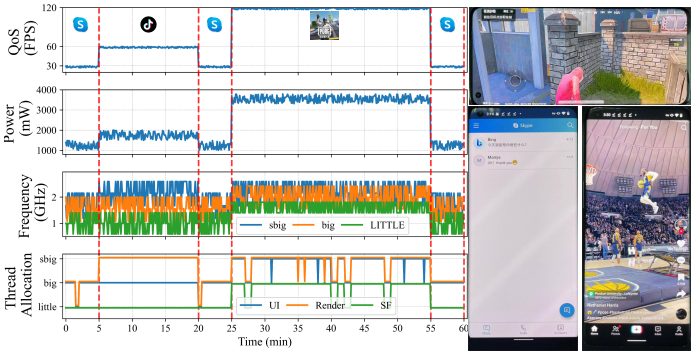


Fig. 18: The performance of Orthrus while operating with various applications in users' daily lives.

dynamic changes in frequency and thread location in the system. We can see that Orthrus can successfully adjust the frequency and schedule threads for different applications. For example, when users switch from Skype to PUBG (third red dash line), the frequency of all three clusters is increased, UI Thread and Render Thread are migrated to the super big (sbig) cluster, and SurfaceFlinger is migrated from the LITTLE cluster to the big cluster.

We collect the same trace for the native system and find Orthrus can guarantee QoS loss within 0.1% for each application, with a 10% reduction in power consumption compared to the native system. We further estimate the overhead of Orthrus and find that when switching between different applications, the model loading time ranges from 10-40 ms, which is negligible compared to the time users spend on their phones daily. Moreover, Orthrus occupies no more than 2% of CPU resources and 1% of memory resources. Overall, we believe Orthrus can be deployed on mobile devices to ensure QoS and increase battery life.

8 Related Work

In the research literature, Orthrus falls into the area of *QoS-aware power management* that jointly leverages *scheduling* and *governing* technologies on mobile devices.

Scheduling on Mobile Devices. Scheduling technology, as the main module in the kernel, is crucial to the system. However, the current scheduling strategy cannot meet the needs of all scenarios, so a lot of work optimizes scheduling for different goals, such as Naithani *et al.* [42] proposed reliability-aware scheduling on heterogeneous multi-core processors, which monitored the reliability characteristics of applications and dynamically scheduled applications to the different clusters to maximize system reliability. Since the computing capacity of different clusters of heterogeneous platforms is different, there are studies such as Kim *et al.* [43], Saez *et al.* [44] and Salami *et al.* [15] focused on using scheduling technology to improve the fairness of the system. There are also some works on how scheduling improves system performance and power consumption. Feng *et al.* [45] exploited the event-driven nature of mobile Web applications, and speculatively executed future events ahead of time in a way that satisfies the QoS while minimizing global energy consumption. Djigal *et al.* [46] presented BUDA that schedules tasks under budget and deadline constraints to minimize execution time. For concurrent applications, Shamsa *et al.* [47] combined an offline power-performance model of each application with an online predictive strategy to make resource allocation decisions for minimizing energy consumption while honoring performance requirements. Yu *et al.* [48] made coordinated core assignment and thread selection decisions based on the performance of each thread on cores and communication patterns.

Governing on Mobile Devices. Governing the heterogeneous cores on mobile devices based solely on CPU utilization is a common practice, but it will cause performance loss and power consumption waste due to ignoring the characteristics of the system and application. Therefore, there are many optimization methods for governing, which can be divided into traditional methods and learning-based methods. Traditional approaches, such as Pathania *et al.* [49], used power and performance models to predict the impact of governing on mobile workloads. Chen *et al.* [50] proposed a heuristic-based algorithm to adjust the CPU frequency based on the video quality adaptively. Yang *et al.* [51] modeled the effects of CPU frequency on TCP throughput and system power. Then a governing algorithm was proposed to save data transmission and CPU energy. Benmoussa *et al.* [52] designed a green metadata-based DVFS scheme for energy-efficient decoding videos. Deshwal *et al.* [53] proposed PaRMIS, a novel information-theoretic framework, to create Pareto-optimal governing policies for given target applications and design objectives. Learning-based methods, such as that proposed by Kim *et al.* [12] utilized a RL-based governor that jointly scaled CPU and GPU frequencies to maximize performance in an energy-efficient manner. Mandal *et al.* [54] proposed an online imitation learning method to construct an offline policy and optimize a given metric (e.g. energy). Choi *et al.* [55] exploited the RL technique to learn the optimal execution speed of the web browser's processes, and adjusted the speed at runtime, thus saving energy and ensuring the QoS. These works lack the study of the interaction between scheduling and governing, which wastes space to reduce power consumption further.

QoS-aware Power Management. Defining QoS precisely is a challenging task in the realm of mobile devices, given its diverse interpretations in research. Unlike real-time scheduling, the tasks addressed in this paper are not time-sensitive, and QoS can be defined in various ways without a decline resulting in serious failures. For mobile devices, QoS can be variously defined as frame rate [56] [57], latency of user-triggered events [24] [58], or IPS [19], among others. These studies employ different methodologies to ensure QoS. For instance, Sahin *et al.* [56] coordinated closed-loop frequency control with thermally-efficient scheduling to deliver the desired QoS. Qian *et al.* [59] considered the inherent connection between the device's state of motion, BS factor, video bitrate and QoS to achieve better QoS and save energy. Rapp *et al.* [19] adopted imitation learning-based scheduling and governing, focusing on an objective QoS measured in IPS. Donyanavard *et al.* [60] optimized reliability under QoS using migration and DVFS implemented at hardware and software layers. Building upon this prior work, our study adopts frame rate as the QoS metric and makes scheduling and governing work jointly to meet the QoS, which has not been explored in existing studies.

9 Conclusion

In this paper, we proposed Orthrus, a scheduling and governing co-optimization approach to optimize the QoS and the power consumption on mobile platforms. We found the decoupled designs among the scheduling method, the governing method and the QoS optimization led to QoS degradation and power wastage. To address the issues, we proposed a finite state machine-based scheduler and a proximal policy optimization-based governer, which migrate QoS-related threads and control the frequency of different clusters while perceiving QoS, and an expert fuzzy control-based coordinator making the scheduler and governor work together to further improve QoS and minimize power consumption. We presented an Orthrus prototype implementation and evaluated Orthrus through different situations. We developed a case study that demonstrates Orthrus can be applied to users' daily use of mobile phones, where its effective scheduling and governing ensure QoS and reduce power consumption.

References

[1] S. Park, Y. Choi, and H. Cha, "WebMythBusters: An in-depth study of mobile web experience," in *Proc. of IEEE INFOCOM*, 2021.

[2] Y. Choi, S. Park, and H. Cha, "Optimizing energy efficiency of browsers in energy-aware scheduling-enabled mobile devices," in *Proc. of ACM MobiCom*, 2019.

[3] C. Qiao, G. Li, Q. Ma, J. Wang, and Y. Liu, "Trace-driven optimization on bitrate adaptation for mobile video streaming," *IEEE Transactions on Mobile Computing*, vol. 21, no. 6, pp. 2243–2256, 2022.

[4] X. Li and G. Li, "An adaptive CPU-GPU governing framework for mobile games on big.LITTLE architectures," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1472–1483, 2021.

[5] S. Dey, A. K. Singh, X. Wang, and K. McDonald-Maier, "User interaction aware reinforcement learning for power and thermal efficiency of CPU-GPU mobile MPSoCs," in *Proc. of IEEE DATE*, 2020.

[6] J. Yu, H. Han, H. Zhu, Y. Chen, J. Yang, Y. Zhu, G. Xue, and M. Li, "Sensing human-screen interaction for energy-efficient frame rate adaptation on smartphones," *IEEE Transactions on Mobile Computing*, vol. 14, no. 8, pp. 1698–1711, 2015.

[7] G. Lee, S. Lee, G. Kim, Y. Choi, R. Ha, and H. Cha, "Improving energy efficiency of android devices by preventing redundant frame generation," *IEEE Transactions on Mobile Computing*, vol. 18, no. 4, pp. 871–884, 2019.

[8] "Big.LITTLE - Arm," https://bit.ly/3WGaZz7, 2023.

[9] P.-C. Hsiu, P.-H. Tseng, W.-M. Chen, C.-C. Pan, and T.-W. Kuo, "User-centric scheduling and governing on mobile devices with big.LITTLE processors," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 1, 2016.

[10] S. Sarma, T. Muck, L. A. D. Bathen, N. Dutt, and A. Nicolau, "SmartBalance: A sensing-driven linux load balancer for energy efficiency of heterogeneous MPSoCs," in *Proc. of ACM/EDAC/IEEE DAC*, 2015.

[11] K. R. Basireddy, A. K. Singh, B. M. Al-Hashimi, and G. V. Merrett, "AdaMD: Adaptive mapping and DVFS for energy-efficient heterogeneous multicores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2206–2217, 2020.

[12] S. Kim, K. Bin, S. Ha, K. Lee, and S. Chong, "zTT: Learning-based DVFS with zero thermal throttling for mobile devices," in *Proc. of ACM MobiSys*, 2021.

[13] Y. Choi, S. Park, S. Jeon, R. Ha, and H. Cha, "Optimizing energy consumption of mobile games," *IEEE Transactions on Mobile Computing*, vol. 21, no. 10, pp. 3744–3756, 2022.

[14] J. Haj-Yahya, M. Alser, J. Kim, A. G. Yağlıkçı, N. Vijaykumar, E. Rotem, and O. Mutlu, "SysScale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors," in *Proc. of ACM/IEEE ISCA*, 2020.

[15] B. Salami, H. Noori, and M. Naghibzadeh, "Fairness-aware energy efficient scheduling on heterogeneous multi-core processors," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 72–82, 2021.

[16] Y. Choi, S. Park, and H. Cha, "Graphics-aware power governing for mobile devices," in *Proc. of ACM MobiSys*, 2019.

[17] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications," in *Proc. of IEEE HPCA*, 2015.

[18] V. Chau, X. Chu, H. Liu, and Y.-W. Leung, "Energy efficient job scheduling with DVFS for CPU-GPU heterogeneous systems," in *Proc. of ACM e-Energy*, 2017.

[19] M. Rapp, N. Krohmer, H. Khdr, and J. Henkel, "NPU-accelerated imitation learning for thermal- and QoS-aware optimization of heterogeneous multi-cores," in *IEEE Proc. of DATE*, 2022.

[20] "Slow Rendering — Android Developers," https://bit.ly/3jkFpci.

[21] "Scheduling for Android display," https://bit.ly/3HFZNyf, 2020.

[22] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra, "Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs," in *Proc. of ACM/EDAC/IEEE DAC*, 2015.

[23] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative CPU-GPU thermal management," in *Proc. of ACM/EDAC/IEEE DAC*, 2016.

[24] O. Sahin, L. Thiele, and A. K. Coskun, "MAESTRO: Autonomous QoS management for mobile applications under thermal constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 8, pp. 1557–1570, 2019.

[25] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim, "Synergistic CPU-GPU frequency capping for energy-efficient mobile games," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, 2017.

[26] U. Gupta, R. Ayoub, M. Kishinevsky, D. Kadjo, N. Soundararajan, U. Tursun, and U. Y. Ogras, "Dynamic power budgeting for mobile systems running graphics workloads," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 1, pp. 30–40, 2018.

[27] "Frame rate — Android Developers," https://bit.ly/3HjA5y9.

[28] "Energy Aware Scheduling (EAS) - Wiki," https://bit.ly/41OKOK1, 2023.

[29] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and H. Ye, "Application-specific performance-aware energy optimization on android mobile devices," in *Proc. of IEEE HPCA*, 2017.

[30] "stress-android," https://bit.ly/3Jnkj83, 2023.

[31] "cgroups," https://bit.ly/3wDDqmE, 2023.

[32] D. Zhang, K. Shen, F. Wang, D. Wang, and J. Liu, "Towards joint loss and bitrate adaptation in realtime video streaming," in *Proc. of IEEE ICME*, 2022.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[34] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proc. of ACM CASES*, 2013.

[35] D. Cheng, Y. Guo, C. Jiang, and X. Zhou, "Self-tuning batching with DVFS for performance improvement and energy efficiency in internet servers," *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 1, 2015.

[36] "sysfs-Wikipedia," https://bit.ly/3WJQeCv, 2023.

[37] "perf_event_open(2) — Linux manual page," https://bit.ly/43Z3Bn9, 2020.

[38] "Monsoon-solutions. HVPM," https://bit.ly/3XNeF3f, 2023.

[39] "TensorFlow Lite," https://bit.ly/3wBadsi, 2023.

[40] "Dhrystone - Wiki," https://bit.ly/3Hn1XBO, 2023.

[41] K. Yan, X. Zhang, J. Tan, and X. Fu, "Redefining QoS and customizing the power management policy to satisfy individual mobile users," in *Proc. of ACM MICRO*, 2016.

[42] A. Naithani, S. Eyerman, and L. Eeckhout, "Reliability-aware scheduling on heterogeneous multicore processors," in *Proc. of IEEE HPCA*, 2017.

[43] C. Kim and J. Huh, "Fairness-oriented os scheduling support for multicore systems," in *Proc. of ACM ICS*, 2016.

[44] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, "Towards completely fair scheduling on asymmetric single-isa multicore processors," *Journal of Parallel and Distributed Computing*, vol. 102, pp. 115–131, 2017.

[45] Y. Feng and Y. Zhu, "PES: Proactive event scheduling for responsive and energy-efficient mobile web computing," in *Proc. of ACM/IEEE ISCA*, 2019.

[46] H. Djigal, L. Liu, J. Luo, and J. Xu, "BUDA: Budget and deadline aware scheduling algorithm for task graphs in heterogeneous systems," in *Proc. of IEEE/ACM IWQoS*, 2022.
[47] E. Shamsa, A. Kanduri, P. Liljeberg, and A. M. Rahmani, "Concurrent application bias scheduling for energy efficiency of heterogeneous multi-core platforms," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 743–755, 2022.
[48] T. Yu, P. Petoumenos, V. Janjic, H. Leather, and J. Thomson, "COLAB: a collaborative multi-factor scheduler for asymmetric multicore processors," in *Proc. of ACM/IEEE CGO*, 2020.
[49] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra, "Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs," in *Proc. of ACM/EDAC/IEEE DAC*, 2015.
[50] X. Chen and G. Cao, "Energy-efficient 360-degree video streaming on multicore-based mobile devices," in *Proc. of IEEE INFOCOM*, 2023.
[51] Y. Yang, W. Hu, X. Chen, and G. Cao, "Energy-aware CPU frequency scaling for mobile video streaming," *IEEE Transactions on Mobile Computing*, vol. 18, no. 11, pp. 2536–2548, 2019.
[52] y. Benmoussa, E. Senn, N. Derouineau, N. Tizon, and J. Boukhobza, "Joint DVFS and parallelism for energy efficient and low latency software video decoding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 858–872, 2018.
[53] A. Deshwal, S. Belakaria, G. Bhat, J. R. Doppa, and P. P. Pande, "Learning pareto-frontier resource management policies for heterogeneous SoCs: An information-theoretic approach," in *Proc. of ACM/IEEE DAC*, 2021.
[54] S. K. Mandal, G. Bhat, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "An energy-aware online learning framework for resource management in heterogeneous platforms," *ACM TODAES*, 2020.
[55] Y. Choi, S. Park, and H. Cha, "Optimizing energy efficiency of browsers in energy-aware scheduling-enabled mobile devices," in *Proc. of ACM MobiCom*, 2019.
[56] O. Sahin and A. K. Coskun, "QScale: Thermally-efficient QoS management on heterogeneous mobile platforms," in *Proc. of IEEE/ACM ICCAD*, 2016.
[57] K.-T. Ho, C.-T. King, B. Das, and Y.-J. Chang, "Characterizing display QoS based on frame dropping for power management of interactive applications on smartphones," in *Proc. of IEEE DATE*, 2018.
[58] X. Li, S. Hong, J. Chen, G. Yan, and K. Wu, "Using psychophysics to guide power adaptation for input methods on mobile architectures," in *Proc. of IEEE HPCA*, 2022.
[59] C. Qian, D. Liu, and H. Jiang, "Harmonizing energy efficiency and QoE for brightness scaling-based mobile video streaming," in *Proc. of IEEE/ACM IWQoS*, 2022.
[60] B. Donyanavard, T. Mück, A. M. Rahmani, N. Dutt, A. Sadighi, F. Maurer, and A. Herkersdorf, "SOSA: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management," in *Proc. of ACM MICRO*, 2019.

Qianlong Sang received his BS degree in Cyber Science and Engineering from Wuhan University in 2022. He is currently pursuing his Ph.D. in Computer Science at Wuhan University. His research interests include edge computing and big data systems.

Jinqi Yan is an undergraduate student at the School of Cyber Science and Engineering from Wuhan University. He is about to pursue a master's degree in Computer Science at Wuhan University. His research interests include edge computing and low-power computing.

Rui Xie is currently the chief architect in system software and architecture field of Guangdong Oppo Mobile Telecommunications Corp. He has been engaged in mobile phone system stability and basic experience optimization for more than 10 years.

Chuang Hu received his BS and MS degrees from Wuhan University in 2013 and 2016. He received his Ph.D. degree from the Hong Kong Polytechnic University in 2019. He is currently an Associate Researcher in the School of Computer Science at Wuhan University. His research interests include edge learning, federated learning/analytics, and distributed computing.

Kun Suo received his BS degree in software engineering from Nanjing University, China, in 2012 and his Ph.D. degree from the University of Texas, Arlington, in 2019. He is currently an Assistant Professor in the Department of Computer Science at Kennesaw State University. His research interests include the areas of cloud computing, virtualization, operating systems, edge computing, Internet-of-things, and software-defined network.

Dazhao Cheng (Senior Member, IEEE) received his BS and MS degrees in Electrical Engineering from the Hefei University of Technology in 2006 and the University of Science and Technology of China in 2009. He received his Ph.D. from the University of Colorado at Colorado Springs in 2016. He was an AP at the University of North Carolina at Charlotte in 2016-2020. He is currently a professor in the School of Computer Science at Wuhan University. His research interests include big data and cloud computing.