An Intelligent Scheduling Approach on Mobile OS for Optimizing UI Smoothness and Power

XINGLEI DOU, Beihang University, Beijing, China
LEI LIU, Beihang University, Beijing, China
LIMIN XIAO, Beihang University, Beijing, China

Mobile devices need to respond quickly to diverse user inputs. The existing approaches often heuristically raise the CPU/GPU frequency according to the empirical rules when facing burst inputs and various changes. Although doing so can be effective sometimes, the existing approaches still need improvements. For instance, raising processors' frequency can lead to high power consumption when the frequency is over-provisioned or fail to meet user demands when the frequency is under-provisioned. To this end, we propose MobiRL, a reinforcement learning-based scheduler for intelligently adjusting the CPU/GPU frequency to satisfy user demands accurately on mobile systems. MobiRL monitors the mobile system status and autonomously learns to optimize UI smoothness and power consumption by conducting CPU/GPU frequency-adjusting actions. The experimental results on the latest delivered smartphones show that MobiRL outperforms the widely used commercial scheduler on real devices—reducing the frame drop rate by 4.1% and reducing power consumption by 42.8%, respectively. Moreover, compared with a study using Q-Learning for CPU frequency scheduling, MobiRL achieves up to a 2.5% lower frame drop rate and reduces power consumption by 32.6%, respectively. Our approach has been deployed in mobile phone products.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Human-centered computing → Mobile computing**; • **Computing methodologies → Reinforcement learning;**

Additional Key Words and Phrases: Reinforcement learning, frequency scheduling/scaling, mobile OS, UI smoothness, power

ACM Reference Format:
Xinglei Dou, Lei Liu, and Limin Xiao. 2025. An Intelligent Scheduling Approach on Mobile OS for Optimizing UI Smoothness and Power. *ACM Trans. Arch. Code Optim.* 22, 1, Article 12 (March 2025), 27 pages. https://doi.org/10.1145/3674910

1 Introduction

Mobile devices (e.g., smartphones) that use the Android **Operating System (OS)** had a global market share close to 70% in 2022 [1]. The Android OS kernel is derived from the Linux kernel

New Paper, Not an Extension of a Conference Paper.
This work was supported by the Key-Area R&D Program of Guangdong (grant no. 2021B0101310002) and the NSFC (grant no. 62072432).
Authors' Contact Information: Xinglei Dou, Lei Liu (Corresponding author), and Limin Xiao, Beihang University, No. 37 Xueyuan Road Zhongguancun, Haidian District, Beijing, China, 100191, China; e-mails: lei.liu@zoho.com, liulei2010@buaa.edu.cn, xiaolm@buaa.edu.cn.

ACM 1544-3973/2025/03-ART12
https://doi.org/10.1145/3674910

with designs for human–machine interaction. Smartphone users are sensitive to the slow responsiveness of the **user interface (UI)** and the high power consumption. Therefore, the smoothness of the UI and power consumption are two critical optimization goals on mobile platforms.

Computing resources on mobile systems—CPU/GPU frequency and enabled CPU cores—directly affect UI smoothness and power consumption. The existing mobile systems, e.g., the ARM big.LITTLE heterogeneous computing architecture, provide both power-saving little cores and high-performance, power-hungry big cores. System designers can have new mechanisms that effectively leverage this specific computing architecture to have better performance on the UI and power. For computing resource scheduling, the **CPU performance scaling (CPUFreq)** [2] and **device frequency scaling (devfreq)** [3] subsystems in the Linux kernel also provide frequency governors for dynamically adjusting CPU and GPU frequency. The **Android Open Source Project (AOSP)** provides the powerHint interface [4] that allows **original equipment manufacturers (OEMs)** to implement customized frequency scheduling/scaling (the two terms are used interchangeably for different cases) policies. OEMs also provide SDK interfaces that allow applications to tell the system to raise the frequency in response to burst inputs or changing loads. These mechanisms are deployed on mobile devices such as Samsung GameDev [5], OPPO Hyper Boost [6], and Huawei PerfGenius [7]. Prior frequency scheduling mechanisms [6, 25] often use heuristic algorithms and empirical experiences. They schedule frequency according to predefined policies. Facing diverse changing loads generated by today's mobile users, existing approaches might not detect frequency over-provision (wasting power) or under-provision (cannot meet users' demands) due to their limited access to OS runtime features. We show a typical instance. The schedutil governor in the CPUFreq subsystem only relies on the CPU utilization estimated by per-entity load tracking [2] for frequency scheduling. It cannot access other OS runtime features, such as cache misses, **instructions per clock (IPC)**, and so forth. Thus, it lacks sufficient information to identify performance issues. As a result, mobile users often encounter application UIs becoming stuck (high frame loss rate) and high power consumption.

We summarize several challenges that need to be conquered in existing frequency scheduling mechanisms: (1) to identify the performance issues and make accurate frequency scaling decisions accordingly to avoid frequency over-provision and under-provision, (2) to handle some cases where the load is high and dynamically changing, and (3) to simultaneously schedule CPU and GPU frequency to avoid sub-optimal performance. To this end, this article proposes MobiRL, a **reinforcement learning (RL)**-based scheduling mechanism for intelligent scheduling/adjusting CPU/GPU frequency on mobile systems. MobiRL formulates the frequency scheduling as a classification problem and employs a customized **Deep Deterministic Policy Gradient (DDPG)** RL model to solve it. Using RL, MobiRL learns to adjust CPU/GPU frequency dynamically to achieve high UI smoothness (low frame loss rate) and low power consumption. When MobiRL performs scheduling, it captures the system status and uses the ML model to predict an action for adjusting CPU/GPU frequency. After conducting the action, MobiRL receives a reward from the mobile system and learns from it. MobiRL can provide precise and timely CPU/GPU frequency scheduling actions compared with prior work [6, 25]. **Machine learning (ML)** has already shown tremendous potential and advantages in many studies on OS and architecture [8–11]. In this article, we leverage ML technologies to improve the scheduling performance of mobile systems. Using ML/AI technologies for mobile systems is a new topic, and there are not many published studies in this field. We are among the pioneers using ML/AI to improve (mobile) systems. We make the following contributions:

(1) We show that the existing heuristic-based CPU/GPU frequency scheduling mechanism used in off-the-shelf mobile devices cannot effectively handle high-load cases where the screen is not being touched. Moreover, this mechanism raises processors' frequency according to

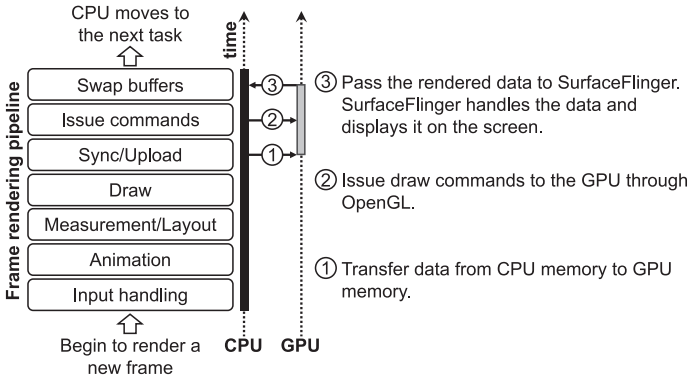


Fig. 1. Frame rendering pipeline.

predefined policies, rather than adjusting frequency on demand. Thus, it cannot promptly satisfy users' diverse loads and often leads to frequency under-provision (UI stuck) or over-provision (energy wasting).

(2) We propose MobiRL, an intelligent frequency scheduling mechanism for mobile systems. MobiRL leverages a reinforcement learning model—DDPG customized for the mobile systems—and learns to dynamically schedule CPU/GPU frequencies according to system loads and user demands, optimizing the UI smoothness and power consumption at the same time.

(3) To use ML on mobile optimizations, we formulate mobile frequency scheduling as a classification problem and further design a discrete frequency scheduling action space. This approach simplifies the decision-making process by reducing the complexity of the action space and speeding up the convergence of the ML model. Our approach reduces the deployment overheads on resource-constrained mobile systems.

(4) We implement MobiRL on the latest real smartphone delivered by a well-known mobile corporation. The experimental results show that MobiRL outperforms the state-of-the-art industrial frequency scheduler, reducing the frame drop rate by 4.1% and reducing power consumption by 42.8%, respectively. Our approach has been adopted by industrial.

2 Background

2.1 UI Rendering

Among all applications running on mobile systems, the currently active applications on the screen are TOP-APPs. UI rendering is the process of displaying frames generated from the TOP-APP on the screen. Mobile systems typically render frames at a stable rate (**Frames per Second (FPS)**). For example, when the FPS is 60, each frame should be rendered within 16.7 ms (i.e., 1,000 ms/60). Frames that take longer than 16.7 ms to render are janky frames. They cause frame drops and hinder UI smoothness. Consecutive frame drops seriously hurt the mobile user experience. As illustrated in Figure 1, the UI rendering pipeline has several steps that require the collaboration of the CPU and GPU [22]. Low CPU or GPU frequency can prolong the frame rendering time, leading to janky frames. When rendering a frame, CPU processing and GPU processing consume approximately 87% and 13% of the time, respectively. Therefore, CPU processing is the performance-critical stage.

2.2 ARM big.LITTLE Heterogeneous Architecture

The ARM big.LITTLE heterogeneous architecture is widely employed in recent smartphone processors [31]. It offers the possibility to optimize the UI smoothness and power consumption

Table 1. Platform Specification

	Specification
OS	Android 12.0 (kernel 5.4.147)
Processor	Qualcomm Snapdragon 888
Processor Config	4×Cortex-A55@1.8 GHz (Little cores, 128 KB L2) 3×Cortex-A78@2.4 GHz (Middle cores, 512 KB L2) 1×Cortex-X1@2.8 GHz (Big core, 1,024 KB L2) 4 MB L3
DRAM	16 GB LPDDR5@3,200 MHz 4×16bit
Storage	512 GB UFS 3.1
GPU	Qualcomm Adreno 660@0.84 GHz

We conduct the experiments on the real h/w delivered by a well-known mobile company.

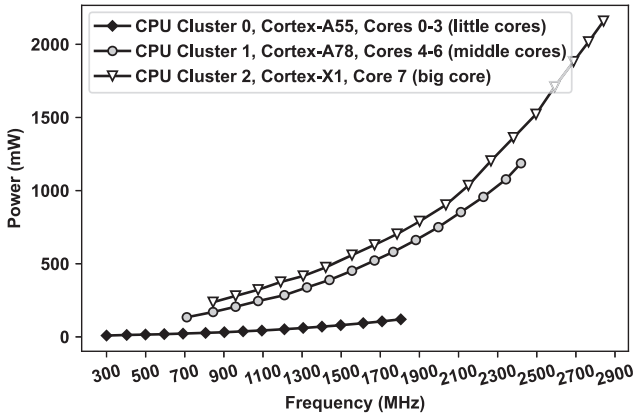


Fig. 2. Single-core power consumption of Qualcomm Snapdragon 888.

simultaneously. For example, Table 1 shows the device specification used in this study. The Snapdragon 888 processor has eight cores that can be grouped into three clusters with different clock speeds and L2 cache capacities [12]. Specifically, CPU cluster 0 includes four little cores (4×Cortex-A55, cores 0 to 3); CPU cluster 1 includes three middle cores (3×Cortex-A78, cores 4 to 6); CPU cluster 2 includes one big core (1×Cortex-X1, core 7). For all cores in each cluster, their frequency and lower and upper frequency limits can be one of several predefined discrete values. For instance, Snapdragon 888's CPU cluster 0 has 16 discrete frequency levels, ranging from 300 MHz to 1,804.8 MHz. Figure 2 shows how single-core power consumption varies with the increase of processor frequency in each CPU cluster. We have two key observations: (1) The cores in CPU cluster 0 are power-saving ones. They can achieve a similar frequency with much lower power consumption than cores in CPU clusters 1 and 2. (2) For some CPU clusters (e.g., cluster 1 or 2 in Figure 2), the increase of power consumption can be sharp with the frequency increases. Thus, in terms of scheduling, the power-saving CPU clusters and frequency levels should be better utilized to reduce power consumption.

2.3 Tunable Knobs on Mobile Systems

Android OS has multiple mechanisms for developers to control performance and power consumption, e.g., sched_boost, core_ctl, CPUFreq, and devfreq. These mechanisms are important to

system developers. Critical applications can temporarily receive a higher CPU scheduling priority by configuring the boost policy in the sched_boost mechanism [23]. The core_ctl mechanism [24] can be used to adjust the maximum and minimum number of cores for each CPU cluster. However, we observe that both sched_boost and core_ctl have risks of causing system performance fluctuations. For example, improper configurations of sched_boost may lead to high context switching overheads; disabling CPU cores using the core_ctl mechanism may result in frame drops and slow responsiveness due to insufficient computing resources. Therefore, they should be used carefully. In our work, we use the CPUFreq [2] and devfreq [3] subsystems in the OS kernel to adjust CPU frequency and GPU frequency. CPUFreq and devfreq subsystems provide editable configuration files that manage each CPU cluster and GPU, including the governor being used, upper frequency limit, lower frequency limit, and so forth. For the CPUFreq subsystem, the governor is used to dynamically adjust the CPU cluster's frequency according to CPU resources required by individual tasks or task groups. For example, when the frequency update is triggered, Android's default governor schedutil adjusts the CPU frequency dynamically to control the estimated CPU utilization close to 80%. If the estimated CPU utilization is above 80%, it dynamically scales up the CPU frequency if possible; otherwise, if utilization is below 80%, it scales down the CPU frequency. The value of the upper frequency limit is the maximum frequency that a CPU cluster can achieve; the value of the lower frequency limit is the minimum achievable frequency. For the devfreq subsystem, the governor and frequency limits work in the same way for GPU. Raising the frequency limits can improve UI smoothness but may incur high power consumption. Conversely, lowering them saves power but may incur frame drops and hinder UI smoothness. By adjusting the frequency limits and observing the feedback from the mobile system, MobiRL learns to optimize UI smoothness and power consumption simultaneously on the fly.

3 Motivation

Smartphone users may tap or swipe the screen at any time, leading to bursty inputs, and thus, system loads fluctuate. To meet user demands promptly, the existing frequency scheduling approach that is widely used in mobile devices raises the lower frequency limits of the CPU to predefined values when the screen is being touched to handle high system loads [6]. It also raises the frequency limits when applications are launched to reduce launch time. Through this approach, the actual running frequency of the processors will not fall below these lower limits. We use this approach as the baseline in this study. However, it does not work well in some cases.

An example is shown in Figure 3. In this example, we use TikTok [35] as the TOP-APP and **Android debug bridge (adb)** [36] to simulate users' swipe actions. In Figure 3, (a) through (d) share the same x-axis. Figures 3(a) through 3(c) show the frequency changes for each CPU cluster. In Figure 3(a), the dot curve line represents the frequency at CPU cluster 0 runs. The black solid curve represents CPU cluster 0's upper frequency limit—the maximum frequency that CPU cluster 0 can achieve. The gray solid curve represents CPU cluster 0's lower frequency limit. CPU cluster 0 has 16 predefined discrete frequency values (y-axis) indicated by the horizontal dashed lines in Figure 3(a), ranging from 300 MHz to 1,804.8 MHz. The lower and upper frequency limits of CPU cluster 0 can be one of these predefined values. When the TOP-APP is launched, the existing approach sets the lower frequency limit of CPU cluster 0 to a predefined value of 691.2 MHz and sets the upper frequency limit to 1,804.8 MHz. When the screen is swiped at frame 12, 44, 69, and 92 (highlighted in Figures 3(a)–(c)), the existing approach increases CPU cluster 0's lower frequency limit to 902.4 MHz (highlighted in the left part in Figure 3(a)). The frequency boost will be over when the swipe action ends. Figures 3(b) and 3(c) show the frequency changes of CPU cluster 1 and 2 during this period, respectively. The lower frequency limits of CPU cluster 1 and 2 are also raised to predefined values when the screen is touched. We record the rendering time for each frame, as

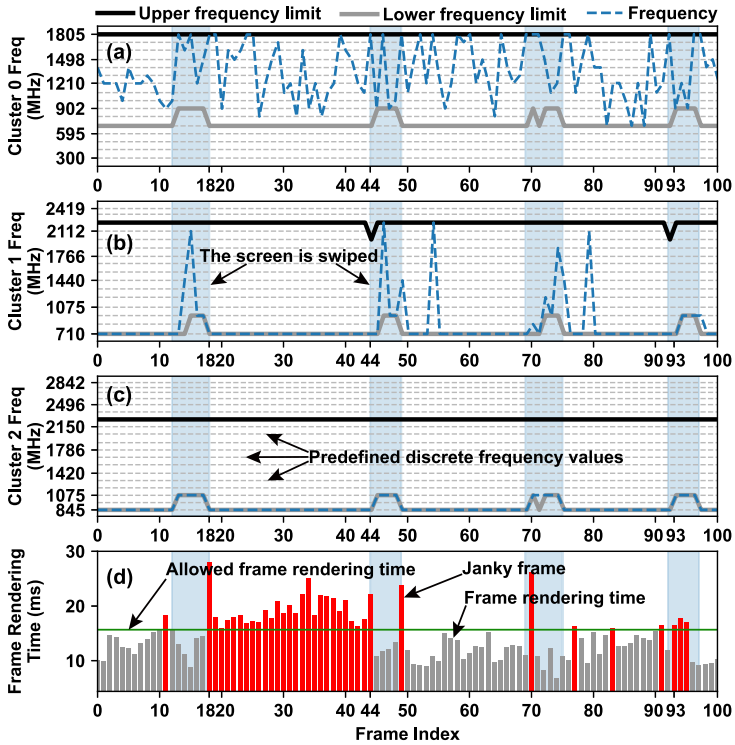


Fig. 3. An example showing the existing approach heuristically boosting the frequency when the user swipes the screen. The hardware is with the latest smartphone delivered by a well-known mobile company (Table 1).

shown in Figure 3(d). Frames with a rendering time higher than the allowed frame rendering time (15.7 ms) are considered janky frames (highlighted in red). Through this example, we can find two shortcomings in the existing frequency scheduling approach:

(1) Cannot handle some high-load cases that are not directly triggered by users' actions. The existing approach is merely designed to tackle the cases where users launch an application or touch the screen. When the system's high loads are not directly caused by launching a new application or touching the screen, the existing approach may fail to handle them—it cannot promptly boost the processors' frequency to satisfy the loads. For example, the current approach fails to address the loads from frame 18 to frame 44 caused by video playing in Figure 3. At frame 18, TikTok switches to the following video. The video triggers data downloads, video decoding, and memory accesses (not brought by the user's direct actions, e.g., touching, etc.), bringing high loads for the mobile system. The existing approach fails to boost the processor's frequency to handle these loads in these cases, leading to consecutive frame drops as shown in Figure 3(d).

(2) Computing frequency under/over-provision for dynamically changing loads. The existing approach often does not adjust the processors' frequency according to system loads' requirements. Instead, it boosts the processors' frequency by raising the lower frequency limit according to predefined rules. For instance, in Figures 3(a) through 3(c), the existing approach raises the lower frequency limits (gray solid curve) of each CPU cluster to 902.4 MHz, 960 MHz, and 1,075.2 MHz (predefined values), respectively, when the screen is being touched/swiped. In frame 93, the system has a high load caused by the swipe

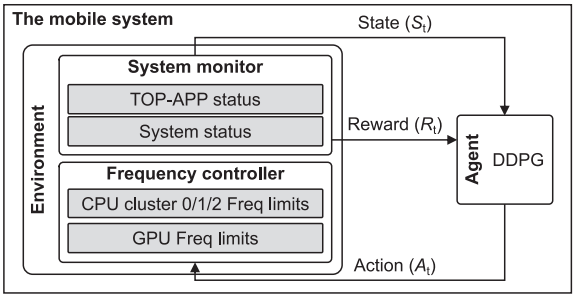


Fig. 4. MobiRL in a nutshell.

action. However, though the lower frequency limits are raised to predefined values in each processor cluster (illustrated in the right of Figures 3(a)–(c)), it is still insufficient to satisfy the load, resulting in three consecutive janky frames (right part in Figure 3(d)). This is the frequency under-provision. Moreover, by contrast, raising the frequency to predefined values may lead to over-provision (a higher frequency than necessary), leading to increased power consumption.

To sum up, on mobile systems, user behaviors may change every second, and scheduling computing frequency to satisfy users' demands can be challenging. Searching in the solution space of processors' frequency for every possible combination of frequency limits (e.g., upper/lower frequency limits) can be difficult. An ideal frequency scheduling mechanism should be able to adjust frequency limits according to system loads accurately. We think that leveraging ML is an ideal approach as it can handle complicated cases with low overheads [18, 29, 34]. So, we want to have a new ML-based frequency scheduling approach, which adjusts processors' frequency on demand according to system loads by a reinforcement learning-based approach, avoiding janky frames for UI smoothness and saving power.

4 MobiRL: System Design

4.1 The Overview of Our Design

In this work, we propose MobiRL, an intelligent RL-based CPU/GPU frequency scheduling mechanism for mobile systems. MobiRL leverages an RL model to learn how to optimize power and UI smoothness simultaneously by adjusting CPU/GPU frequency. Several challenges need to be addressed for MobiRL to achieve its goal: (1) The state space and action space are complicated, making MobiRL challenging to converge. (2) MobiRL has two optimization goals, i.e., UI smoothness and power consumption, which are difficult to achieve at the same time on mobile systems. Optimizing UI smoothness (needs more power) may negatively affect another goal, and vice versa. (3) Since the battery power of mobile devices is limited, the ML models used in this work must be low overhead. To address the above challenges, we design a new ML-based approach, a DDPG model [20] customized for frequency scheduling on mobile systems. Our design has a reduced action space and a reward function that optimizes power consumption while avoiding frame drops. Our design can work on mobile systems with a low overhead.

Figure 4 shows MobiRL in a nutshell. MobiRL aims to schedule CPU/GPU frequency to optimize power and UI smoothness simultaneously. A typical RL framework employs an agent to learn how to conduct an action based on the state acquired from the environment and subsequently improves the agent's decision-making by learning from the reward. MobiRL's RL model takes CPU/GPU frequency scheduling actions according to the mobile system's current status. After the

Table 2. Controlled Configuration Parameters

Controlled configuration parameters	Predefined discrete frequency values (MHz)	Pruned values (MHz)
(1) CPU cluster 0 upper frequency limit	300, 403.2, 499.2, 595.2, 691.2, 806.4, 902.4, 998.4, 1,094.4, 1,209.6, 1,305.6, 1,401.6, 1,497.6, 1,612.8, 1,708.8, 1,804.8	300, 403.2, 499.2
(2) CPU cluster 0 lower frequency limit		1,612.8, 1,708.8, 1,804.8
(3) CPU cluster 1 upper frequency limit	710.4, 844.8, 960, 1,075.2, 1,209.6, 1,324.8, 1,440, 1,555.2, 1,670.4, 1,766.4, 1,881.6, 1,996.8, 2,112, 2,227.2, 2,342.4, 2,419.2	710.4, 844.8, 960
(4) CPU cluster 1 lower frequency limit		2,227.2, 2,342.4, 2,419.2
(5) CPU cluster 2 upper frequency limit	844.8, 960, 1,075.2, 1,190.4, 1,305.6, 1,420.8, 1,555.2, 1,670.4, 1,785.6, 1,900.8, 2,035.2, 2,150.4, 2,265.6, 2,380.8, 2,496, 2,592, 2,688, 2,764.8, 2,841.6	844.8, 960, 1,075.2
(6) CPU cluster 2 lower frequency limit		2,688, 2,764.8, 2,841.6
(7) GPU upper frequency limit	315, 379, 443, 491, 540, 608, 676, 738, 778, 840	–
(8) GPU lower frequency limit		–

frequency scheduling action, the reward evaluates the mobile system's power consumption and UI smoothness. MobiRL works as a component in the OS.

4.2 Design Details of MobiRL

4.2.1 Environment. The RL environment, in this context, refers to the mobile system. In our design, it includes a system monitor and frequency controller. The system monitor monitors and collects the mobile system's status. It can collect essential information from the OS during the last frame rendering each time some frames are rendered. The information includes (1) RL model input features (refer to Section 4.2.3); (2) features used for the MobiRL control flow, e.g., CPU/GPU frequency limits; and (3) features used for performance evaluation, e.g., frame rendering time, allowed frame rendering time, device temperature, voltage, and current. In MobiRL, the default setting is to collect this information each time 10 frames are rendered to reduce the feature collection overhead.

The frequency controller conducts and validates CPU/GPU frequency scaling actions. It provides interfaces that allow the scheduler to perform frequency scheduling actions. It manages frequency limits by updating editable configuration files in the CPUFreq and devfreq subsystems. MobiRL controls parameters, including CPU cluster 0/1/2 and the GPU's upper frequency and lower frequency limits, as listed in Table 2. The frequency limits can be one of the predefined values in Table 2. During online training, MobiRL may have frequency limit configurations that lead to excessive power consumption or continuous frame drops when it explores in the scheduling space. To ensure reliability, MobiRL identifies these configurations through offline sampling and prunes them (i.e., pruned values in Table 2), preventing scheduling actions that may cause serious performance issues. Additionally, MobiRL can be easily extended to manage other resource dimensions, including the number of cores enabled in each CPU cluster, DDR frequency, and so forth.

4.2.2 Agent. The agent is the DDPG model [20] customized for mobile systems. DDPG is a reinforcement learning model that uses an Actor-Critic framework. We employ DDPG for two reasons: (1) DDPG is suitable for continuous state space on mobile systems, and (2) DDPG has an Actor-Critic architecture. The Critic provides a stable baseline for the Actor, leading to stable and efficient learning. MobiRL's DDPG model learns to conduct CPU/GPU frequency scheduling actions that can maximize long-term rewards. DDPG has four deep neural networks, i.e., Actor, Critic, Actor target, and Critic target. The Actor network is used to infer a frequency scheduling action given a state S_t. The Critic network infers the Q-value given a state S_t and an action A_t. The Q-value is the expected cumulative reward of conducting the action A_t given the state S_t. The Actor and Critic target networks have identical structures as the Actor and Critic networks, respectively. Compared to Actor and Critic networks, Actor target and Critic target networks have lower learning rates, which can prevent drastic changes in the action selections during training, enhancing the stability

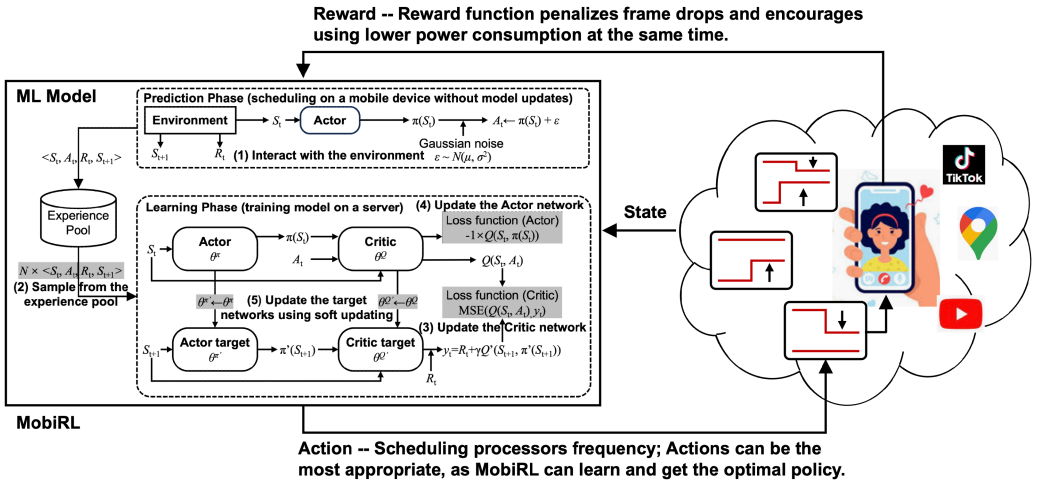


Fig. 5. MobiRL's working process.

of the model training process. In DDPG, each network (structures for Actor/Critic network) has an input layer, output layer, and two hidden layers. Each hidden layer has 40 neurons. We use this setup because adding more layers does not further improve the performance but incurs higher training overhead. We customize the DDPG model for deployment on a mobile system. In the Actor network and Actor target network, we used softmax as the activation function in the output layer, thereby transforming the frequency scheduling into a classification problem. Doing so simplifies the action space, speeding up the ML model convergence. All other layers use ReLU as the activation function. Actor and Critic networks use Adam optimizer [32] for gradient updates.

As shown in Figure 5, the control flow of MobiRL's DDPG model has two phases, i.e., learning phase and prediction phase. During the learning phase, we build a DDPG model using TensorFlow on a Linux server. By interacting with the mobile system through adb, we obtain experiences including runtime traces collected by the system monitor, frequency scheduling actions predicted by the model, and corresponding reward values. The neural network parameters are updated based on these experiences. During the prediction phase, we deploy a simplified version of MobiRL on the mobile system for low-overhead frequency scheduling. Experiences obtained during the learning phase can be stored and exported for further model updates. To train MobiRL's DDPG model, the following steps are required:

(1) Interact with the environment. When DDPG performs scheduling, the state S_t is fed into the Actor network. The Actor network outputs $\pi(S_t) = A_t$, where π represents the policy function of the Actor network. Gaussian noise is added to A_t based on a mean μ and a standard deviation σ to encourage the agent to explore the environment more effectively. As the Actor network uses softmax as the activation function in the output layer, A_t represents the probability distribution of available actions. Algorithm 1 is invoked to obtain the actual frequency scheduling action. In Algorithm 1, the action with the maximum probability in the softmax output is selected. After the action is conducted, the reward R_t is calculated based on the UI smoothness and the power consumption according to the reward function (Equation (3)). The state transitions to S_{t+1} after the action is conducted. The tuple $<S_t, A_t, R_t, S_{t+1}>$ is stored in the experience pool for experience replay.

(2) Sample from the experience pool. Randomly select a batch of tuples from the experience pool for model learning and gradient updates. The default batch_size is 64, as shown in Table 4.

ALGORITHM 1: Obtain the actual frequency-adjusting action using the Actor network's output

```
Input: The Actor network's output A
max_value ← max (A);
max_index ← A.indexOf (max_value);
Get the target parameter and scaling direction of the action corresponding to max_index from the action space;
if the action is an idle action that does not adjust any configuration parameter then
    Return;
end
if the scaling direction is to scale up then
    l_up ← 3; // maximum step size for scaling up the target parameter
    while scaling up the target parameter for l_up levels is an illegal action do
        l_up ← l_up − 1; // reduce the max step
    end
    Scale up the target parameter for l_up levels;
end
if the scaling direction is to scale down then
    l_down ← 3; // maximum step size for scaling down the target parameter
    while scaling down the target parameter for l_down levels is an illegal action do
        l_down ← l_down − 1; // reduce the max step
    end
    Scale down the target parameter for l_down levels;
end
```

(3) Update the Critic network. The **Mean Squared Error (MSE)** between the predicted Q-value and the target Q-value is used as the loss function for the Critic network. The gradient of this loss function with respect to the parameters of the Critic network (i.e., the set of weights and biases) is computed and used to update the network's parameters. Specifically, the predicted Q-value is calculated as $Q(S_t, A_t)$, where Q represents the action-value function of the Critic network. The target Q-value is calculated as $y_t = R_t + \gamma Q'(S_{t+1}, \pi'(S_{t+1}))$. Here, γ ($\gamma \in [0, 1]$) is a discount factor that defines the weight of immediate rewards and future rewards. A higher value of γ gives more weight to future rewards, and the agent tends to prioritize long-term benefits, while a lower value of γ places more emphasis on immediate rewards. Q' represents the action-value function of the Critic target network. π' represents the policy function of the Actor target network.

(4) Update the Actor network. The goal of the Actor network is to maximize $Q(S_t, \pi(S_t))$, i.e., to maximize the expected cumulative reward of the state–action pair (S_t, $\pi(S_t)$). Therefore, the negative Q-value is used as the loss function for the Actor network. The gradient of this loss function concerning the parameters of the Actor network is computed and used to update the network's parameters.

(5) Update the target networks using soft updating. The soft updating blends the parameters of the Actor and Critic networks into the Actor target and Critic target networks, achieving smooth updates for the target networks and avoiding fluctuations in target values. The update processes are as follows:

$$\begin{aligned} \theta^{Q'} &= \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\pi'} &= \tau\theta^{\pi} + (1-\tau)\theta^{\pi'}. \end{aligned} \tag{1}$$

Table 3. Input Features of the DDPG Model

OS-related features
(1) CPU_load_cluster_0, (2) CPU_load_cluster_1, (3) CPU_load_cluster_2, (4) CPU_freq_cluster_0, (5) CPU_freq_cluster_1, (6) CPU_freq_cluster_2, (7) DDR_freq, (8) GPU_freq, (9) is_swiping, (10) device_temperature, (11) Freq_upper_limit_cluster_0, (12) Freq_lower_limit_cluster_0, (13) Freq_upper_limit_cluster_1, (14) Freq_lower_limit_cluster_1, (15) Freq_upper_limit_cluster_2, (16) Freq_lower_limit_cluster_2, (17) Freq_upper_limit_GPU, (18) Freq_lower_limit_GPU, (19) instructions_per_cycle, (20) cache_miss_rate
TOP-APP-related features
(1) cache_miss_rate, (2) main_thread_cluster_id, (3) render_thread_cluster_id, (4) task_load

Here, θ represents the parameters in the neural network. θ^{Q}, $\theta^{Q'}$, θ^{π}, and $\theta^{\pi'}$ correspond to the parameters in the Critic, Critic target, Actor, and Actor target networks, respectively. The hyperparameter τ ($\tau \in [0, 1]$) controls the update rate of the target networks.

4.2.3 State. The state represents the input features of the MobiRL ML model. As shown in Figure 5, MobiRL captures the system states of the mobile system, including system load, device temperature, cache miss rate, and so forth. The states are inputs of the ML model. They accurately reflect the current status of mobile systems. The Actor network predicts a frequency scheduling action based on the state. In MobiRL, the state includes 20 operating system-related features and 4 TOP-APP-related features, as shown in Table 3. Operating system-related features include (1) OS performance counters (i.e., CPU load, IPC, and cache miss rate), (2) CPU/GPU/DDR frequency and frequency limits, and (3) device statuses (i.e., device temperature, whether the screen is being touched or not). TOP-APP-related features include the cache miss rate of the process, the CPU load generated by the process, and the ID of the CPU cluster where the main thread and the render thread are running. The features are normalized into [0,1] according to Equation (2):

$$Feature_{\text{normalized}} = \frac{Max - Feature}{Max - Min}, \tag{2}$$

where $Feature$ is the raw value, and Max and Min are predefined maximum and minimum values of each feature, respectively.

4.2.4 Action. The action represents the frequency scheduling decisions made by the DDPG model in response to the mobile system status. The Actor network outputs a k-dimensional vector $A = \{a_1, a_2, \ldots, a_k\}$, $\sum_{i=1}^{k} a_i = 1$. As the Actor network uses softmax as the activation function, A represents the probability distribution of available actions over the action space. For the eight configuration parameters in Table 2, we design an action space that contains 17 frequency scheduling actions. For each configuration parameter, there are two actions: one for scaling up and another for scaling down the parameter. Additionally, there is an idle action that does not adjust any configuration parameter. Each time the Actor network outputs A, the action with the highest probability is selected as the chosen action.

A frequency configuration parameter can take a number of predefined discrete values. For example, the upper/lower frequency limits of CPU cluster 0 can take 16 predefined discrete levels

ranging from 300 MHz to 1,804.8 MHz. We set the maximum step size to be three levels for a specific frequency scheduling action. Additionally, there is a constraint that the upper frequency limit must be greater than or equal to the lower frequency limit. Actions that violate the constraint are invalid. To avoid invalid actions, we compute the upper and lower bounds for scaling up or down each resource configuration parameter based on the current frequency limit settings and the parameter's counterpart. The counterpart represents another frequency limit of the same CPU cluster or GPU. For example, the counterpart of CPU cluster 0's upper frequency limit is CPU cluster 0's lower frequency limit. Algorithm 1 shows more details.

As illustrated in Figure 5, MobiRL's ML model can intelligently output frequency scheduling actions that optimize the UI smoothness and power consumption. The actions can be the most appropriate, as the customized DDPG in MobiRL can learn and get the optimal policy.

4.2.5 Reward. The reward function defines the objective of MobiRL. It is designed to minimize power consumption while ensuring UI smoothness. MobiRL schedules each time it receives the system status information the system monitor collects. The reward function is invoked before the scheduling to calculate the reward for the last state–action pair. The reward includes both the current reward and the historical reward. The current and historical rewards are calculated using the same reward function (Equation (3)), but they differ in the data range used for computation. The current reward is calculated using all data from the conduction of the last action up to the present. On the other hand, the historical reward is calculated using all data from the past 3 seconds. The final reward value is the sum of the current and historical rewards, each multiplied by a weight of 0.5. MobiRL has two optimization goals, i.e., UI smoothness and power consumption. To optimize them simultaneously, the reward function is designed as a piecewise function. When there are frame drops, i.e., the frame rendering time t exceeds the allowed frame rendering time t_{allowed}, MobiRL only optimizes the UI smoothness, i.e., $r = r_{\text{UI}}$. MobiRL learns to optimize the power consumption when there are no frame drops, i.e., $r = r_{\text{power}}$. The reward function is shown as follows:

$$\begin{aligned} r &= \begin{cases} r_{\text{UI}} & \text{if } t \geq t_{\text{allowed}} \\ r_{\text{power}} & \text{if } t < t_{\text{allowed}} \end{cases} \\ r_{\text{UI}} &= -\alpha \log(t/t_{\text{allowed}}) \\ r_{\text{power}} &= \beta(1 - p/\text{TDP}), \end{aligned} \tag{3}$$

where t denotes the maximum frame rendering time of all data used to calculate the reward. t_{allowed} is the allowed frame rendering time, which is typically set as 1 s/FPS −1 ms in the industrial track. This is because frame drops may occur when the frame rendering time is close to 1 s/FPS. α and β are the weight parameters of r_{UI} and r_{power}, respectively.

The UI smoothness reward r_{UI} penalizes frame drops. When the frame rendering time exceeds the allowed frame rendering time, i.e., $t > t_{\text{allowed}}$, the ratio of t and t_{allowed} is greater than 1, and r_{UI} yields a negative reward value. The power reward r_{power} encourages lower power consumption. It is designed as 1 minus the ratio of measured power consumption and the **thermal design power (TDP)** of the processor. TDP refers to the power consumption of a processor under the maximum theoretical load. The TDP of Snapdragon 888 is 5 W. The lower the measured power consumption, the higher the r_{power} value. As in Figure 5, the rewards are the feedback after actions are conducted. MobiRL can have ideal solutions as the reward function penalizes frame drops and encourages lower power consumption. Moreover, MobiRL can meet user demands with minimum power consumption by using the reward function, avoiding overheating and voltage spikes. Therefore, MobiRL can reduce device heat and increase the processor core's lifetime.

4.3 The Central Control Logic

The central control logic manages the data and control flow of MobiRL. It has a learning phase and a prediction phase.

4.3.1 Learning Phase. Each time a configurable number (default as 10) of frames is rendered, MobiRL obtains the data collected during the last frame (Section 4.2.1). The collected data is added to the list of runtime traces. Upon receiving the data, MobiRL conducts scheduling and extracts the state S_t from the collected data and feeds it into the Actor network to obtain the frequency scheduling action A_t. After conducting the action, MobiRL obtains S_{t+1}. The reward R_t is calculated before the next scheduling process. The tuple $<S_t, A_t, R_t, S_{t+1}>$ is stored in the experience pool for experience replay. After a round of scheduling, MobiRL extracts a batch of tuples and updates the networks (Section 4.2.2).

4.3.2 Prediction Phase. During the prediction phase, we deploy a simplified version of MobiRL on mobile systems for low-overhead frequency scheduling. We deploy only the Actor network on the mobile system leveraging TensorFlow Lite. When 10 frames are rendered, MobiRL collects the mobile system state using the system monitor. MobiRL feeds the state into the Actor network to predict a frequency scheduling action and conducts it. Therefore, the sampling rate and the frequency of MobiRL's decision-making are six times per second (6/second). In practice, MobiRL's action is prompt enough to handle transient load and user demands. It can quickly schedule CPU/GPU frequency limits to achieve the frequency limit configuration that can maximize the long-term rewards within seconds.

4.4 Implementation

We implement the MobiRL training framework and train the DDPG model on a server running Linux 5.15.0. We also implement MobiRL on the latest real smartphone (Table 1) by integrating it into Android's system_server process so that MobiRL has permission to collect system status and conduct frequency scheduling actions. The well-trained model is deployed on the mobile system using TensorFlow Lite v2.5.0. TensorFlow Lite also supports on-device training [21]. But we decided to train the model on the server to avoid the high overhead during the model training. Both CPU and GPU can be used for on-device inference of the model. Moreover, we use the default schedutil and msm-adreno-tz governors for CPUFreq [2] and devfreq [3] subsystems, respectively. The model training hyperparameters are determined using Hyperopt [33], which can infer the parameter configuration that yields the highest performance by sampling in the exploration space. We use the average reward value after convergence as the objective to minimize and then define a value range for each parameter to be tuned, for example, Actor learning rate $\in$ [0.001,0.0001], Critic learning rate $\in$ [0.01,0.001]. Then we employ the Random Search algorithm in Hyperopt to sample several configurations in the exploration space. The parameter configuration that yields the highest reward after convergence is used. The training parameters are in Table 4.

5 Evaluations

We compare MobiRL with the most related work in [2, 6, 44]: (1) *Hyper Boost* [6]. Hyper Boost [6] is the state-of-the-art industrial frequency scheduler on the newly released mobile device (Table 1). Hyper Boost raises the frequency when users launch an application or swipe the screen. Besides that, Hyper Boost can boost the frequency for dozens of applications by recognizing critical scenarios and notifying the mobile system through Hyper Boost SDK interfaces [6]. Details are in Section 5.4. (2) *Q-Learning approach* [44]. The work in [44] employs a reinforcement learning algorithm for CPU frequency scheduling. We denote this approach as Q-Learning. It selects the appropriate CPU frequency using Q-Learning based on estimated CPU loads to minimize

Table 4. RL Training Parameters

Parameter	Value
μ (mean of the added Gaussian noise)	0
σ (standard deviation of the added Gaussian noise)	0.2
batch_size	64
memory_pool_size	20,000
γ (a discount factor used when calculating the target Q-value, see Section 4.2.2)	0.99
τ (a parameter that controls the update rate of the target networks, see Section 4.2.2)	0.001
α (weight parameter of r_{UI})	5
β (weight parameter of r_{power})	1
steps_per_episode	200
Actor_learning_rate	0.0005
Critic_learning_rate	0.005

energy consumption. Details are in Section 5.5. (3) *Default frequency scaling governors in the CPUFreq subsystem* [2]. The CPUFreq [2] subsystem provides several default frequency scaling governors that adjust the CPU frequency within the frequency limits. They are frequency scaling algorithms that are widely used in mobile systems. We compare MobiRL with four representative governors including performance, powersave, schedutil, and conservative to illustrate MobiRL's effectiveness over existing frequency scheduling mechanisms in mobile systems. The performance governor adjusts the frequency to the upper frequency limits for higher performance. The powersave governor adjusts the frequency to the lower frequency limits to save power. Both the schedutil and conservative governors make frequency scaling decisions based on the estimated CPU utilization. The schedutil governor is more flexible due to its shorter scheduling time interval and larger-frequency scheduling steps. The experimental results are in Section 5.6.

5.1 Methodology

The following metrics are used in evaluations.

Frame Drop Rate. The frame drop rate refers to the percentage of frames that should be rendered but are not rendered during real-time rendering due to frequency under-provision. It is measured using the gfxinfo tool in Android.

Power Consumption. Power consumption is calculated by multiplying the voltage by the current. In MobiRL, adb is used for communication between the server and the mobile device. To eliminate the impact of charging current on power measurement, we establish the adb connection using WiFi instead of the USB cable.

For all experiments, we set the FPS to 60; i.e., the allowed frame rendering time is (1 s/60) −1 ms = 15.7 ms. The system monitor collects the mobile system's status each time 10 frames are rendered; i.e., the sampling rate is six times per second. Each time MobiRL receives the collected data, it makes a frequency scheduling decision. Therefore, the frequency of MobiRL decision-making is also six times per second. In the experiment, we ensure that the initial device temperature is below 35 °C for each trial to avoid forced throttling of the mobile device due to high temperatures.

5.2 Benchmarks

We evaluate MobiRL using some representative applications, including video playing, social media, photo taking, and so forth. They exhibit different computing/memory patterns. They are TikTok [35], Weibo [37], Taobao [38], Camera [52], and Browser [51]. In addition, we also

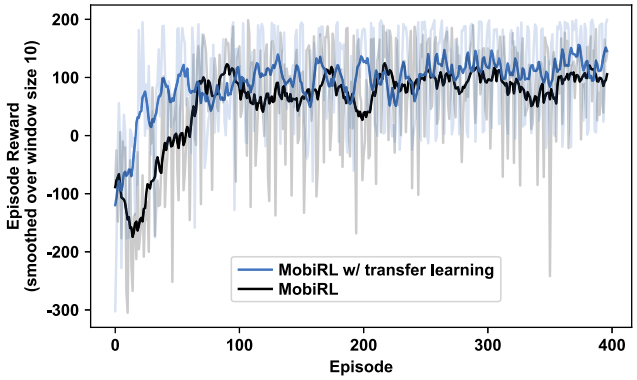


Fig. 6. Learning curve of MobiRL's DDPG model. Using transfer learning can reduce MobiRL's retraining cost. MobiRL's DDPG model trained using transfer learning (the upper curve in black) converges much faster and achieves 23.6% higher episode reward upon convergence compared with the model trained without transfer learning (the lower curve in black).

evaluate cases where the system load varies by running various background applications before launching the TOP-APP.

5.3 MobiRL Training and Convergence

We show that MobiRL's DDPG model can converge faster using transfer learning in Figure 6. We first train the model at FPS 60 (i.e., each frame should be rendered in 1 s/60 = 16.7 ms) using the benchmark applications in Section 5.2. The black curve (the lower curve) in Figure 6 shows the model's episode reward during the online training. Each episode has 200 training steps (Table 4). The episode reward is the sum of the reward value in each training step of an episode. A higher episode reward indicates that the model can achieve lower frame drop rates and power consumption during this episode. Then, we change the FPS to 120 (i.e., each frame should be rendered in 1 s/120 = 8.3 ms). In this case, the previous model trained at 60 FPS does not work well and needs to be retrained because the user demand changes. We retrain the model at FPS 120 using transfer learning based on the pre-trained model at FPS 60. The blue curve (the upper curve) shows the model's episode reward.

5.3.1 Model Training at 60 Frames per Second. The black curve (the lower curve) in Figure 6 shows that MobiRL's DDPG model learns to maximize the episode reward over time. The model quickly converges in 100 episodes; i.e., the episode reward stabilizes and consistently remains at a high level. At the beginning of the learning process (from episode 0 to 15), MobiRL's DDPG model learns to schedule frequency by exploring the search space, leading to fluctuations in UI smoothness and power consumption. After that, we observe a rapid increase in episode reward from episode 15 to 100. During this period, DDPG learns to improve the model's decision-making, achieving a higher reward by optimizing UI smoothness and power consumption. Additionally, we observe that UI smoothness and power consumption optimize during the training process. For each 10 training episodes, we save the checkpoint of model parameters. To evaluate the model trained for a specific number of episodes, we load the saved parameters and compare it with Hyper Boost in terms of frame drop rate and power consumption when handling the same workload. The starting policy is no better than Hyper Boost. But after episode 100, MobiRL outperforms Hyper Boost. We evaluate MobiRL with parameters at episode 390. MobiRL outperforms Hyper Boost by 5.8% and 32.3% in terms of frame drop rate and power consumption, respectively.

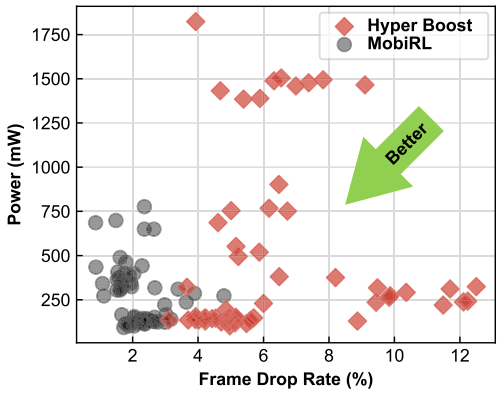


Fig. 7. Performance distributions for 58 workloads scheduled w/ Hyper Boost and MobiRL, respectively.

5.3.2 Transfer Learning with a Low Retraining Cost at 120 Frames per Second. The blue curve (the upper curve) in Figure 6 shows that transfer learning can reduce the retraining cost and make the model converge faster. We first copy the network parameters of the pre-trained model to the new model at FPS 120 and then conduct online training. The episode reward of the model trained using transfer learning increases quickly in the first 50 episodes and consistently remains high; i.e., the model converges in around 50 episodes. Moreover, the model trained using transfer learning achieves 23.6% higher episode reward upon convergence compared with the case when transfer learning is not used; i.e., the model trained using transfer learning can achieve a lower frame drop rate and power consumption in practice. Training the model for 50 episodes takes around 30 minutes. This indicates that MobiRL's DDPG model has a low retraining cost by using transfer learning, making MobiRL easier to generalize across new cases or platforms.

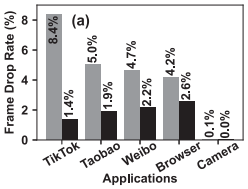
5.4 Effectiveness of MobiRL Compared with the Industrial Scheduler

We first evaluate MobiRL against the state-of-the-art industrial frequency scheduler Hyper Boost [6] that is widely deployed on modern mobile devices. We show the effectiveness of MobiRL as follows.

5.4.1 Performance Distribution. MobiRL exhibits better UI smoothness and lower power consumption (Figure 7). Using ML, MobiRL can schedule the frequency accurately to quickly satisfy user demands. We construct 58 workloads by varying the TOP-APP, the time interval for swipe actions, and the number of background applications. For instance, in workload 1, the TOP-APP is TikTok, the interval for swipe actions is 5 seconds, and three background applications are launched. For each workload, we use MobiRL and Hyper Boost, respectively. We collect the frame drop rate and power during the 120-second period after the scheduling begins. Figure 7 shows the distributions of the scheduling results of these 58 workloads for MobiRL and Hyper Boost, respectively. The x-axis shows the frame drop rate; the y-axis denotes the power. Generally, MobiRL can achieve a lower frame drop rate with lower power consumption for these workloads. On average, MobiRL has a frame drop rate of 2.2% and a power consumption of 275.8 mW, while Hyper Boost has a frame drop rate of 6.3% and a power consumption of 482.1 mW. MobiRL reduces the frame drop rate and power by 4.1% and 42.8% for these workloads, respectively. We also record the frame rendering time of the cases in Figure 7. Compared with Hyper Boost, MobiRL reduces the average frame rendering time and frame rendering time of the 50th, 90th, 95th, and 99th percentiles by 21.2%, 21.0%, 23.1%, 20.8%, and 16.8%, respectively. Table 5 has more details. MobiRL performs

Table 5. Frame Rendering Time Comparison

Scheduler	Framed rendering time (ms)				
	50th	90th	95th	99th	Average
MobiRL	9.0	14.7	17.5	25.9	10.0
Hyper Boost	11.5	19.1	22.1	31.2	12.7



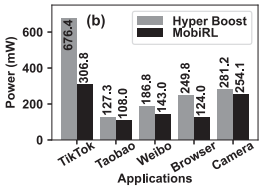
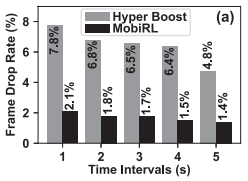


Fig. 8. MobiRL vs. Hyper Boost w/ varying applications.



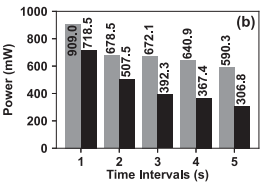


Fig. 9. MobiRL vs. Hyper Boost w/ varying TOP-APP loads.

better mainly because it has the learning ability and can dynamically adjust the frequency to satisfy user demands. By contrast, Hyper Boost cannot promptly handle some high-load cases that are not directly triggered by the user's action, which often leads to frequency under/over-provision when handling dynamically changing loads, leading to worse UI smoothness and increased power consumption.

5.4.2 Performance for Varying Applications. We study how MobiRL performs for each application in Figure 8. Figures 8(a) and 8(b) show each application's frame drop rate and power consumption, respectively. Each application's results are based on the average of five experimental results. MobiRL can optimize the frame drop rate and power consumption for all applications. Specifically, for TikTok, Taobao, Weibo, Browser, and Camera, MobiRL can reduce the frame drop rate by 7.0%, 3.1%, 2.5%, 1.6%, and 0.1%, respectively. MobiRL can also save power consumption by 54.6%, 15.2%, 23.5%, 50.4%, and 9.6%, respectively. We observe the most significant optimization in frame drop rate and power consumption for TikTok, a video-playing application. For Camera, it achieves a frame drop rate of 0.1% when using Hyper Boost for scheduling. Using MobiRL for scheduling can further reduce its power consumption by 9.6%.

5.4.3 Performance for Varying TOP-APP Loads. We test how MobiRL performs for varying loads of TOP-APP in Figure 9. We control the load of the TOP-APP by changing the time intervals between swipe actions. The shorter the time interval, the higher the load. We use TikTok as the TOP-APP in Figure 9. We observe that MobiRL consistently outperforms Hyper Boost in every time interval setting. For time intervals ranging from 1 s to 5 s, MobiRL outperforms Hyper Boost by 5.7%, 5.0%, 4.8%, 5.6%, and 3.4% in terms of frame drop rate, and saves power by 30.0%, 25.2%, 41.6%, 47.9%, and 48.0%, respectively. Note that our ML model is only trained on TikTok with a 5-second swipe action interval. However, MobiRL can perform well under other interval settings, indicating that MobiRL performs well in generalization across varying TOP-APP loads. Interactive applications may have user inputs at random time intervals, i.e., varying TOP-APP loads. MobiRL also works well for them.

5.4.4 Performance for Varying Background Loads. In Figure 10, we test how MobiRL performs with varying background loads by launching a variable number of background applications. TikTok is used as the TOP-APP (active app on the screen). The interval of swipe action is set to 5 s. When 0, 3, 6, 9, and 12 background applications are launched, MobiRL outperforms Hyper Boost by

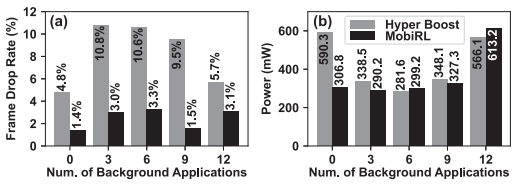


Fig. 10. MobiRL vs. Hyper Boost w/ varying background loads.

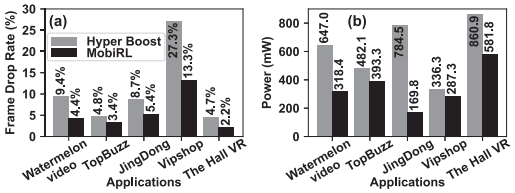


Fig. 11. MobiRL vs. Hyper Boost w/ unseen applications.

3.4%, 7.8%, 7.3%, 8.0%, and 2.7% in terms of frame drop rate, respectively. MobiRL can save power by 48%, 14.2%, and 6.0% when 0, 3, and 9 background applications are launched, respectively. When the system has a high load, MobiRL prioritizes UI smoothness. When 6 and 12 background applications are launched, MobiRL only incurs 6.3% and 8.3% higher power consumption, respectively.

5.4.5 Performance for Unseen Applications. We study how MobiRL performs for applications that are not used to train MobiRL's DDPG model in Figure 11. We evaluate MobiRL using Watermelon video [41], TopBuzz [39], JingDong [40], and Vipshop [42]. For these unseen applications, MobiRL outperforms Hyper Boost by 5.0%, 1.4%, 3.3%, and 14.0% in terms of frame drop rate and saves power by 50.8%, 18.4%, 78.4%, and 14.6%, respectively. Furthermore, we evaluate MobiRL using a VR application (Hall VR [57]). This application is resource intensive. It constantly processes motion sensor data and renders the scene in real time, requiring lots of computing/memory resources. It is also performance intensive and needs to respond to users promptly. For this application, MobiRL outperforms Hyper Boost by 2.5% in terms of frame drop rate and 32.4% in terms of power consumption. MobiRL achieves a lower frame drop rate and power consumption in these unseen cases, showing that MobiRL is generalizable and performs stably in various cases.

5.4.6 MobiRL's Performance During Runtime. We further show how MobiRL performs in detail. In Figure 12, we use TikTok as the TOP-APP and do not launch any background applications. We use adb to simulate swipe actions with a 5-second time interval. Figures 12(a) and 12(b) illustrate the frame rendering time and power consumption during the scheduling process, respectively. Figures 12(c) through 12(f) show how MobiRL schedules the CPU/GPU frequency limits for achieving ideal UI smoothness and power consumption. It schedules threads according to the CPU resources required by tasks. Android OS schedules TikTok threads on CPU clusters 0 and 1 for power efficiency for these two cases.

MobiRL can achieve higher UI smoothness and low power consumption simultaneously. As shown in Figure 12(a), during the 100-second frequency scheduling period using MobiRL and Hyper Boost, MobiRL has a lower frame drop rate (more smooth UI)—the frame drop rates of the TOP-APP scheduled by MobiRL and Hyper Boost are 1.3% and 6.3%, respectively. In Figure 12(b), the average power consumption for MobiRL is 372.4 mW, whereas Hyper Boost has an average power consumption of 561.2 mW. MobiRL saves 33.6% power compared with Hyper Boost.

MobiRL can quickly schedule CPU/GPU frequency limits to achieve the frequency limit configuration that can maximize the long-term rewards within seconds. MobiRL schedules frequency starting from time point 0. As shown at the beginning of Figures 12(d) and 12(f), MobiRL promptly conducts four consecutive frequency scaling actions within 1.5 seconds to raise the lower frequency limits of CPU cluster 1 and the GPU. Finally, the lower frequency limit of CPU cluster 1 is raised from 960 MHz to 1,670.4 MHz, and the lower frequency limit of GPU is raised from 315 MHz to 676 MHz. A higher CPU or GPU frequency increases power consumption but does not further reduce the frame drop rate. By monitoring the mobile system

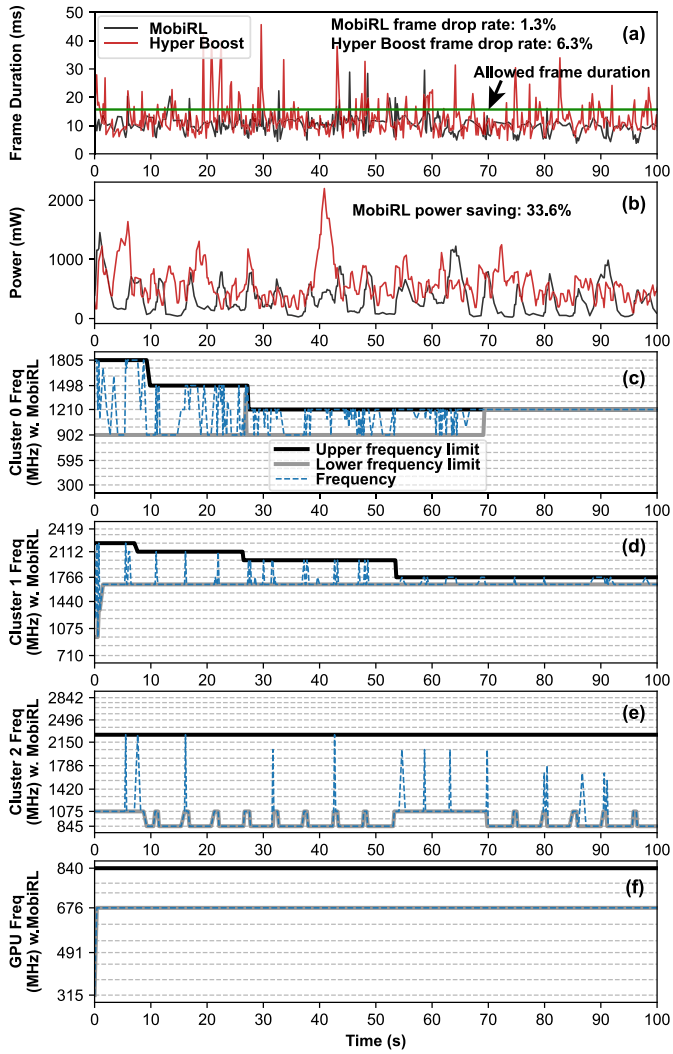


Fig. 12. MobiRL's performance in reality for TikTok with swipes at a 5-second time interval. In (c), (d), (e), and (f), the frequency and upper and lower frequency limits can be one of the predefined discrete values, indicated by the horizontal dashed lines.

status, MobiRL raises the lower frequency limits to values that can minimize potential frame drops while avoiding high power consumption. MobiRL has fast responsiveness. In MobiRL, the system monitor collects data when 10 frames are rendered (Section 4.3). Thus, when FPS is 60, the theoretically minimum scheduling interval of MobiRL is 1 s/60 $\times$10 = 167 ms. MobiRL can reduce power consumption by scaling down the frequency limits while ensuring UI smoothness. As shown in Figures 12(c) through 12(e), MobiRL scales down the upper frequency limits of CPU clusters 0 and 1 and the lower frequency limit of CPU cluster 2 for lower power consumption.

Moreover, during MobiRL's scheduling process, there are no frequent changes in the frequency limits. This is because MobiRL tends to optimize UI smoothness and power consumption by selecting a configuration of frequency limits that yields the highest long-term rewards rather than making real-time frequency adjustments.

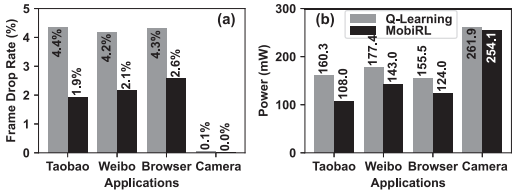


Fig. 13. MobiRL vs. Q-Learning w/ varying applications.

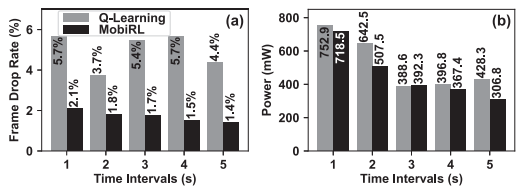


Fig. 14. MobiRL vs. Q-Learning w/ varying TOP-APP loads.

5.5 MobiRL vs. Other ML-based Scheduler for Mobile

In this section, we compare MobiRL with a recent study [44] using Q-Learning for CPU frequency scheduling based on estimated CPU loads. We denote this work as Q-Learning. Q-Learning takes the estimated CPU loads and current CPU frequency as inputs and outputs the CPU frequency that meets the CPU loads with relatively lower power consumption. During online training, Q-Learning learns to adjust CPU frequency to reduce power consumption, aiming to minimize a cost function defined as the device's power consumption. We evaluate MobiRL and Q-learning as follows.

First, we evaluate MobiRL and Q-Learning using workloads that have different TOP-APPs, including Taobao, Weibo, Browser, and Camera. There are no background applications in these workloads. The time interval between swipe/tap actions is set to 1 s for these TOP-APPs. The experimental results are in Figure 13. Compared with Q-Learning, MobiRL can achieve a lower frame drop rate and power consumption simultaneously for these TOP-APPs. MobiRL achieves a 2.5%, 2.1%, 1.7%, and 0.1% lower frame drop rate and 32.6%, 19.4%, 20.3%, and 3.0% lower power consumption for Taobao, Weibo, Browser, and Camera, respectively. The underlying reason is that MobiRL holistically schedules both the CPU and GPU frequency, achieving better scheduling results than Q-Learning, which only schedules the CPU frequency.

Second, we evaluate MobiRL and Q-Learning using workloads with varying TOP-APP loads. We use TiKTok as the TOP-APP and control its load by changing the swipe time interval. A shorter swipe time interval indicates a higher load. There are no background applications in these workloads. The experimental results are illustrated in Figure 14. Our design outperforms the Q-Learning approach in general. Moreover, we show that the Q-Learning approach may outperform our design in one aspect but cannot simultaneously have better solutions on both power and frame drop rate (i.e., UI smoothness). For example, in Figure 14, when the swipe time interval is 3 s, Q-Learning achieves 1.0% lower power consumption but incurs a 3.7% higher frame drop rate than MobiRL. The underlying reason is that Q-Learning uses the mobile device's power consumption as the reward; thus, it optimizes only the power consumption and does not consider the UI smoothness. It tries to achieve a lower power consumption by scaling down the frequency, which can lead to frequency under-provision and incur slow UI responsiveness. By contrast, MobiRL optimizes the UI smoothness and power consumption simultaneously because it considers both the frame drop rate and power consumption in its reward function.

Third, we evaluate MobiRL and the Q-Learning approach using workloads with varying background loads. We launch TikTok as the TOP-APP and launch a varying number of background applications. The interval of the swipe action is set to 5 s. The experimental results are in Figure 15. We show that MobiRL can satisfy the loads with lower power consumption, especially when the background load is heavy. For example, in Figure 15, when there are nine background applications in the workload, MobiRL has 3.1% lower frame drop rate and 47.1% lower power consumption. Besides, in the cases where there are 12 background applications, MobiRL also provides significant benefits. The underlying reason is that MobiRL has more input features, so it

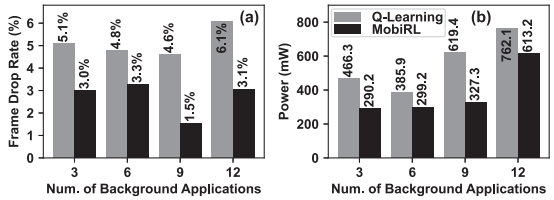


Fig. 15. MobiRL vs. Q-Learning w/ varying background loads.

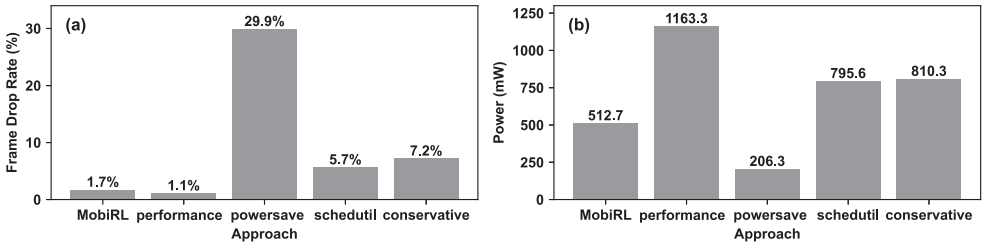


Fig. 16. MobiRL vs. four default governors in the CPUFreq subsystem.

can identify performance issues and make accurate frequency scheduling decisions accordingly, outperforming the other approach. As shown in Table 3, MobiRL's input features include the task load of the TOP-APP and the CPU cluster ID that the TOP-APP is running on. With these two features, MobiRL can satisfy the TOP-APP's load by accurately scheduling the frequency of the CPU cluster that the TOP-APP is running on using the minimum power consumption. By contrast, the Q-Learning approach relies on the estimated CPU loads for scheduling. It schedules frequency for all applications, even those that are not important for improving the UI responsiveness, leading to higher power consumption, especially when the background load is heavy.

5.6 MobiRL vs. Default Frequency Scaling Governors in CPUFreq Subsystem

The Android OS uses the CPUFreq subsystem to support CPU performance scaling [2]. It provides several default frequency scheduling governors that scale frequency within the frequency limits, e.g., performance, powersave, schedutil, and conservative. They have straightforward scheduling policies and are widely used on mobile systems. We compare MobiRL with them to show MobiRL's effectiveness over default mechanisms. In our experiments, we use TikTok as the TOP-APP and set the swipe time interval to 1 s. There are no background applications in the workload. The experimental results are in Figure 16.

The performance governor adjusts the frequency to the upper frequency limits for higher performance. As illustrated in Figure 16, compared with our approach, it further reduces the frame drop by 0.6% but incurs a 2.3× higher power consumption. The powersave governor adjusts the frequency to the lower frequency limits to save power. It saves power consumption by 59.8% compared with MobiRL but incurs a 28.2% higher frame drop rate. Both performance and powersave cannot optimize the frame drop rate and power consumption simultaneously. The schedutil governor scales CPU frequency dynamically based on the estimated CPU utilization. More specifically, it adjusts the CPU frequency dynamically to control the estimated CPU utilization close to 80%. If the estimated CPU utilization is above 80%, it dynamically scales up the CPU frequency if possible; otherwise, if utilization is below 80%, it scales down the CPU frequency. The conservative governor also schedules based on the estimated CPU utilization, but it has a larger scheduling time interval and smaller frequency scheduling steps. Compared with

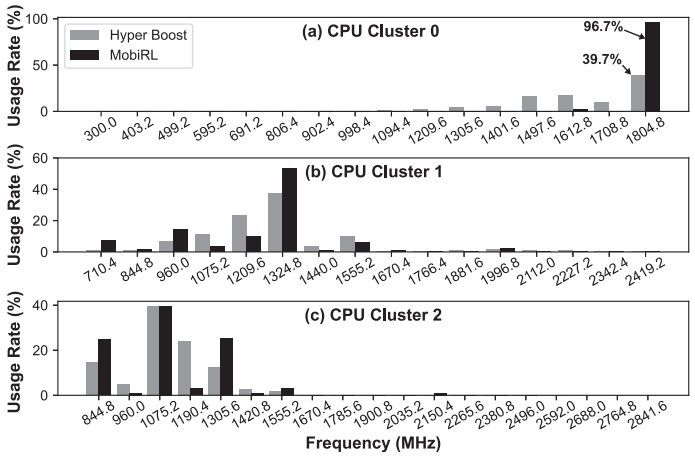


Fig. 17. CPU frequency usage.

schedutil and conservative, MobiRL achieves a 3.9% and 5.5% lower frame drop rate, respectively, and achieves a 35.5% and 36.7% lower power consumption, respectively. The schedutil performs better than conservative because it is more flexible and works better on mobile devices with dynamically changing loads. In general, MobiRL can optimize the frame drop rate and power consumption simultaneously and outperforms these governors in the CPUFreq subsystem.

5.7 CPU Frequency Usage: Why MobiRL Works Effectively

We analyze the usage of CPU frequency to study why MobiRL can optimize the UI smoothness and power consumption simultaneously. Figure 17 shows the CPU frequency usage for 50 workloads where MobiRL outperforms Hyper Boost. Each workload is run for 1 minute. The total time for the plots is 50 minutes. Figures 17(a) through 17(c) show the frequency usage for CPU clusters 0, 1, and 2, respectively. The x-axis represents each CPU cluster's predefined discrete frequency values; the y-axis represents the proportion of time during the entire scheduling process that the CPU cluster stays at a specific frequency. As shown in Figure 17, when MobiRL is used for scheduling, CPU cluster 0 stays at 1,804.8 MHz (the highest frequency value) for 96.7% of the entire scheduling process. By contrast, with Hyper Boost, CPU cluster 0 stays at 1,804.8 MHz for only 39.7% of the scheduling process. We learn from Figure 2 (Section 2.2) that the cores in CPU cluster 0 are power-saving ones. The high usage of CPU cluster 0's highest-frequency value means that MobiRL makes good use of the power-saving cores in CPU cluster 0. Therefore, MobiRL can optimize the UI smoothness and power consumption simultaneously.

5.8 Overhead

MobiRL has a low overhead. The time required for getting model input features is 2.4 ms. The Actor network runs on the CPU when MobiRL is deployed on the mobile system. It takes 6.7 ms on average to forward the input parameters to the model and obtain the output. The memory usage of MobiRL is 7 MB. In terms of CPU usage, the model prediction and frequency scheduling account for 10.5% utilization of a little core (Cortex-A55) on the experimental platform. As MobiRL does not send or receive network requests, it does not consume network bandwidth. As for storage overhead, the model consumes only 13 KB.

Moreover, we evaluate MobiRL's impact on background applications including Gmail, Google Maps, Amazon Shopping, and WeChat. They run in the background and are not on the screen. For

each background application, we use TikTok as the TOP-APP and leverage simpleperf to measure the background application's IPC with and without using MobiRL, respectively. The experimental results show that for Gmail [53], Google Maps [54], Amazon Shopping [55], and WeChat [56], enabling MobiRL decreases their IPC by 2.9%, 3.6%, 3.3%, and 4.8%, respectively, which is negligible and does not impact the user experience in practice.

6 Discussion and Future Work

User-specific Model Training. It is necessary to ensure that MobiRL performs consistently and stably on diverse devices. Sending experiences from mobile devices to the cloud is practical for user-specific model training. The cloud server can classify users and train user-specific models accordingly. On-device training leads to high power consumption and cannot guarantee consistent model quality across all devices.

Generalization. If mobile devices have new hardware that significantly differs from existing ones (e.g., a new processor with a different number of cores), the RL model needs to be retrained for accurate scheduling due to the hardware changes. Transfer learning can be used to retrain the model with a small training overhead. MobiRL is generalizable. Its model can be used on devices with similar hardware configurations. This is because the relationship between OS features, rendering time, and computational resources captured by MobiRL does not change drastically on these devices. Moreover, MobiRL can effectively schedule resources even for devices with different hardware through low-overhead transfer learning (Section 5.3).

One-for-all vs. One-for-each Category. Training a model for each category of applications may lead to a lower frame drop rate and lower power consumption compared to using a single model to handle all applications. For instance, using a model tailored to video playback applications (e.g., TikTok, YouTube, etc.) would result in higher accuracy and better scheduling results compared to scheduling in a one-for-all manner.

7 Related Work

DVFS on Mobile Systems. Several related studies propose ML-based **Dynamic Voltage Frequency Scaling (DVFS)** on mobile systems [11, 43–46]. The study in [44] predicts the CPU loads and employs Q-learning to adjust CPU frequency to minimize energy consumption. ML-Gov [11] leverages an offline linear regression model to estimate CPU/GPU frequencies that maximize energy savings with minimal FPS degradation. AHDL [45] classifies the TOP-APP into several types (e.g., computing intensive, memory intensive, etc.) and allocates computing resources based on predefined rules. Yet, the studies in [44, 45] only focus on managing CPU frequency and fail to manage GPU frequency, leading to sub-optimal performance and higher power consumption. Moreover, the studies in [44, 45] rely on limited features for scheduling, such as CPU load (the work in [44]) or predicted application types (AHDL [45]). They do not consider other critical OS runtime features like cache misses, IPC, and so forth. Therefore, they cannot comprehensively detect system performance issues and schedule the frequency accurately. The studies in [44] has only one optimization goal, i.e., energy consumption. This might result in frame drops in mobile systems due to frequency under-provision. In terms of generalization, these studies [11, 44, 45] cannot be easily generalized across new platforms because they use offline ML models (e.g., tree-based piecewise linear models in ML-Gov [11], k-NN-based power predictor in work [44], and CNN-based application classifier in AHDL [45]). When deployed on new platforms, they require offline data collection and model retraining.

By contrast, MobiRL learns from more critical OS runtime features, so it can identify performance issues and make accurate frequency scheduling decisions accordingly. It manages

Table 6. Representative DVFS Studies on Data Center Servers

Studies	Target component	Goals	Core mechanism	Differences from mobile DVFS
Work in [47]	GPU core and memory frequency	Estimate GPU kernel performance to conserve energy through core/memory DVFS	A GPU pipeline analysis model that predicts GPU kernel execution time under different core and memory frequencies	Different architecture: No consideration of heterogenous architecture on mobile devices
Δ-DVFS [48]	Multi-processor voltage and frequency	Approach maximum power efficiency step by step through voltage and frequency scaling	A profile-then-select strategy that profiles the performance-power characteristic curve (PPCC) and achieves the optimal voltage/frequency step by step	Different optimization goals: Optimizing power efficiency, no consideration of minimizing power consumption
wDVS [49]	Multi-processor voltage and frequency	Find the optimal execution cycles for output quality maximization through frequency and voltage scaling	A heuristic frequency scaling methodology to optimally decide the processor execution cycles for applications with diminishing return	Different optimization goals: Optimizing output quality for applications with diminishing return, no consideration or minimizing power consumption
Work in [50]	GPU frequency and voltage	Estimate GPU power consumption for DVFS management and power bottleneck analysis	A DVFS-aware GPU power model that predicts GPU power consumption using hardware performance events	Different architecture: No consideration of heterogenous architecture on mobile devices

both CPU and GPU frequency to achieve better UI responsiveness and lower power consumption simultaneously. Moreover, MobiRL does not require offline data collection; it uses reinforcement learning to autonomously explore the scheduling exploration space and learn to optimize UI responsiveness and power consumption. And using transfer learning can reduce MobiRL's training overhead when deployed on new platforms. In Section 5.5, we qualitatively and quantitatively compare MobiRL to the prior work [44] that employs Q-learning for CPU scheduling. MobiRL significantly reduces the frame drop rate and power consumption compared with [44].

DVFS on Data Center Servers. DVFS on data center servers is a well-studied approach [47–50]. Table 6 summarizes several typical DVFS studies on data center servers and shows their differences from DVFS on mobile devices. Generally, mobile systems often use the big.LITTLE heterogeneous computing architecture, which provides power-saving little cores and high-performance, power-hungry big cores. DVFS mechanisms on mobile devices focus on effectively leveraging this specific computing architecture to achieve better performance and lower power consumption. And, though some studies are also conducted on data center servers with heterogeneous processors, we find that some of their scheduling behaviors are coarse grained (e.g., longer scheduling time intervals, task-level partitioning/scheduling [58, 59], etc.) than our approach on mobile systems. Besides, mobile systems are prone to be more energy efficient. On mobile systems, power is a problem that is often on top of others. By contrast, though some studies on data center servers also try to reduce the power and improve the QoS [34, 60, 61], their platforms, environments, applications, and use cases differ from mobiles. So, the solutions are also different at the root.

ML/AI for Systems. Using ML/AI is a promising approach for system optimizations. Many studies try to make OS intelligent using ML/AI technologies, e.g., resource scheduling for cloud services [14, 18, 26, 29], load balancing [17], parameter tuning for OS/system software [16, 27], VM failure mitigation [30], and so forth. Generally, these studies fall into three categories. The first category is statistical learning [13–15, 26]. The second category is deep learning [16, 17, 27, 28]. Deep learning models are data driven. They require sufficient training data for accurate prediction and generalization. The third is reinforcement learning [18, 19, 29, 30]. They can learn online from historical scheduling decisions and the feedback from the environment, making them resilient to changes in the environment and the workloads.

8 Conclusions

We present MobiRL, a reinforcement learning-based frequency scheduler for mobile systems. MobiRL autonomously learns to schedule CPU and GPU frequency for better UI smoothness and lower power consumption. Experiments on a newly released smartphone show that MobiRL outperforms the scheduler in off-the-shelf mobile smartphones. Mobile systems are everywhere now. We think our work can be valuable for system designers. Moreover, we advocate applying ML/AI techniques for OS, and optimizations can be an ideal approach for system performance. This project is supported by a well-known mobile phone company, and MobiRL has been used in mobile phone products.

Acknowledgment

We thank the reviewers, AE and EIC, for their invaluable comments. We also thank the previous student members who paid attention to this project. We thank the cooperation that supports this project. X. Dou is a student member in Sys-Inventor Lab led by L. Liu, who is the PI of this project. L. Liu is the corresponding author and co-first author of this paper.

References

[1] F. Laricchia. Market share of mobile operating systems worldwide 2012-2022. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
[2] R. J. Wysocki. CPU performance scaling. https://docs.kernel.org/admin-guide/pm/cpufreq.html
[3] Device frequency scaling—the Linux kernel documentation. https://docs.kernel.org/driver-api/devfreq.html
[4] Google. power/1.0 - platform/hardware/interfaces - Git at Google. https://android.googlesource.com/platform/hardware/interfaces/+/master/power/1.0/default/Power.h
[5] Samsung. Gamedev | Samsung developers. https://developer.samsung.com/galaxy-gamedev/overview.html
[6] OPPO. Hyper boost | OPPO developer. https://developers.oppomobile.com/newservice/capability?pagename=hyper_boost
[7] Huawei. Service Introduction-PerfGenius-Accelerate Kit | HUAWEI developers. https://developer.huawei.com/consumer/en/doc/development/HMSCore-Guides-V5/introduction-0000001054817121-V5
[8] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann. 2018. CALOREE: learning control for predictable latency and low energy. In *ASPLOS*.
[9] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *OSDI*.
[10] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO*.
[11] J. Park, N. D. Dutt, and S. Lim. 2017. Ml-gov: A machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming. In *ESTIMedia*.
[12] Qualcomm. Qualcomm Snapdragon 888 5g mobile platform | Qualcomm. https://www.qualcomm.com/products/application/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-888-5g-mobile-platform
[13] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. N. Chinya, J. Gaur, K. Sankaranarayanan, C. Lin, R. Chappell, R. Singhal, and H. Wang. 2019. Post-silicon CPU adaptation made practical using machine learning. In *ISCA*.
[14] J. Rahman and P. Lama. 2019. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *IC2E*.
[15] Z. Xie, X. Xu, M. Walker, J. Knebel, K. Palaniswamy, N. Hebert, J. Hu, H. Yang, Y. Chen, and S. Das. 2021. APOLLO: An automated power modeling framework for runtime power introspection in high-volume commercial microprocessors. In *MICRO*.
[16] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. 2019. iBTune: Individualized buffer tuning for large-scale cloud databases. In *VLDB Endowment*.
[17] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer. 2020. Machine learning for load balancing in the Linux kernel. In *APSys*.
[18] R. Nishtala, V. Petrucci, P. M. Carpenter, and M. Själander. 2020. Twig: Multi-agent task management for colocated latency-critical cloud services. In *HPCA*.
[19] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. 2016. Continuous control with deep reinforcement learning. In *ICLR*.
[21] On-device training with Tensorflow lite. https://www.tensorflow.org/lite/examples/on_device_training/overview
[22] Analyze with Profile GPU Rendering | App quality | Android developers. https://developer.android.com/topic/performance/rendering/profile-gpu
[23] kernel/sched/boost.c - kernel/msm - Git at Google. https://android.googlesource.com/kernel/msm/+/refs/tags/android-10.0.0_r0.4/kernel/sched/boost.c
[24] S. A. Siewior, R. Russell, S. Vaddagiri, A. Raj, J. Schopp, and T. Gleixner. CPU hotplug in the Kernel – The Linux Kernel documentation. https://docs.kernel.org/core-api/cpu_hotplug.html
[25] Performance boost at game loading time | Android Open Source Project. https://source.android.com/docs/core/perf/boost
[26] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*.
[27] Y. Zhang and Y. Huang. 2019. 'Learned': Operating systems. In *ACM SIGOPS Operating Systems Review*.
[28] R. Bitirgen, E. Ipek, and J. F. Martínez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*.
[29] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *OSDI*.
[30] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. K. Govindaraju, X. Li, Q. Lin, G. L. Shafriri, and M. Chintalapati. 2020. Predictive and adaptive failure mitigation to avert production cloud VM interruptions. In *OSDI*.
[31] H.-D. Cho, K. Chung, and T. Kim. 2012. Benefits of the big.LITTLE Architecture. *SAMSUNG Electronics White Paper*.
[32] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv 1412.6980*.
[33] Hyperopt documentation. http://hyperopt.github.io/hyperopt/
[34] L. Liu, X. Dou, and Y. Chen. 2023. Intelligent resource scheduling for co-located latency-critical services: A multi-model collaborative learning approach. In *FAST*.
[35] TikTok: Vidéos, LIVE, Musique - Apps on Google Play. https://play.google.com/store/apps/details?id=com.zhiliaoapp.musically&hl=en_US&pli=1
[36] Android Debug Bridge (adb) | Android Studio | Android developers. https://developer.android.com/tools/adb
[37] Weibo - Apps on Google Play. https://play.google.com/store/apps/details?id=com.sina.weibo&hl=zh&gl=US
[38] Taobao - Apps on Google Play. https://play.google.com/store/apps/details?id=com.taobao.taobao&hl=en_US
[39] TouTiao APP. https://app.toutiao.com/
[40] Jingdong - Apps on Google Play. https://play.google.com/store/search?q=jingdong&c=apps&fpr=false&gl=FR&hl=en_US
[41] Watermelon video. https://sj.qq.com/appdetail/com.ss.android.article.video?fromcase=10003&from_wxz=1
[42] ViPSHOP - Apps on Google Play. https://play.google.com/store/apps/details?id=me.GooApp.Apps.ViPSHOP&hl=en_US&gl=FR
[43] J. Haj-Yahya, M. Alser, J. S. Kim, A. G. Yaglikçi, N. Vijaykumar, E. Rotem, and O. Mutlu. 2020. SysScale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors. In *ISCA*.
[44] S. A. L. Carvalho, D. C. Cunha, and A. G. Silva-Filho. 2019. Autonomous power management in mobile devices using dynamic frequency scaling and reinforcement learning for energy minimization. In *Microprocessors and Microsystems*.
[45] S. Dey, S. Saha, A. K. Singh, and K. D. McDonald-Maier. 2021. Asynchronous Hybrid Deep Learning (AHDL): A deep learning based resource mapping in DVFS enabled mobile MPSoCs. In *WF-IoT*.
[46] S. Kim and Y. J. Kim. 2015. GPGPU-Perf: Efficient, interval-based DVFS algorithm for mobile GPGPU applications. In *The Visual Computer*.
[47] Q. Wang and X. Chu. 2020. GPGPU performance estimation with core and memory frequency scaling. In *IEEE TPDS*.
[48] Y. Yao and Z. Lu. 2020. Pursuing extreme power efficiency with PPCC guided NoC DVFS. In *IEEE TC*.
[49] H. Yu, Y. Ha, B. Veeravalli, F. Chen, and H. El-Sayed. 2021. DVFS-based quality maximization for adaptive applications with diminishing return. In *IEEE TC*.
[50] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas. 2018. GPGPU power modeling for multi-domain voltage-frequency scaling. In *HPCA*.
[51] OPPO browser review. https://communityin.oppo.com/thread/48606
[52] Camera for OPPO. https://play.google.com/store/apps/details?id=com.perffectselfie.thelastedcamera.oppocamerastyle&hl=en_US
[53] Gmail - Apps on Google Play. https://play.google.com/store/apps/details?id=com.google.android.gm&hl=en_US
[54] Google Maps - Apps on Google Play. https://play.google.com/store/search?q=google%20maps&c=apps&hl=en_US

[55] Amazon Shopping - Apps on Google Play. https://play.google.com/store/apps/details?id=com.amazon.mShop.android.shopping&hl=en_US
[56] WeChat - Apps on Google Play. https://play.google.com/store/search?q=wechat&c=apps&hl=en_US
[57] The hall VR. https://cecropia.github.io/thehallaframe/
[58] J. Chen, A. Schranzhofer, and L. Thiele. 2019. Energy minimization for periodic real-time tasks on heterogeneous processing units. In *IPDPS*.
[59] C. Yang, J. Chen, T. Kuo, and L. Thiele. 2009. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *DATE*.
[60] Lei Liu, Yong Li, Chen Ding, Hao Yang, and Chengyong Wu. 2006. Rethinking memory management in modern operating system: Horizontal, vertical or random? In *IEEE TC*.
[61] L. Liu, S. Yang, L. Peng, and X. Li. 2019. Hierarchical hybrid memory management in OS for tiered memory systems. In *IEEE TPDS*.

Received 24 October 2023; revised 1 April 2024; accepted 20 May 2024