

CPU frequency scheduling of real-time applications on embedded devices with temporal encoding-based deep reinforcement learning

Ti Zhou, Man Lin*

Department of Computer Science, St. Francis Xavier University, Nova Scotia, Canada

ARTICLE INFO

Keywords:

Energy management for small devices
Reinforcement learning with temporal encoding
Soft-deadline constrained application

ABSTRACT

Small devices are frequently used in IoT and smart-city applications to perform periodic dedicated tasks with soft deadlines. This work focuses on developing methods to derive efficient power-management methods for periodic tasks on small devices. We first study the limitations of the existing Linux built-in methods used in small devices. We illustrate three typical workload/system patterns that are challenging to manage with Linux's built-in solutions. We develop a reinforcement-learning-based technique with temporal encoding to derive an effective DVFS governor even with the presence of the three system patterns. The derived governor uses only one performance counter, the same as the built-in Linux mechanism, and does not require an explicit task model for the workload. We implemented a prototype system on the Nvidia Jetson Nano Board and experimented with it with six applications, including two self-designed and four benchmark applications. Under different deadline constraints, our approach can quickly derive a DVFS governor that can adapt to performance requirements and outperform the built-in Linux approach in energy saving. On *Mibench* workloads, with performance slack ranging from 0.04 s to 0.4 s, the proposed method can save 3%–11% more energy compared to *Ondemand*. AudioReg and FaceReg applications tested have 5%–14% energy-saving improvement. We have open-sourced the implementation of our in-kernel quantized neural network engine. The codebase can be found at: <https://github.com/coladog/tinyagent>.

1. Introduction

1.1. The context of energy saving problem

Soft-deadline periodic real-time systems are commonly seen in many IoT/CPS/smart city/wearable computing systems to provide ubiquitous and rich services. The following are some sample systems reported in IEEE IoT Magazine.

- **Smart dairy farm:** deploying sensors on cows to collect the biological information for the purpose of classifying their status [1].
- **Pest detection in precision agriculture:** using cameras to photograph the crop to detect the location of pest [2].
- **Covid-19 screening and detection:** putting sensors on drones to collect biological information and detect Covid-19 infection [3].
- **Smart irrigation:** collecting weather, crop growth, and soil conditions to analyze and predict whether the soil needs irrigation [4].
- **Distance violation detection:** mounting cameras on vehicles to detect distance violation [5].

These systems normally consist of multiple small devices filled with sensors/network calls and pre-defined periodically running workloads to be completed before the next task period. Besides the performance requirement, *low power consumption* is another important QoS requirement for such small devices for the battery life.

Our goal in this work is to derive an adaptive model-free method that can save energy for such types of applications (Soft-deadline periodic CPS applications) running on small devices that have limited computing capacity.

1.2. Model-based or model-free energy saving method?

Modern small-device computing mainly relies on low-power CPUs. Dynamic Voltage and Frequency Scaling (DVFS) technology, which tunes the CPU's voltage (V) and frequency (f) on-demand, is a popular way to address such needs. It is a classical problem in the real-time system community to schedule the CPU performance as energy-efficient as possible while satisfying the computational performance need. Many research works have been performed in this area.

* Corresponding author.

E-mail address: mclin@stfx.ca (M. Lin).

<https://doi.org/10.1016/j.sysarc.2023.102955>

Received 30 December 2022; Received in revised form 20 June 2023; Accepted 25 July 2023

Available online 28 July 2023

1383-7621/© 2023 Elsevier B.V. All rights reserved.

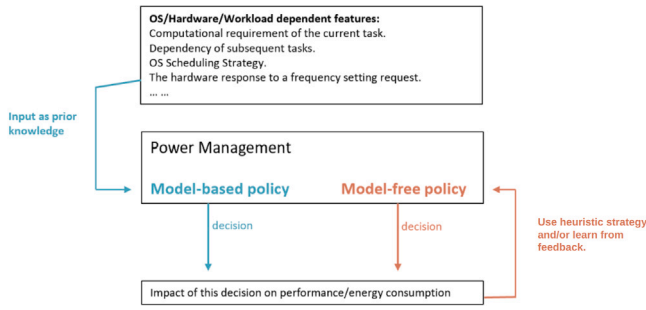


Fig. 1. Model-based and model-free power management policy.

Power management algorithms can be classified into two categories: model-based and model-free. Their differences are shown in Fig. 1. Model-based algorithms need a specification of the system with prior knowledge of the tasks [6–17], including the attributes of the tasks such as worst-case execution time (WCET) [7,9,17], deadline [7,9–16], task period [8–10,13,15], possible priorities [8,10], or the relations of the tasks specified as a DAG task graph with the communication cost and precedence order of the tasks [6,7,9–12,14–16]. The mathematical model of the tasks, together with the machine architecture features, are used to find an optimal strategy for power management. On the other hand, a model-free DVFS algorithm does not require the input of task-specific information. They either make decisions on frequency scaling based on some predefined assumptions (e.g. CPUFreq governors in Linux assume the future utilization is the same as the current CPU load measured by the performance counter), or collect feedback from the system during operation to adapt to the characteristics of the environment [18–29].

The Model-based approach is limited to systems with a known explicit model. The detailed timing behavior of the tasks running on a specific target machine needs precise knowledge of the tasks and the architecture of the target machine, which could affect the timing behavior of the tasks. The system model (tasks + device) is normally hard to obtain. Therefore, such methods are only adopted for hard real-time systems that are safety-critical when a careful analysis of the system and tasks is necessary.

1.3. Problem statement and proposed approach

Model-free DVFS methods are the common practice in general-purpose operating systems, such as Linux. Their power management policy can be applied to any task. We adopt a model-free power management method, given that many CPS applications are deployed to systems without prior knowledge of an explicit task model and device model.

We first look at the common structure (architecture) of a model-free DVFS algorithm, which includes the following.

1. Use performance counters to construct the current system features.
2. Use a model for inference.
3. Select the next period's CPU frequency based on the model's output.

The Linux built-in methods (*Ondemand*, *Conservative* and *Schedutil*) are a classic application of the above architecture. They sample the CPU utilization of the past period as a system feature, predict the constant computational demand for the next period, and then update the CPU frequency based on some heuristic rules.

Recent research efforts have focused on how to improve this architecture. Methods include: enabling more performance counters to build

complex system features [19,20,27], using powerful models for prediction [18–21,24], designing better control rules [18–21], or learning control policy based on reinforcement learning [24–27,30].

For general systems, where the arrival time of tasks is highly variable and diverse, improving power management strategies can be difficult. However, periodic systems are special from a temporal point of view, as they run a set of pre-defined tasks periodically. Can we exploit this feature to develop a better CPU power management policy for a system?

This work focuses on how to **deriving efficient model-free methods for periodic tasks with a soft-deadline running on small devices**.

1.3.1. Study the limitation of existing model-free DVFS governors through profiling

To avoid reinventing the wheel, our first step in approaching this problem is to study the behavior of existing Linux built-in governors through kernel-level profiling. We want to study if simply tuning the existing built-in governor will result in a better energy-efficient governor that is tailored for the periodic tasks with a soft deadline. In tuning the power management strategy for periodic soft real-time systems commonly found in contemporary embedded systems, we observe that the structure of existing DVFS governors can be ineffective for three frequently occurring system patterns. To be more specific, CPU cores only have coarse-grained voltage/frequency level (limited) support, which is typical for small devices, cores can experience unbalanced load distributions, and tasks can have internal slack.

This is mainly because existing built-in DVFS algorithms focus on the short-term computational characteristics of the system, whereas a good strategy for achieving an overall optimal solution often requires macroscopic knowledge: what has been computed in the past and what will be computed in the future.

This motivates us to find alternative methods to obtain a model-free governor rather than attempting to tune the existing governor for energy saving for the particular class of workload (soft-deadline periodic tasks) that we are interested in. The problem can be thought of as a sequence of decision-making of assigning a frequency to the CPU at each decision point. Reinforcement learning is a natural strategy to apply in the absence of an explicit model of the tasks to help with decision-making.

1.3.2. Using reinforcement learning with temporal encoding to derive model-free governors

A reinforcement learning approach will learn a DVFS inference model (a governor) through the feedback of the sequence frequency decisions, including its effect on the system load, timing, and energy consumption. Note that every frequency assignment in the sequence makes a difference in the amount of energy used at the end and how long it takes to complete the task. So, a pool of time series data will form the foundation of the learning process.

One of the most crucial design issues for reinforcement learning is the state representation. Automatic encoding of features from raw input data has been the main focus of recent artificial intelligence. In a previous work [30], RNN was used to encode time series automatically. Until now, this approach has relied on complex computations and lengthy learning from large amounts of data. Another drawback of automatic encoding is its poor interpretability.

We choose an explicit encoding for the temporal information in this work to achieve higher interpretability and to ease the burden of model learning, which is important for small devices with limited computing resources.

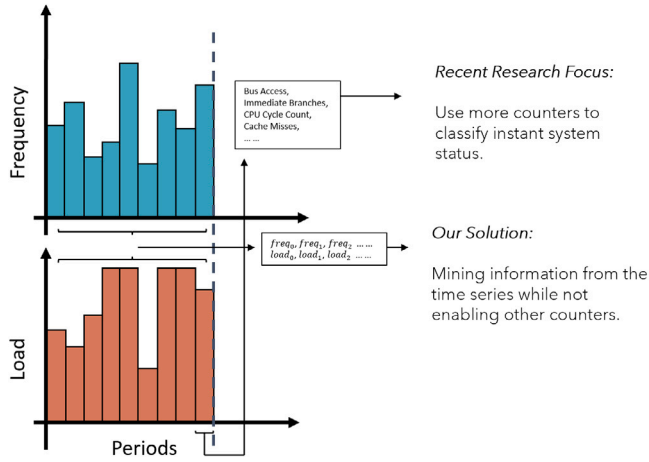


Fig. 2. How to understand the computing demand for the next period?

1.4. Contribution

We choose Nvidia Jetson Nano 2 GB board as our experiment testbed.

Our first contribution is to study the limitation of existing Linux built-in DVFS methods through profiling. To analyze the CPU frequency control policy, we designed and implemented a low-overhead in-kernel profiler to collect the complete ms-level runtime data of the Linux CPUFreq governor. With this profiler, we identified three scenarios when the Linux built-in DVFS methods are ineffective.

- Target devices only support coarse-grained voltage (or frequency) levels, which are common for small devices. For example, both Raspberry Pi 4B+ and Nvidia Jetson Nano Board 2 GB only support two V/f levels.
- Multi-core architectures have unbalanced CPU load distribution.
- There exist internal slacks within the workload caused by IO calls (sensors, cameras, microphones, network interactions, etc.).

Our second contribution is to design a reinforcement learning-based frequency governor under the CPUFreq framework with a temporal encoded system state to better address the above challenges.

We only use CPU load (utilization) for decision-making as standard built-in Linux governors. However, the instance information of CPU load (utilization) cannot provide sufficient distinction for the CPU to make different frequency scaling choices that consider the computation requirement of the current tasks. Therefore, the state construction of our method is based on the temporal sequence of the load instead of the load value at the previous instance, as shown in Fig. 2. The temporal sequence is encoded as a vector reflecting the progress of task execution for the RL governors to make a decision. The temporal encoding enables a reinforcement learning method to efficiently understand the workload from a macro perspective by mining the timing sequence even without the explicit model of the workload. The domain-assisted encoding is different from Standard RNN, where the encoding vector is learned, making on-device learning infeasible if the application workload sequence is long. Experiment results show that the encoding, together with the reinforcement learning method, is effective for finding good DVFS scaling strategies through on-device reinforcement learning.

The following summarizes the advantage of our energy-saving framework with on-device learning for periodic CPS applications with soft deadlines.

- **High Interpretability.** We carefully designed the system state used in reinforcement learning to include features that intuitively contain valuable information.

- **Low Deployment Complexity.** Excessive training time due to large amounts of trial and error can lead to increased deployment complexity. We achieve low training time by properly designing the state to contain explicit and useful information from a human expert perspective to help the model quickly link the cause and the result. In our experiments, the proposed method learns a good DVFS policy with only three hundred workload runs.
- **Low Resource Overhead.** Similar to [30], our work only implements the decision inference component at the kernel level. The learning component is implemented at the user level with data collected by the in-kernel profiler. Thus, the kernel state is only burdened with little inference overhead. Our work further reduces the overhead by applying quantization [31], with which the kernel can avoid floating-point calculation. In our experiments, an inference of the proposed method takes only 25.62 us with 1.479 GHz on average.

2. Background

2.1. Dynamic power consumption

The dynamic power consumption P_d of a CMOS circuit is determined by [32]:

$$P_d = \alpha \times C \times V_{dd}^2 \times f, \quad (1)$$

where V_{dd} is the supply voltage, f is the clock frequency, α is the switching activity level, and C is the capacitance of the circuit.

Supply voltage V_{dd} and clock frequency f are related as follows:

$$f \propto \frac{\beta(V_{dd} - V_{th})^2}{V_{dd}}, \quad (2)$$

where V_{th} is the threshold voltage, and β is a technology-dependent constant. For $V_{dd} \gg V_{th}$ and β closed to 1, clock frequency f is roughly proportional to V_{dd} . In this case, the dynamic power consumption is proportional to V_{dd} and f through a cubic relationship:

$$P_d \propto V_{dd}^3 \propto f^3 \quad (3)$$

Dynamic power consumption reduces with the frequency following a cubic relationship, whereas execution time increases following a nearly linear relationship. This property determines that, for the same task, it can be executed with less energy at a lower frequency/voltage. Suppose only the frequency is reduced, but the voltage stays the same. Due to the reduced current in this scenario, the instantaneous power consumption is lower. However, since the running task will take longer to complete, the total amount of energy used to complete one task will not be lowered.

It is worth mentioning that even if the energy consumption cannot be reduced, the heat generation of the system will be reduced due to the decrease in the instantaneous power. However, if hardware costs permit, it is desirable to regulate the voltage and frequency together.

2.2. Static power consumption

Static power consumption P_s represents 20%–40% of the power budget of microprocessors in modern fabrication technologies [32], it is determined by:

$$P_s = I_{static} \times V_{dd} \quad (4)$$

I_{static} is primarily due to subthreshold leakage current, and gate leakage current [32], which are affected by the supply voltage V_{dd} . Lowering V_{dd} can save both dynamic power consumption and static power consumption. When the CPU is idle, lowering the CPU voltage can effectively save energy. On Jetson Nano Board 2 GB, when the system is idle, by setting the CPU to the lowest voltage, the board-wide power consumption (measured by a power meter) can be reduced by 36%.

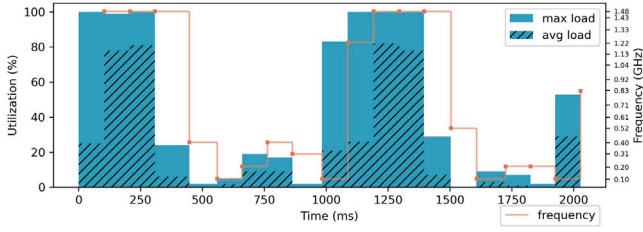


Fig. 3. Profiling of Ondemand governor using our profiler.

CPU Idle Time Management, which shuts down part of the CPU hardware function when idle, is another efficient way to reduce static energy consumption. However, the more CPU functions are turned off, the more time and energy are required to switch back to a normal state. Software-level algorithms need to be implemented to predict the idle duration of the CPU to select the appropriate idle state to enter. Poorly designed idle control algorithms can waste energy and lose performance at the same time.

3. Low overhead kernel profiling

The process of CPU frequency scaling can be viewed as an agent (frequency governor) observing the environment (the computing device managed by the OS that runs the work-load) and taking actions accordingly (frequency scaling). In order to better comprehend the advantages and disadvantages of different policies, we want to depict the decision-making process, which can also enhance the interpretability of a learning-based solution.

Reading kernel data via default Linux support (for example, character file systems like `sysfs`) or advanced tools (for example, `perf` [33]) often involves reading a string from a buffer/file and then extracting data from it. The two built-in Linux CPU frequency governors (*Ondemand* and *Conservative*) typically perform 100 inferences per second by default [34]. Performing `Perf`-like [33] operations at such a high rate will put a non-negligible burden on the real workload of the system, which would cause the resulting profiling data to be meaningless.

Our objective is to profile the complete in-kernel CPU tuning data at the micro-second level while ensuring low latency. Our solution involves inserting a profiler into the CPU governor that, at runtime, sends data directly to the kernel's data structures and writes data to the shared file system only at the end of the system run.

Each time the CPU governor makes an inference, our profiler collects necessary data into an array in DRAM. Specifically, our profiler writes 42 bytes of data per inference. If the governor performs 100 inferences a second, then a 14.17 MB array is sufficient to store all the information generated in one hour. At the end of the system run, the profiler writes the collected data to the shared file system accessible to the user-state. The final write-out overhead barely has any effect on the runtime workload that is being profiled.

Fig. 3 shows a visualization of *Ondemand* governor's runtime profiled by the proposed method. The system in this example runs a face recognition workload per second (the blue dash line in the figure is the task period and the deadline). At each sampling point (orange nodes in Fig. 3), the profiler records the maximum/average CPU load (utilization) among cores in the last period, and the subsequent frequency *Ondemand* governor decides to set. The timestamp of each action is precisely recorded in an ms-view.

4. Linux built-in methods: Limitations

As of Version 5.13, Linux provides three dynamic DVFS policies [34]: *Ondemand*, *Conservative*, and *Schedutil*. *Ondemand* and *Conservative*

Algorithm 1 Ondemand DVFS governor in Linux V5.13: a simplified description

- 1: F denotes the provided frequency options.
- 2: \min_f / \max_f denotes the min/max supported frequency in F .
- 3: next_f denotes the frequency to be applied.
- 4: **for** each sampling period **do**
- 5: Calculate the last period's CPU utilization u , $u \in [0, 1]$.
- 6: **if** $u > \text{up_threshold}$ (tunable, $\in [0, 1]$, 0.8 by default) **then**
- 7: $\text{next}_f = \max_f$.
- 8: **else**
- 9: $\text{next}_f = \min_f + (\max_f - \min_f) \times u$.
- 10: $\text{next}_f = (1 - \text{powersave_bias}) \times \text{next}_f$ (tunable, $\in [0, 1]$, 0 by default) $\times \text{next}_f$.
- 11: next_f = the highest frequency below or at next_f supported in F .
- 12: Apply next_f .

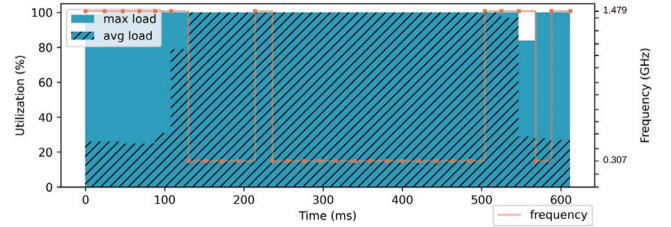


Fig. 4. Use only two V/f supports to fill the fine slack.

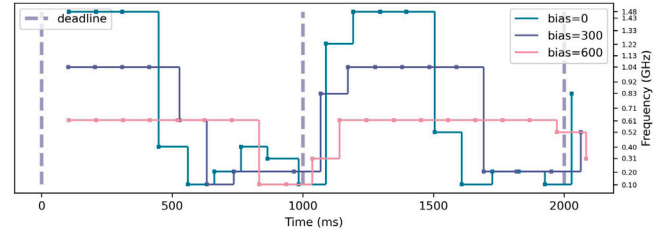


Fig. 5. Ondemand tuning when $\text{powersave_bias} = 0, 0.3, 0.6$.

are time triggered. They use a timer to regularly sample data from the past period and control the frequency of the next period. *Schedutil*, implemented as part of the scheduler, does not rely on a timer but is actively woken up by the scheduler to tune the frequency. The core strategies for *Ondemand* and *Schedutil* are similar. Algorithm 1 gives a description of *Ondemand* in Linux V 5.13. This strategy is effective in its ability to reduce frequency/voltage in low-utilization periods (see 300 ms–1000 ms in Fig. 3 for example), thus saving dynamic and static energy consumption. The conservative governor slowly changes the frequency at a fixed pace (different from line 9 in Algorithm 1), which is less responsive to changes in utilization.

Linux built-in policies are designed for general-purpose systems, and they are low overhead and do not need prior knowledge of the workload. Thus they are the current practice of DVFS. Next, we study the limitation of Linux built-in DVFS for soft-deadline workloads. We achieve this by identifying a few system settings and workload patterns that the built-in governors cannot effectively handle. This motivates the development of a reinforcement learning governor for energy saving that needs little prior knowledge of the tasks and machine.

4.1. Coarse-grained voltage/frequency support

As Linux V 5.13, Linux has one built-in parameter (powersave_bias in *Ondemand*) for tuning DVFS (downscaling CPU performance to fill the

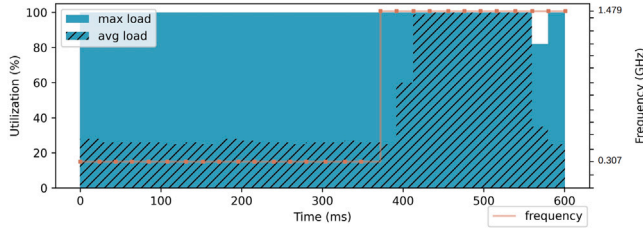


Fig. 6. A strategy with lower average utilization.

slack). The strategy is shown in line 10 of Algorithm 1. Fig. 5 displays the tuning of *powersave_bias* on the workload shown in Fig. 3.

Such a strategy will be less effective for embedded CPUs that do not support fine-grained voltage/frequency support. Although the ARM-A57 CPU in Nvidia Jetson Nano Board 2 GB supports 15 frequency levels, it only supports two voltage levels, and energy-saving requires the CPU frequency to be reduced along with the voltage. We can observe that the downscaling performed by the *powersave_bias* shown in Fig. 5 cannot save energy. Only when many frequencies are dropped to 0.307 GHz (a lower voltage value) the loss of performance begins to have energy-saving benefits. This means a 5x CPU slowdown (drop from 1.479 GHz to 0.307 GHz in high utilization periods), and in many cases, the system would not have such a considerable slack to fill. Conversely, if the system supports the fine-grained V/f option, the user can drop the CPU performance slightly to fill a small slack (e.g., from 1.479 GHz to 1.326 GHz).

As one of the most well-known embedded boards, the ARM-A72 in Raspberry Pi 4B also supports only 2 CPU V/f levels. For Raspberry Pi 4B, users can edit */boot/config.txt* to enable the undervoltage function [35]. In this case, for the lowest frequency (0.6 GHz), the corresponding CPU voltage is reduced, but it also means a 2.5x CPU slowdown (from 1.5 GHz to 0.6 GHz).

In IoT systems where a large number of small devices need to be deployed, people tend to want cheaper devices and, therefore, potentially face challenges of energy savings for systems with coarse-grained V/f support. We want to point out that even Raspberry Pi 4 and the Nvidia Jetson Nano Board 2 GB, the two relatively high-end embedded devices that nowadays cost more than a hundred dollars, support only coarse-grained V/f. Thus, coarse-grained V/f support is a common system setting that we need to consider when designing DVFS governors for embedded systems.

For machines that only support the coarse-grained V/f option, can the DVFS governor be tuned to fill the fine slack? For example, for the workload shown in Fig. 3, for the default setting of *Ondemand* on Nvidia Jetson Nano Board 2 GB, a task takes about 0.35 s to complete. Fig. 4 shows such a possible solution for the 0.6-second deadline setting. However, such a solution cannot be found by tuning the built-in governors. If one wishes to tune an energy-saving policy based on *powersave_bias* for the built-in governor, the deadline for a task cannot be less than 1.5 s.

We observe that for the same high CPU utilization periods, the solution shown in Fig. 4 sets part of the periods to high frequency and part of the periods to low frequency. This is not possible for the built-in Linux methods since they determine the demand for the next period based on the utilization of the past period (the system's instantaneous computational demand). In order to develop the strategy shown in Fig. 4, the governor needs to be able to develop the strategy without being bound to instant characteristics and based on the overall execution of a task.

4.2. Unbalanced load distribution

Since many embedded CPUs nowadays (such as the ARM-A57 in the Nvidia Jetson Nano Board 2 GB and the ARM-A72 in the Raspberry Pi

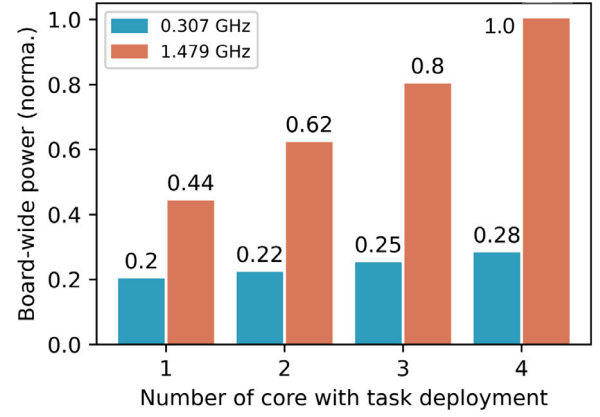


Fig. 7. Power consumption for different CPU utilization on Jetson Nano Board 2 GB.

4B) do not support per-core DVFS, the entire CPU package must run at one single frequency. The Linux built-in method chooses the subsequent frequency based on the highest utilization among all the cores in the previous time period.

For periods with different average utilization, the same frequency downscaling may result in different energy gains with similar performance loss. This is because a task will only be considered finished when all the tasks on all CPU cores have been completed. For example, for the workload shown in Fig. 3, downscaling the period of 0–100 ms would have a similar performance loss as downscaling the period of 100–200 ms because the max CPU utilization among cores in these two periods is both 100%. In this case, executing the instructions in one period with 1.479 GHz can be converted to around five periods with 0.307 GHz. However, downscaling the period of 100–200 ms could have more energy gains because of the higher average utilization (more cores at work), which means more dynamic energy consumption caused by 0/1 flipping can be saved. As shown in Fig. 7, downscaling V/f on higher-average-utilization periods can save more power, which means more energy-saving when the running time is consistent.

Fig. 4 (denoted by *Policy_{high_util}*) and Fig. 6 (denoted by *Policy_{low_util}*) show two DVFS strategies on workload Fig. 3 when the deadline for one task is 0.6 s. *Policy_{high_util}* can save more energy compared to *Policy_{low_util}*.

Both strategies allocate a similar amount of time for the CPU to run at low frequency/voltage (around 57% at 0.307 GHz and around 43% at 1.479 GHz). In this way, they consume a similar amount of static energy (formula (4)). They differ in that *Policy_{high_util}* prioritizes frequency reduction for periods with high average utilization (high avg load), thus resulting in more dynamic energy consumption (formula (3)).

We would like to trade the same performance loss for more energy benefits. When uneven load distribution occurs, the DVFS governor should give preference to periods with high average utilization for downscaling with similar performance loss, which results in higher average CPU utilization. This is not possible for algorithmic architectures similar to Linux's built-in DVFS approach, which performs inference based on short-time system characteristics.

4.3. Internal slack

The slacks discussed in the above examples all appear after the task execution has finished. Due to the presence of IO blocks, slack can also occur during the execution of a task.

Consider a scenario in which the system periodically performs feature analysis on both photos and audio. The process of recording audio is typically an IO-intensive calculation, which can then be used to perform the photo analysis process. The photo analysis process can

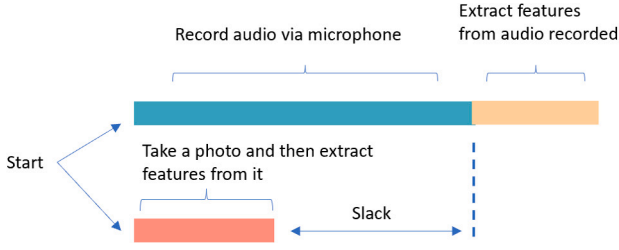
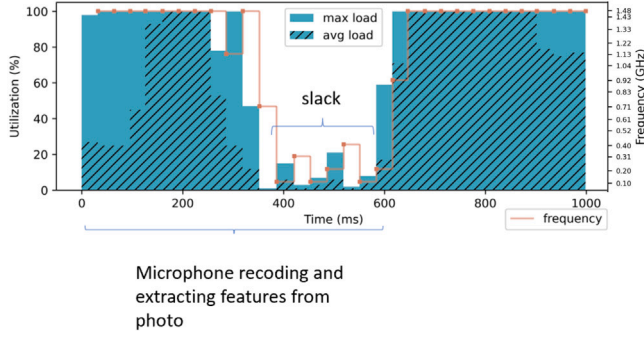
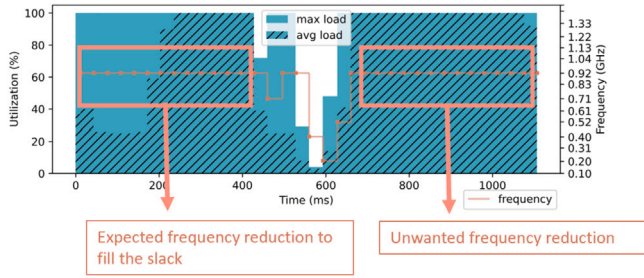


Fig. 8. An internal slack example.

Fig. 9. Profiling of default *Ondemand* on the internal slack example.Fig. 10. Fill the internal slack via *Ondemand's powersave_bias*.

be slowed down to fill the slack caused by recording audio, thus saving energy without compromising overall performance. The photo analysis workload is still the face recognition program we used in previous sections. The microphone recording is set to be 0.6 s. Figs. 8 and 9 show the pipeline of this example.

To handle this situation, the frequency governor should lower the frequency in the early stages and raise it in the later stages.

If we use *Ondemand* to fill this slack, there will be an inevitable performance loss (Fig. 10). This is because, in the built-in governors' perspective, both the front and back sections are CPU-intensive computations, and they should be treated equally.

In order to be able to fill the slack without losing performance, the governor needs to further understand the characteristics of the task. Fig. 11 shows such an example.

In fact, such internal slacks are common for IoT applications. Modern IoT systems [1–5] contains a variety of sensors and network calls. Longer slacks include microphone recording, video recording, etc. Shorter slacks include temperature detection, etc.

4.4. Why extending this DVFS framework cannot cope with the three patterns?

One CPU frequency control flow of the Linux built-in method can be summarized as follows.

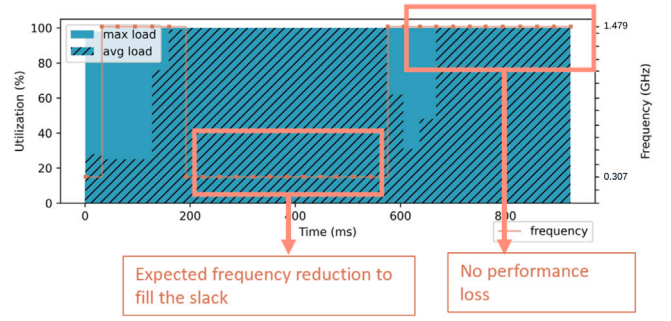


Fig. 11. Fill the internal slack without performance loss.

1. Collect system features for the past period. All three governors consider only CPU utilization in this step. *Ondemand* and *Conservative* use the calculation of CPU runtime divided by total time, and *Schedutil* uses the PeLT metric provided by the scheduler.
2. Predict the events of the next period. All three governors predict that the computational demand for a future period is consistent with that of a past period.
3. Determine the CPU frequency for the next period using predicted events. Taking *Ondemand* as an example, it selects the highest frequency if the utilization is above a threshold. Otherwise, it sets the frequency in equal proportion.

The three patterns we discuss in this section are difficult to handle because the instance CPU utilization cannot capture the system state for making a good frequency decision for a given workload with a soft deadline. As shown in Figs. 4 and 11, instances with the same measured utilization can be assigned different frequencies to reduce energy consumption most effectively.

How can we make a DVFS governor aware of the difference between computing requirements? An intuitive approach is to use more performance counters, making the system characteristics complex enough to distinguish. But there are two problems with doing so.

1. The usefulness of performance counters is specific to the workload. Adding extra performance counters may or may not help. The workload in Fig. 11 will benefit from using the counters provided by the microphone hardware, but that in Fig. 4 will not benefit from using them. Whether the problem can be solved by adding more performance counters is case-by-case, and it is challenging to transfer a solution from one application to another.
2. As the complexity of the inputs increases, it becomes more challenging to develop control strategies. The Linux built-in methods only use a value that logically ranges from 0 to 1 (CPU utilization) and designs some policies based on its explicit meaning. When multiple counters are enabled as input, the meaning of the input is no longer intuitive and even requires some degree of data mining. We will need more powerful models to handle the input, and the Linux kernel's resource constraints prevent it from supporting sophisticated mathematical models.

In summary, the DVFS algorithm that makes frequency scaling decision based on the system features of the past period has difficulty coping with the three patterns we discussed. We need a DVFS governor that can understand the global workload computation demands.

5. Proposed method

In this paper, we design and implement a DVFS governor that adapts to workload requirements to better address the three challenges mentioned above. The proposed method, like the Linux built-in method, only requires the system to provide CPU utilization as input and contains two important components:

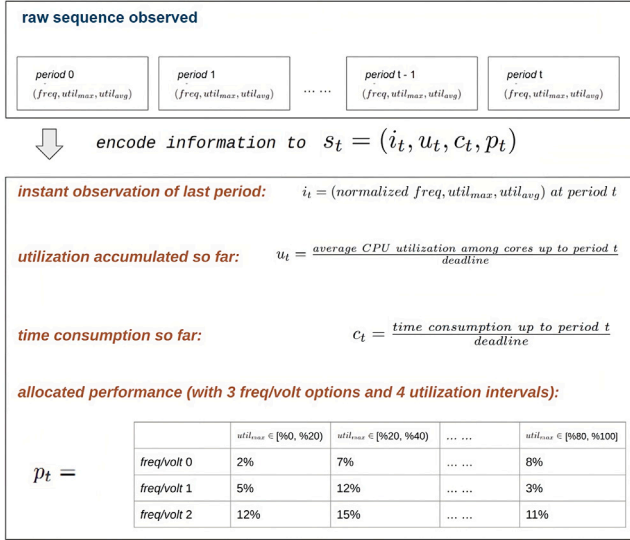


Fig. 12. Encoding the observed time sequence to construct a state.

1. **Temporal encoder:** Construct a state based on the observed sequence of features to better understand the progress of task execution.
2. **Reinforcement learning driven component:** Develop a frequency control strategy based on trial-and-error experience.

In this section we will present our design, and the method of implementation, separately.

5.1. Temporal features of workload and learning

5.1.1. Understanding workload in terms of time

We observe that deadline-constrained periodic workloads have a special feature. That is, the workload has a fixed deadline and periodicity with stable tasks to run, which means a task run can be viewed as an episode. This gives two inspirations.

1. The events that will take place for each execution are similar.
2. The DVFS governor can predict the events that will occur in the future if it understands both the complete events to be experienced and the events that have already occurred.

In addition, the CPU utilization and its corresponding frequency over a period of time reflect the number of 0/1 bits flipped by the CPU. The CPU frequency, utilization, and period experienced by the CPU imply the progress of the task execution. For the DVFS governor, this is a set of observed sequences. We want to mine the workload execution information implicitly contained in this sequence to help the DVFS governor better understand the task's requirements.

5.1.2. Explicit temporal encoding

We first define the time series observed by the DVFS governor. For a periodic soft deadline real-time system, the system executes a task every T seconds. T is also used as the deadline of one task execution. A DVFS governor, such as *Ondemand* or *Conservative*, performs frequency adjustment according to a pre-defined period. For example, if a DVFS governor is set to adjust the CPU frequency ten times per second, it operates with a period of 0.1 s. In practice, a DVFS governor cannot work strictly according to the set period. Some system events, such as hanging at idle moments, can affect the length of a period. Therefore, each period can be of variable length.

Without enabling additional performance counters for each period, the DVFS governor observes the CPU utilization and CPU frequency

within that period. This paper considers time series consisting of these observations. Fig. 12 (top portion) shows the format of a raw observed sequence.

The time series experienced by DVFS governor is indefinitely long. They contain intuitively useful information, but the question is how to understand the time series and develop strategies that produce time series resulting in low energy consumption. A previous work [30] used a Recurrent Neural Network (RNN) for the adaptive processing of time series. However, this leads to lengthy training times, poor interpretability, and results in model architectures that are tuned to task needs. While adaptive extraction of features is more in line with the definition of AI, doing so relies on powerful learning algorithms.

In this work, we consider OS kernel-level code's inherent efficiency and reliability requirements and propose a method to develop a lightweight and interpretable learning and inference scheme. We extract information based on domain knowledge from an observed time series to provide a highly interpretable and low-dimensional encoding scheme. The pipeline is shown in Fig. 12.

A time series at time t is encoded as $s_t = \{i_t, u_t, c_t, p_t\}$, shown in Fig. 12. Next, we explain each component.

i_t denotes the observation of the past period. i_t is to help the governor predict the current position of workload. This information is also used by the built-in DVFS method of Linux.

u_t denotes the average CPU utilization up to sampling point t in the current task period. u_t is to help solve the problem of unbalanced load distribution that occurs in multi-core architectures. In the case of a similar performance impact of frequency tuning, priority is given to downscaling the periods of high average utilization, which saves more energy and shows an increase in the overall average utilization. This information is mainly intended to serve the purpose of exploring DVFS strategies based on reinforcement learning, which we will discuss in detail in the next section.

c_t denotes task progress up to sampling point t within the current task period, ranging in $[0, 1]$. With the CPU performance allocation information, we introduce the time consumption progress $c_t = \frac{\text{time consumption up to } t^{\text{th}} \text{ period}}{\text{deadline}}$. We want the DVFS governor to be able to combine the use of p_t (described below) and c_t to understand the events that have been experienced.

p_t is to help encode an abstract concept of "what events have happened and what events are to occur in the future" and make it concrete into data that the DVFS governor can process. In a period, the CPU's frequency reflects how fast it flips 0/1 bits, and the CPU's utilization and frequency reflect how much workload has been completed. For a sequence, separate statistics on the usage of each CPU frequency in each utilization interval can give a guide to understanding how much workload has been completed. The allocated performance matrix shown in Fig. 12 gives an example, where we count 3 frequency/voltage usages with few utilization intervals.

Algorithm 2 Temporal encoder

- 1: **INPUT:** The observation of t^{th} period: $(\text{freq}, \text{util}_{\text{avg}}, \text{util}_{\text{max}})$, and time consumption x during this period;
- 2: $s_{t-1} = \{i_{t-1}, u_{t-1}, c_{t-1}, p_{t-1}\}$, denoting the state for series from period 0 to $t-1$
- 3: **OUTPUT:** s_t , denoting the state encoded from period 0 to period t .
- 4: $\text{freq}_{\text{normalized}} = \frac{\text{freq} - \text{freq}_{\text{min}}}{\text{freq}_{\text{max}} - \text{freq}_{\text{min}}}$.
- 5: $i_t = (\text{freq}_{\text{normalized}}, \text{util}_{\text{avg}}, \text{util}_{\text{max}})$.
- 6: $u_t = u_{t-1} + \frac{x \times \text{util}_{\text{avg}}}{\text{deadline}}$.
- 7: $c_t = c_{t-1} + \frac{x}{\text{deadline}}$.
- 8: $\text{interval}_{\text{id}_x}$ = the index of the utilization interval util_{max} belongs to.
- 9: $p_t = p_{t-1}$.
- 10: $p_t[\text{freq}][\text{interval}_{\text{id}_x}] = p_t[\text{freq}][\text{interval}_{\text{id}_x}] + \frac{x}{\text{deadline}}$.
- 11: $s_t = (i_t, u_t, c_t, p_t)$.

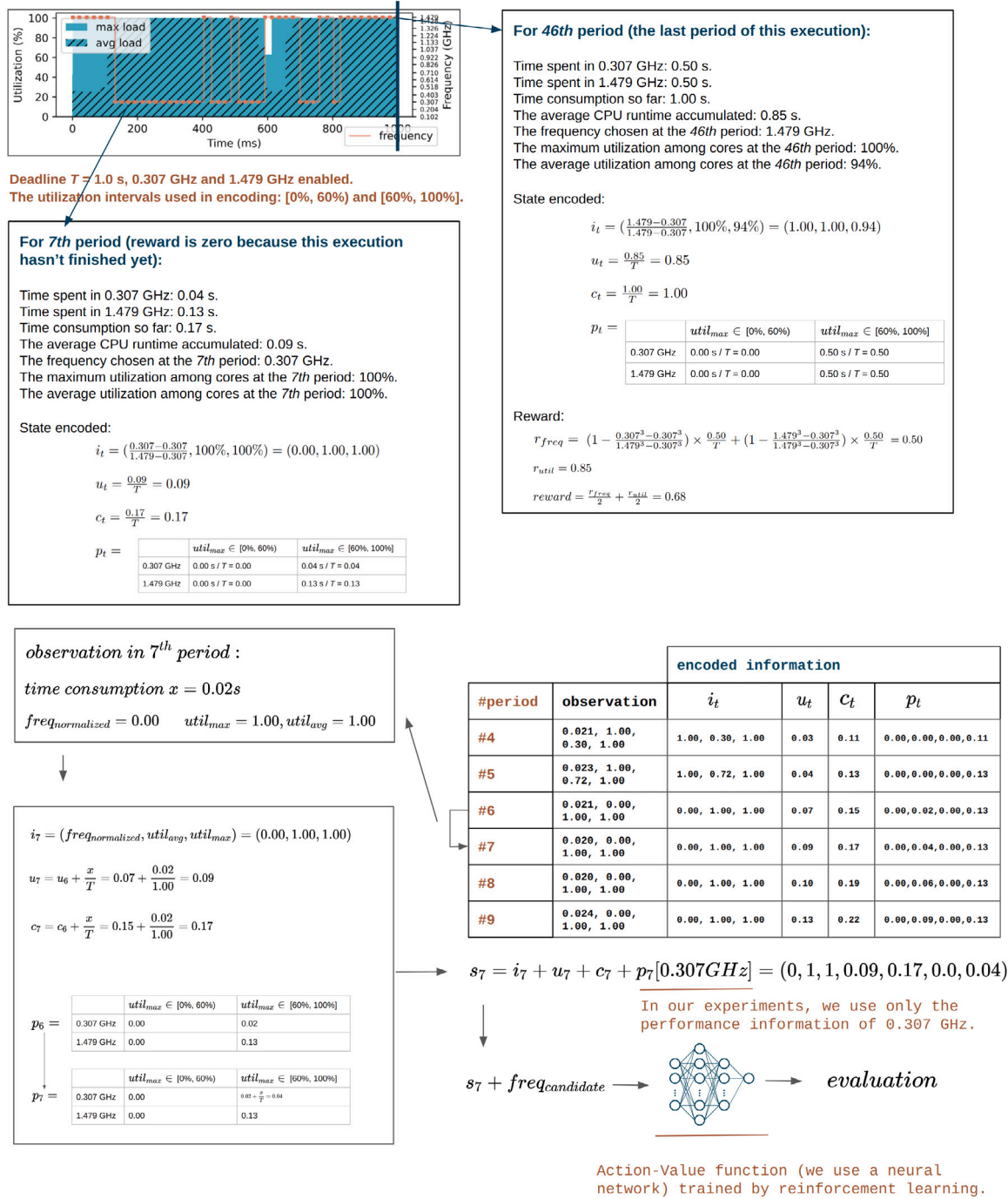


Fig. 13. Examples showing how states and rewards are calculated.

Overall, for a sequence, we encode its information explicitly into four parts: u_t , i_t , c_t , and p_t . The pseudo-code is shown in Algorithm 2. Fig. 13 shows examples of how we encode the observed temporal sequence into a state and how we assign a reward value to it. The purpose of the encoded information is to include the cause for insufficient or excessive task execution performance. We next describe how we use reinforcement learning to construct a DVFS policy based on this information. Ideally, we would like to use reinforcement learning to summarize which execution sequences lead to an encoding state with a high reward value (low power consumption and satisfying performance requirements), and the model can select actions during execution to bring the encoded state closer to a final state with a high reward value.

5.1.3. Reinforcement learning driven policy development

One challenge in considering complex features in the CPU frequency control process is how to map the information to the final decision. The Linux strategy is to use features that are simple and explicitly contain useful information. For example, in *Ondemand*, future frequencies are linearly equated to the observed utilization. When the features under consideration become complex, it becomes more challenging to design a heuristic strategy. We use reinforcement learning to summarize control strategies from experience.

Reinforcement learning is a class of algorithms that observe the reward values harvested from behaviors and then explore strategies that can collect high reward values. In reinforcement learning, one

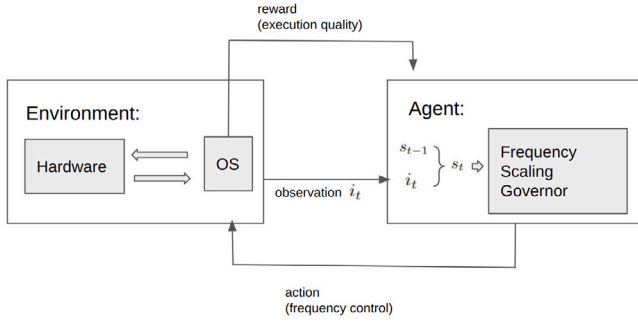


Fig. 14. Frequency scaling as a reinforcement learning scenario.

transition is defined as (s_t, a_t, s_{t+1}, r_t) , where s_t the current state, a_t the action taken, s_{t+1} the resulting state, and r_t the immediate reward assigned to (s_t, a_t) . A reinforcement learning algorithm uses such transitions to update its value-action function, which is used to evaluate the optimality of an action (the CPU frequency, in our case) for a given state. Fig. 14 shows how to model CPU frequency scaling as a reinforcement learning scenario.

Algorithm 3 reward calculation for s_t

```

1: Let  $T$  denote the deadline for one execution.
2: Let  $F$  denote the frequency table provided by hardware.
3: Let  $f_{max}/f_{min}$  denote the max/min frequency supported in  $F$ .
4: if  $s_t$  is not the last state before the deadline reached then
5:    $r_t = 0$ 
6: else
7:   if deadline missed then
8:      $r_t = 0$ 
9:   else
10:     $x = \frac{\text{time spend in } f \text{ during } T}{T}$ 
11:     $r_{freq} = \sum_{f \in F} (1 - \frac{f^3 - f_{min}^3}{f_{max}^3 - f_{min}^3}) \times x$ 
12:     $r_{util} = \text{the average CPU utilization during } T$ 
13:     $r_t = \frac{r_{freq}}{2} + \frac{r_{util}}{2}$ 

```

We want the model to develop an ideal policy by harvesting more rewards for each workload. The principle of designing a reward at a state is to reward a state that leads to low energy consumption, high average utilization and satisfies deadline.

Our reward definition has three components: a reward for low-frequency selection (considering energy), a reward for high CPU utilization, and a penalty for exceeding the deadline (too low performance). The reward value is 0 when the deadline miss occurs. Otherwise, the reward $\in [0, 1]$. All transitions after the deadline miss are discarded. The calculation of the reward is shown in Algorithm 2.

We design the reward to be sparse. Only at the end of the last transition does the model receive the non-zero reward value. The model will only receive a value of 0 as an immediate reward value in all other transitions. We design it this way for two reasons.

- We want the model to predict a value $\in [0, 1]$, which reduces the possibility that the model parameters diverge for predicting large values.
- Without prior knowledge, it is hard to tell if a workload will be time out during its execution. In this case, if we want to assign an immediate reward, we can only confer rewards from the point of view of energy consumption for all the transitions before the deadline miss. We observe that this leads to a rapid tendency of the model to choose low frequencies, thus increasing the learning difficulty.

This reward value can be considered a heuristic energy consumption measure. The ideal way is to use the actual energy consumption, but measuring the energy consumption with high accuracy in a short time requires special hardware support. Therefore in this work, we use this heuristic measure.

For any reinforcement learning problem, there is a need for a model to learn an action-value function. Array-based Q Learning [36] is popular in system design for its easy implementation and low overhead. However, since our state is continuous and large, we need a function approximator to reduce the memory footprint and speed up learning.

Specifically, in our scenario, the temporal encoder encodes the sequence as a vector of length six, and each element belongs to 0 to 1. If we use table-based reinforcement learning (e.g., Q Learning), our first step is to discretize the state consisting of floating-point numbers so that it can be stored in a limited space. This brings up two questions:

1. The state space is likely to remain huge. Suppose we discretize each element of the vector to ten values, then the size of the entire state space is 10^6 , which is unaffordable in kernel space.
2. It is not easy to judge whether a discrete solution is good enough. If the discrete method is too fine-grained (for example, discretizing each vector element to ten values), it will lead to high memory usage and low learning efficiency. If the discrete method is too coarse (for example, discretize each element of the vector to three values), it may result in the information contained in the state being ignored.

It is worth mentioning that for standard table-based Q Learning, each state has its own Q Value, which means that learning based on one set of data cannot be applied to the knowledge of other states. As an example, the information contained in states 96 and 97 may be close, but since their Q values are stored separately, states 96 and 97 are two completely different states in terms of model learning, and thus may lead to a decrease in learning efficiency. Using a function approximator (e.g., a neural network) can improve these problems, as it can:

1. Process floating-point numbers with a pre-defined memory size (the approximator's parameter size).
2. Improve data utilization, since each update involves the parameters of the entire approximator (e.g., one backpropagation of the neural network). In this approach, an update to one state allows similar states to be updated as well.

We use the Double Deep Q-Network (DDQN) model [37], which uses a neural network as the action-value function and a target network during training to reduce the overestimation of actions.

This method will eventually train a neural network that reads the state input and predicts the reward value that the candidate action will receive.

5.2. Implementation

Similar to [30], our work only implements the decision component at the kernel level. The learning component is implemented at the user level with data collected by the in-kernel profiler. Thus, the kernel state is only burdened with little inference overhead.

Our work further reduces the overhead by applying quantization [31], with which the kernel can avoid floating-point calculation. As of today, the Linux kernel does not recommend the use of floating-point calculations. A complete integer-based kernel-state code would increase security (avoid breaking Linux design principles), enhance method pervasiveness (some low-end CPUs do not support floating-point computation), and reduce inference overhead on devices that are poorly optimized for floating-point computation. The training pipeline is shown in Fig. 15. For states that are numerically close has been shown to be detrimental to the training of reinforcement learning models.

We used a simple quantization technique. With the state/reward design (all values are within $[0, 1]$), the parameters of the model and

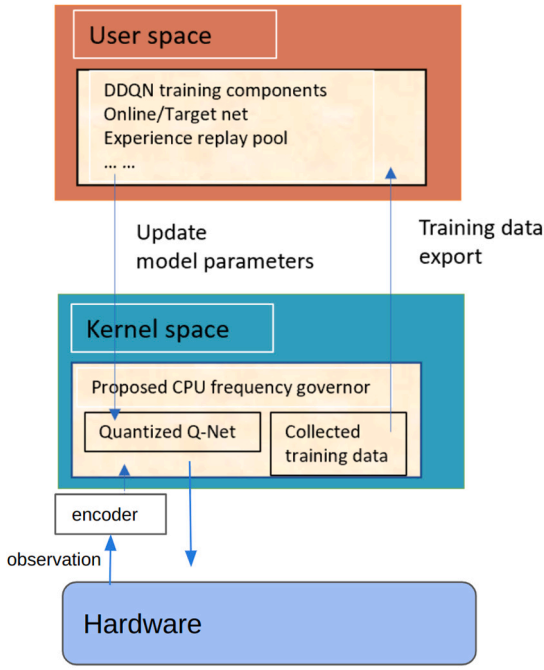


Fig. 15. The user-kernel interleaving training framework with quantization.

the values generated during calculation are within the range of $[-10, 10]$. We quantize all the values into $[-2^{30}, 2^{30}]$, and store them in 32-bit datatype (*int* in C). More advanced quantization techniques can further leverage the range of parameters and the datatype used, thus increasing the precision and reducing the memory footprint on huge-size models [31]. In our case, the model is small (around 150 parameters). We do not choose a more advanced quantization technique considering the expense of a more complicated code architecture needed for further optimization.

In this work, we implemented our proposed governor in the kernel from scratch, including:

1. A Linux C standard Neural Network engine, which can read the NN model resulting from the training by PyTorch in the user state and make inference based on this NN model inside the Linux kernel.
2. An integer-based CPUFreq governor running under the CPUFreq framework that can be compiled as a Linux kernel module using the above engine for inference.
3. A ring-buffer-based event profiler embedded inside CPUFreq governor.
4. A CPU frequency control visualization tool to visualize the information extracted by the above profiler.
5. A set of protocols for the communications between the kernel space and the user space and control of the periodic workloads.
6. A framework that assembles the above-mentioned modules to experiment with the proposed method in this work.

With the interactive training-inference framework Fig. 15, the amount of code added to the kernel internals is reduced, and there is no training overhead at the kernel level. We implement a quantization-enabled neural network engine to read the model parameters generated by user-state PyTorch training and make inferences in the kernel. The engine has a lean amount of code and is easy to compile into the Linux kernel. The codebase can be found at: <https://github.com/coladog/tinyagent>. Please refer to the project documents for more details of how we generate models from PyTorch and make inferences inside the Linux kernel, as well as the implementation of the quantization technique.

Algorithm 4 Kernel-state inference module

- 1: Initialize action-value function Q with parameters θ trained in user space.
- 2: Initialize observed sequence $\phi = []$.
- 3: **for** each sampling period **do**
- 4: Calculate last period's observation: $(freq, util_{avg}, util_{max})$, and the time spent during this sampling period.
- 5: Encode state s according to Algorithm 2.
- 6: $freq_{next} = \max_{action_freq} Q^*(s, action_freq; \theta)$.
- 7: **if** training required **then**
- 8: Generate one random number $v \in [0, 1]$.
- 9: **if** $v > \epsilon$ **then**
- 10: $freq_{next} =$ a random frequency.
- 11: Apply $freq_{next}$.
- 12: Add $(s, freq_{next})$ into ϕ .
- 13: **if** training required **then**
- 14: Output ϕ into user space.

5.3. Training

Algorithm 5 User-state training module

- 1: Load action-value function Q with parameter θ .
- 2: Initialize target function parameter $\theta' = \theta$.
- 3: Load replay memory D .
- 4: Fetch newest training data ϕ output by kernel space, convert each node into a transition (s_t, a_t, r_t, s_{t+1}) , and then add it into D .
- 5: Construct training pool B from D .
- 6: **for** each batch (s_t, a_t, r_t, s_{t+1}) in B **do**
- 7: **if** s_{t+1} is non-terminal **then**
- 8: $y_t = r_t + \gamma Q(s_{t+1}, \max_a Q(s_{t+1}, a; \theta); \theta')$.
- 9: **else**
- 10: $y_t = r_t$.
- 11: Perform a gradient descent step on $(y_t - Q(s_t, a_t; \theta))^2$.
- 12: Every C steps reset $\theta' = \theta$.
- 13: Output quantized θ into kernel space.

The kernel state module observes new data at each step, encodes it into the current system state through the temporal encoder, and then uses Q Net to determine which action (frequency) will result in a larger reward. Meanwhile, in the training phase, it randomly selects actions to explore different strategies according to certain odds and sends the observed sequences to the user state module. The module in the user state collects the sequence data provided by the kernel state, calculates its corresponding reward value, and then uses a reinforcement learning algorithm to train the Q Net and return the updated parameters to the kernel state. Through this interactive step, we finally implant a governor in the kernel state that understands workload requirements and saves energy while satisfying performance.

Algorithm 4 and Algorithm 5 describe the training process using the interleaving framework. In the kernel state, the CPUFreq framework initializes an integer-based neural network (Q Net), reads the model parameters derived from the user state, and infers frequency actions based on this network at runtime. In addition, the kernel state module records the data observed during runtime and exports it to the user state for training at the end of the run. The user-state training module updates the model parameters according to the Double Deep Q-Network (DDQN) training method after loading the training data and model parameters.

In the process of generating the training pool (line 5 in Algorithm 5), we use the idea of prioritized experience replay [38]. The native experience replay pool considers each experience to be of equal priority, while the prioritized experience replay pool [38] considers some

Table 1

Experiment settings.

Hardware	Nvidia Jetson Nano 2GB Board
OS	Linux 4.9
Energy measurement	Board-wide energy consumption measured by a power meter
Optimizer	Adam with 0.001 learning rate
Batch size	16
Target Net updating frequency (C in algorithm 5)	per 32 batch learning
NN structure	7-8-8-1 with ReLU activation function
Utilization intervals used in encoding	{[0%, 60%], (60%, 100%]}
Sampling rate	20000 ms, 50 times per second
Action randomly pick rate during training	0.7 at the first 50 episodes, 0.5 at the next 50 episodes, and then 0.3 till the end

data to be more worthy of learning, and thus purposefully selects some high-priority experience when constructing training data.

We divide the experience pool into 10 buckets and add a sequence of transitions to the corresponding bucket according to the reward received. For example, when we have a sequence with reward = 0.27, then the sequence will be added to the 3rd bucket, which contains the sequences with reward $\in [0.2, 0.3)$. For each training pool construction, we will randomly take out 64 sets of sequences from each bucket and construct a training dataset using the transitions contained. With this approach, the model can learn a variety of experiences with different reward levels in one training step.

6. Experimentation

Learning an embedded control algorithm (a frequency control governor) for operating system kernels has not been widely explored by the industry or open-source community. Therefore there is a lack of open-source support. Our work has to build tools from scratch for efficient profiling, in-kernel inference, and frequency control policy visualization.

6.1. Experimental setup

We first verify whether our approach can better address the patterns we discussed in Section 4. The experiment settings can be found in Table 1. Two complex self-designed workloads are used for this purpose.

1. *FaceRecog*: The system periodically reads a photo and then identifies the faces' location. Image reading and pre-processing are done in a single thread, and face recognition is done in multiple threads. Figs. 3 and 5 show the frequency tuning of this workload under *Ondemand*. The image processing is implemented based on OpenCV [39].
2. *AudioRecog*: The system periodically performs an audio recording while running the *FaceRecog* workload and performs feature extraction on an audio clip after the recording is finished. The audio analysis is implemented based on PyAudioAnalysis [40]. Figs. 8 and 9 show the pipeline and the frequency tuning of this workload under *Ondemand*.

We also validate our approach on publicly available workloads with hidden implementation details. In this step, we use four workloads provided by MiBench [41]: *bitcount*, *susan*, *dijkstra* and *typeset*.

On *FaceRecog*, we train three sets of models corresponding to 0.6 s, 0.9 s, and 1.2 s deadlines, respectively. On *AudioRecog*, we train three sets of models corresponding to 1.0 s, 1.3 s, and 1.6 s deadlines, and with 0.6 s, 0.9 s, and 1.2 s microphone recording time, respectively.

We compare our approach with all DVFS governors (*Performance*, *Ondemand*, *Schedutil*, *Conservative*) currently supported by Linux, except

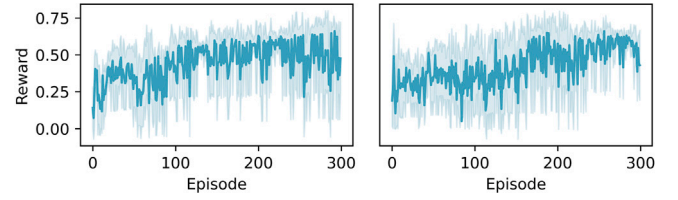


Fig. 16. Reward curve with five training on *FaceRecog* (left) and *AudioRecog* (right).

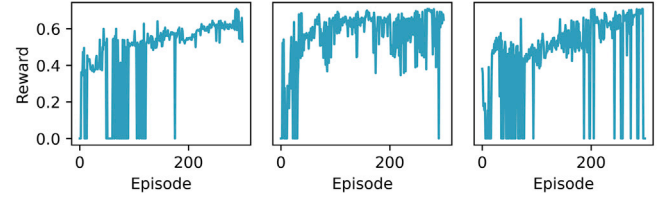


Fig. 17. Three reward curves with single training on *FaceRecog*.

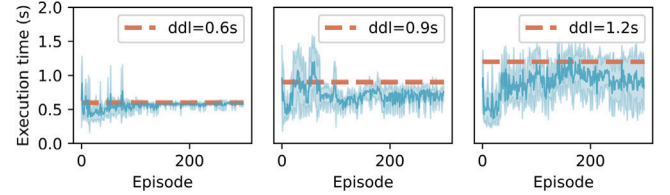


Fig. 18. Execution time curve with five training on *FaceRecog*.

for *Powersave*, which just sets the frequency to the lowest level and thus can only be used in the extreme case.

Our experimental environment is the Nvidia Jetson Nano Board. Our method uses two frequency options (1.479 GHz and 0.307 GHz) with different voltages. However, more frequency levels must be enabled for Linux built-in methods, even with the same voltage support. For *Ondemand*, after calculating the logical frequency based on $\min_f + (\max_f - \min_f) \times \text{utilization}$, it will lookup for a frequency supported by hardware that is below or at the logical frequency. In this case, if it only has two frequency support, unless the utilization is 0, it will always choose the higher one. *Conservative* and *Schedutil* also require fine-grained frequency support for similar reasons. Therefore, we enable full frequency support for the Linux built-in methods, even though the hardware supports only two voltage levels.

6.2. Reward curve

After each training, we executed the workload five times using the latest model and then counted its average harvested reward value. Fig. 17 shows three separate training on *FaceRecog* with 0.6 s deadline. Since training for reinforcement learning is subject to randomness (random selection of actions to explore, random learning of data), a common measure of learning quality is to take the reward curve of multiple learning sessions and count their mean and standard deviation. Fig. 16 shows the result on *FaceRecog* and *AudioRecog*, with 0.6 s deadline and 1.0 s deadline separately. Our method demonstrated the ability to harvest more reward value in training.

6.3. Deadline awareness

We also evaluate the task execution time under the policy after each training step five times and take the average. Figs. 18 and 19 show the mean and standard deviation of the execution time curves based on five training sessions. Our method perceived the need for deadlines very well. **It is worth mentioning that for each deadline,**

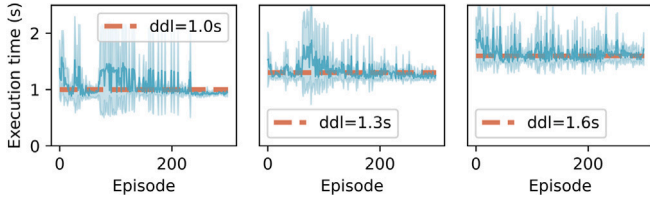


Fig. 19. Execution time curve with five training on AudioRecog.

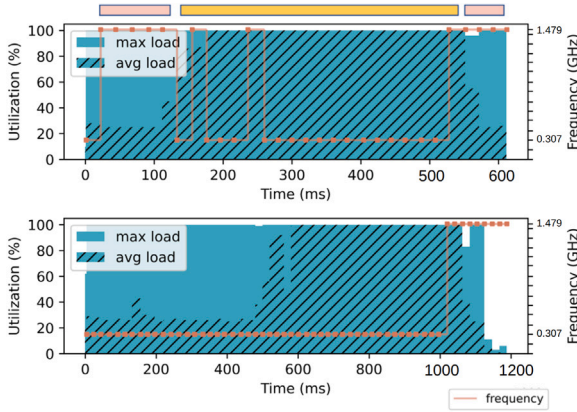


Fig. 20. Frequency policy developed by proposed method on FaceRecog with 0.6 s (top) and 1.2 s (bottom) deadline respectively.

our method receives only one numerical value (for example, 0.6 for 0.6 s deadline), but can generate a DVFS strategy accordingly that respects the deadline. In our experimental setting, the governor uses only two frequency values. It cannot drop the overall frequency a little to accommodate the performance change (e.g., from 1.479 GHz to 1.428 GHz). Adaptation to deadline requires it to combine the only two frequency options available. It takes it upon itself to relate this abstract value to performance requirements and make policy adjustments.

6.4. Learned policy

An important contribution of this work is visualizing the CPU frequency control process within the kernel for any workload with any given strategy. The visualization helps us understand and compare the differences between the policies and the learning process.

Through visualization, we observed that the proposed approach smartly learned very different strategies with the same workload when the deadline changed. Fig. 20 shows an example. For different deadlines, although the maximum utilization among cores was close to 100% throughout the execution, low frequencies were set to save energy without exceeding the deadline.

The visualization showing the frequencies, max utilization, and average utilization together at each observation point allows us to observe when the choices of low frequency or high frequency occur. For Figs. 20 and 21, we use the yellow bar to indicate the range with mostly low frequency and the pink bar to indicate the range with high frequency. For the 0.6 s deadline, we observe that the model preferred to choose the low frequency in periods with high average utilization (yellow bars). In this case, the strategy chooses the high frequency for periods with low average utilization (pink bars). For the 1.2 s deadline, a large number of low frequencies were adopted due to the reduced performance requirements. For this workload, our proposed method showed the ability to use coarse-grained frequency support to fill the slack as well as to optimize overall CPU utilization.

The AudioRecog workload is designed to test if our approach is able to learn additional hidden abstract information: the existence of an

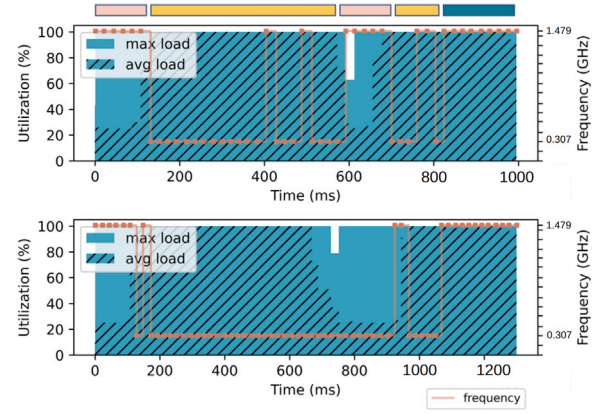


Fig. 21. Frequency policy developed by proposed method on AudioRecog with 1.0 s (top) and 1.3 s (bottom) deadline, and 0.6 s (top) and 0.9 s (bottom) microphone recording, respectively.

internal slack. At the highest speed (1.479 GHz), with 0.6 s microphone recording, it takes around 0.88 s to finish one request. We assigned a 1.0 s deadline, which means there is about 0.12 s slack after the request is finished at the highest speed. Also, there are around 0.28 s internal slack before the microphone recording is finished. In the developed policy (Fig. 21 top), the proposed method exhibits the ability to preferentially select high average utilization periods to reduce the frequency (yellow bars). At the same time, it drops the frequency heavily during the internal slack (yellow bar) and boosts it at the end (green bar), although both intervals exhibit an average utilization close to 100%. For 1.3 s deadline with 0.9 s microphone recording, the proposed method adjusts its policy (Fig. 21 bottom) with the change of performance settings.

6.5. Energy saving

As shown in Fig. 22, 23 and 24, our approach can self-develop DVFS policies to accommodate all chosen workloads (including self-designed workloads and workloads from MiBench) with different deadlines. The longer the deadline, the more energy-efficient our approach is compared to the built-in Linux approach.

In these figures, each row represents the results for one particular workload. Each column corresponds to one specific deadline. Each bar has the format of *governor (running time)* to represent the average time to complete the workload under the given governor for the particular workload (indicated by row) and the particular deadline (indicated by column). To save chart space, we use the first three initials of the governor's name instead of its full name, e.g., *Pro.* means *our Proposed method*. The length of the bar indicates the normalized energy consumption. Each case's normalized energy consumption is shown at the top of the bars. We can observe that the proposed method consumes the least energy in almost all cases. Take the FaceRecog example, the energy consumption is 88%, 80%, and 72% compared to the Performance governor for deadline = 0.6s, 0.9 s, and 1.2 s, respectively. For energy saving for other workloads and deadlines, please refer to Figs. 23 and 24.

On Mibench workloads, with performance slack ranging from 0.04 s to 0.4 s, the proposed method can save 3%–11% more energy compared to Ondemand. On FaceRecog, with performance slack ranging from 0.27 s to 0.87 s, the proposed method can save 5%–14% more energy compared to Ondemand. On AudioRecog, with performance slack ranging from 0.11 s to 0.12 s, the proposed method can save 12%–14% more energy compared to Ondemand. Note that for AudioRecog, the energy-saving opportunity mostly comes from the internal slack.

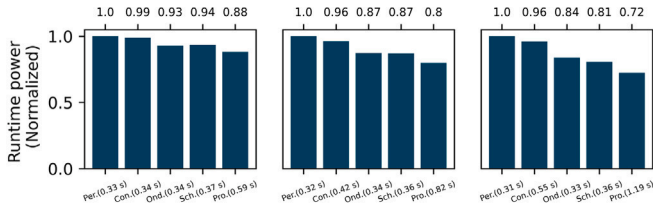


Fig. 22. Power consumption on *FaceRecog* with $ddl = 0.6$ s, 0.9 s, 1.2 s, from left to right respectively.

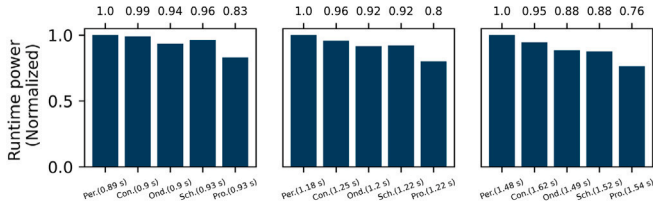


Fig. 23. Power consumption on *AudioRecog* with $ddl = 1.0$ s, 1.3 s, 1.6 s, from left to right respectively.

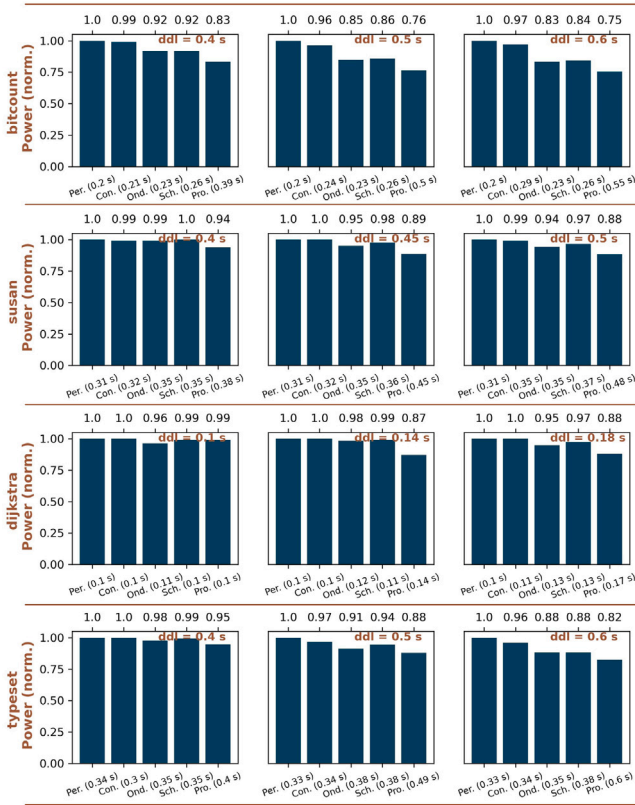


Fig. 24. Power consumption on *MiBench* workloads at various deadlines.

6.6. Inference time overhead

The time overhead of the proposed method consists of constructing the system state and then reasoning about the state using the RL model. We measure the time overhead by timing the execution of this block of code in the kernel. The associated time overhead is influenced by two factors: The frequency at which the CPU is running when performing inference and the number of tasks that the relevant cores are processing concurrently.

benchmark	deadline (s)	mean execution time (s)	standard deviation	exceed deadline 0-2.5%	exceed deadline 2.5-5%	exceed deadline 5%	not exceed deadline
<i>FaceRecog</i>	0.6	0.59	0.012	11.20%	4.70%	0.20%	83.90%
	0.9	0.87	0.015	0.40%	0.00%	0.00%	99.60%
	1.2	1.13	0.008	0.00%	0.00%	0.00%	100.00%
<i>AudioRecog</i>	1.0	0.97	0.032	14.00%	3.00%	0.10%	82.90%
	1.3	1.22	0.048	4.50%	2.30%	1.70%	91.50%
	1.6	1.55	0.031	6.80%	1.80%	0.00%	91.40%

Fig. 25. Deadline missing test.

For a fair measure, on Nvidia Jetson Nano 2 GB Board, we run a task with 100% CPU utilization on the core on which the DVFS governor is running and then measure the time overhead of running the proposed method at 1.479 GHz and 0.307 GHz, respectively. For 1.479 GHz, we collected 1033 sets of data, and the average time overhead is 25.62 us. For 0.307 GHz, we collected 1583 sets of data, and the average time overhead is 41.05 us. We believe this is low enough overhead.

6.7. Deadline missing

For the two self-constructed workloads with three separate deadlines, we execute them 1000 times using a policy generated by reinforcement learning, and count the number of timeouts. The result is shown in Fig. 25. First, the time taken to execute workload per policy is very stable, as reflected by the low standard deviation (0.08–0.048). When the average time taken to execute a workload is very close to a given deadline, it will miss the deadline in about 20% of the cases. For example, for *FaceRecog*, at 0.6 deadline, the generated policy took 0.59 s to execute the workload once on average, and 16.1% of the runs timed out. We also note that the deadline timeout is only concentrated within 5%, which is 0.03 s for a 0.6 s deadline. Such a small number of timeouts can be considered as fluctuations caused by system events, and we believe that such results are good enough for soft real-time requirements.

7. Compare with using RNN for end-to-end learning

Our previous work [30] used Recurrent Neural Network (RNN) to provide end-to-end learning. This section discusses the difference between this work and the RNN-based model and explains why we want to use this encoding-based system. For end-to-end learning, an RNN is used to process the observed workload sequences and generate encoded inputs into the Q Net. No human knowledge is involved in this approach. In our previous work, for some simple workloads, on a hardware environment that supports ten fine-grained frequency support, the RNN approach can find a policy that sets the frequency as low as possible for the overall execution of the task without timeouts.

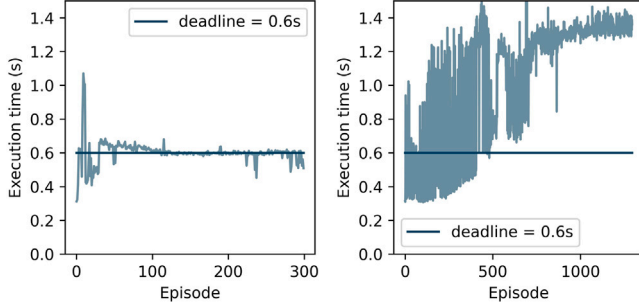
In this work, our target platform is embedded devices oriented to workloads containing three challenging features (coarse-grained f/V, unbalanced load distribution among cores, and workload having internal slack). Our testbed is the Jetson Nano Board.

We conduct a set of experiments here to compare the differences between the two schemes in this embedded device. Our encoding method encodes a feature of length 6 from the workload sequence. In the scheme using RNN, we use a GRU (Gated Recurrent Unit) with input size 3 and hidden vector size 6, to automatically encode the workload sequence into a feature of length 6. Note that the feature lengths extracted by these two methods were set to be the same. The extracted features and candidate frequency action are processed, then

Table 2

Workloads used.

FaceRecog	Self-constructed, including unbalanced load
AudioRecog	Self-constructed, including unbalanced load and internal slack
Mibench	Four workloads are used: <i>bitcount</i> , <i>susan</i> , <i>dijkstra</i> and <i>typeset</i>

**Fig. 26.** Training encoding-based learning (left, 300 episodes) and RNN-based learning (right, 1200 episodes) with large training pool size.

input to a fully connected neural network (Q Net) of size 7-8-8-1. An evaluation of that frequency action results from the Q Net processing. The training parameters and methods are the same for both schemes, as shown in Table 2. The workload we use is *FaceRecog* with a 0.6 s deadline.

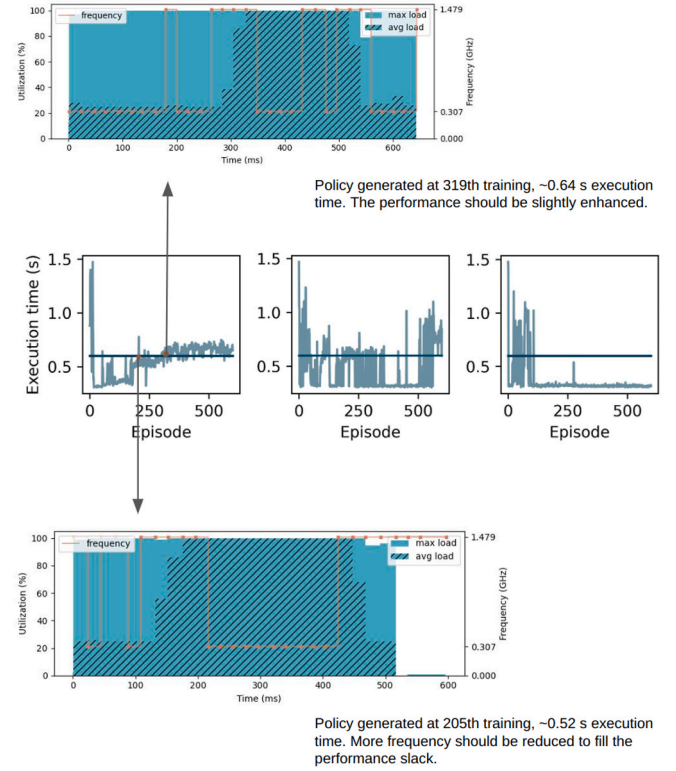
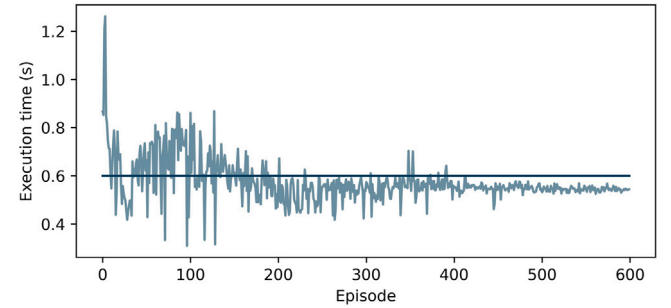
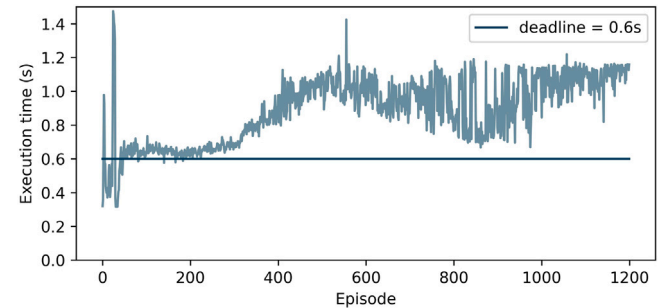
We first train the RNN-based learning the same way as the encoding-based method. For each new episode of experience collected, 64 sets of experiences are selected from each level of the experience pool and divided by reward level for training. Fig. 26 shows the change in execution time of the workload during learning. The encoding-based scheme demonstrates the ability to quickly sense the deadline (good strategies were developed with only 100 episodes) and maintain an understanding of requirements in subsequent training. In contrast, despite learning 1200 episodes, the RNN-based approach shows no signs of approaching the deadline.

We further tune the training of the RNN scheme by changing the amount of the data used for training. This time, 10 sets of experiences are selected from each experience pool bucket with level ≥ 5 , and 3 sets of experiences are selected from each bucket with level < 5 .

This time, for training the RNN-based method, we noticed much better results. Fig. 27 shows three learning curves. One observation is that the RNN-based training method is **unstable**. Sometimes it shows a good perception of the deadline requirements (the figure on the left in Fig. 27), and sometimes a poorer perception (the figure on the right in Fig. 27). In contrast, the learning curves generated by the proposed encoding-based method are consistently similar to Fig. 28.

The RNN-based scheme can sometimes extract some strategies close to the ideal answer (the two policies on the left are visualized in Fig. 27). However, these models eventually could not meet the deadline requirement and kept exporting timeout policies. Extended training time would not help either (as shown in Fig. 29). We also trained the RNN-based method on *AudioRecog*, and it shows similar patterns (Fig. 30). For the challenging scenarios discussed in this paper, the RNN-based method demonstrates the ability to summarize knowledge to some extent, but not optimized. An example is shown in the middle figure in Fig. 30, where the model continuously generates policies that always set the highest frequency (1.479 GHz) so as not to trigger a timeout but does not take advantage of the energy saving opportunity provided by the internal slack.

A problem that should not be overlooked is that during training, the RNN-based approach needs to process the complete observed sequence to obtain a feature. In contrast, the encoding-based approach can

**Fig. 27.** Training RNN-based method on *FaceRecog* with 0.6 s deadline after tuning.**Fig. 28.** Training encoding-based method with smaller training pool size.**Fig. 29.** Training RNN-based method with 1200 episodes.

directly read the saved encoded features. This is because for RNN, each training changes its parameters and thus the extracted features, so the training requires reprocessing the sequence to get the new features. In contrast, the features generated by encoding schemes based on human knowledge are fixed and do not need to be recomputed. For

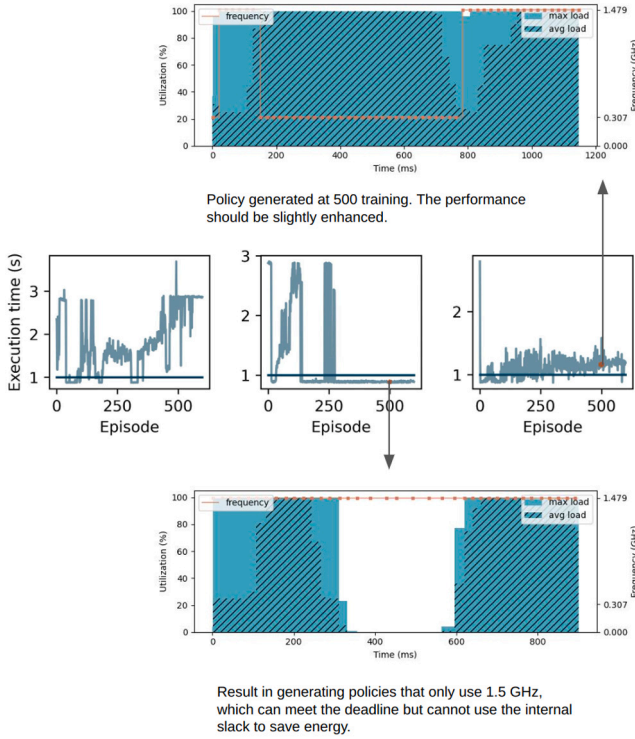


Fig. 30. Training RNN-based learning on *AudioRecog* with 0.6 s internal slack and 1.0 s deadline.

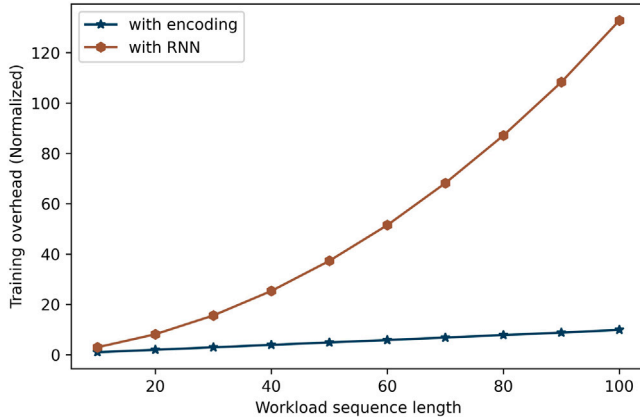


Fig. 31. Training overhead increases as the length of workload increases.

a workload sequence of length N , one complete learning would require RNN processing $1 + 2 + 3 + \dots + N = \frac{N \times (N+1)}{2}$ times data. This overhead grows as the length of the workload grows. For the rule-based encoding scheme, this computational overhead does not exist.

For the network structure used in this experiment, the comparison of training time is shown in Fig. 31. When the workload length is 10, the training time of the RNN-based scheme is about 2.9 times higher than that is based on encoding. When the workload length changes from 10 to 100, the RNN-based training overhead increases by a factor of about 45 and the encoding-based overhead only increases by a factor of about 9.8. The training time with the expert-knowledge-based encoding scheme is much smaller and grows much more slowly.

The first reason we want to explore the encoding-based method over RNN-based is the low interpretability of neural network models. Up to now, the interpretability of neural networks is still poor. There is no obvious better way to improve a model except by

adjusting the structure and making the experimental comparison. In addition, it is difficult to analyze the meaning of the data contained in the features encoded by the neural network from the human perspective. Therefore it is difficult to intervene in its learning process. **It is possible to achieve a better result by adjusting the RNN model structure and training method.** However, even if an ideal RNN structure is tuned for a workload, we cannot be sure that it can handle more complex features. We successfully processed simple workloads with RNN in a hardware environment that supports fine-grained frequency options in the above experiments. The same approach does not work well when dealing with the more challenging patterns discussed in this work. When the workload becomes even more complex, such as multi-tasking with multiple deadlines, the demands on the learning model will be higher. That is unless an extremely powerful model is fixed (which also implies a significant inference overhead), the user may need to adapt the model structure based on the workload, which is against the original purpose of wanting it to be adaptive.

Our encoding scheme can be seen as extracting valuable information from the raw data based on expert knowledge and then handing it over to a machine-learning model to map to the final control. In this approach, the main information extraction task is performed by human experts, thus reducing the reliance on machine learning structures. At the same time, such an approach provides a degree of interpretability: we can control and analyze the information perceived by the model by adjusting the encoded information. Our experiments demonstrate that the reinforcement learning model can generate good frequency control policies quickly and consistently based on the encoding scheme we designed.

Another reason is the training overhead. As we discussed, the overhead of training an RNN-based model is much larger than that of training an encoding-based model. Our aim is to provide fast and lightweight learning on small devices. Excessive computational complexity prolongs learning time and is also a challenge for the device's heat dissipation capability.

8. Related work

Jung et al. [18] used a Bayesian classifier, which is trained offline, to predict the performance and power dissipation of the processor for incoming tasks. The features considered include task priority, queue occupancy, and arrival rate of the task. A policy table calculated offline by dynamic programming was used to map the predicted state to V/f action. Conradhoffmann et al. [30] used the Performance Monitoring Unit (PMU)s provided by the Cortex-A53 processor to offline analyze the correlation between performance counters (Bus Access for Memory Write, Read Alloc Mode, CPU Cycle, etc.) and target task's execution time. An ANN model, which can learn online, was used to take in the selected features and predict task utilization. A set of heuristic-based rules were designed to use the ANN prediction results to adjust the frequency while balancing the load. Park et al. [20] focused on developing highly interpretable solutions. They analyzed the tradeoff between precision and interpretability of various ML models on a dataset of mobile gaming workloads. Tree-based linear models were finally selected and implemented to improve CPU/GPU utilization while achieving the target Frames-per-Second (FPS). Das et al. [21] used a statistical method to detect the application change point, along with an RL-based run-time manager and a hierarchical approach for V/f and thermal management.

Ul et al. [25] used Q learning, a popular RL algorithm, to switch existing DVFS methods dynamically. Based on the previous work, Ramegowda et al. [42] implemented and validated the hybrid DVFS method in various embedded devices running the Linux system. Wang et al. [27] used Double Q learning to explore the energy-performance optimization for both CPU core and uncore parts. Specifically, they used the instruction per cycle (IPC), and the misses per operation (MPO) [43] as the state measurement of the environment and used $\frac{IPC^3}{W}$ as the

reward to describe the tradeoff between energy and performance. Although it was for high-performance computing, the goal was close to an embedded computing scenario: to be as energy-efficient as possible while meeting a global deadline. Shafik et al. [28] proposed a learning transfer-based adaption method, so the Q learning model, which only uses the CPU cycles in the last period as the system state, will not have to learn from scratch again when workload changes, thus reducing the convergence time.

In Section 7, we compared the proposed temporal encoder-based approach with the RNN-based approach previously proposed [30]. However, we have not been able to find other similar studies that can be fairly compared the results. We next explain why based on three recent studies [19,27,44]. The works done by Gupta et al. [44] and Hoffmann et al. [19] can be seen as extending the architecture of Linux built-in methods by using more counters (Linux built-in methods use only utilization) to predict more events (Linux built-in methods assume the future period's utilization is the same as the past one), and designing corresponding rules to map the data to the frequency selection. The biggest challenge of comparing to Gupta's approach is that they did the experiments in a architecture simulator, and there is often a huge gap between the real system kernel environment and a simulator. [19] has done experiments on real systems and hardware. Unfortunately, there is no publicly available code and tuned parameters. And thus, it is difficult to reproduce the corresponding engineering implementation based on the paper description alone to have a fair comparison. Wang et al. [27] implemented their method in userspace and used user-state tools to collect information for states and rewards. Due to the overhead of user-level data collection, the work sets their frequency sampling rate at the sec-level, which would not work in our cases as the execution time of the workload we consider on embedded devices is short. None of the above works consider temporal encoding to optimize frequency scaling by learning the task execution sequences.

9. Conclusion

This work focus on energy saving for periodic systems constrained by the deadline on small devices. We identify three system patterns that may make Linux's built-in and similarly structured DVFS algorithms less effective. We presented a reinforcement learning-driven DVFS governor using explicit temporal coding as input and experimented with it on an Nvidia Jetson Nano Board. Our solution does not require an a priori model of the workload and devices architecture, which makes it practical and simple to deploy. This is similar to the long-established and well-tested Linux built-in systems. Our reinforcement learning method can derive a governor without introducing additional performance counters but can distinguish states with the same instance utilization through rapid profilings and learning with temporal encoding. The governor derived better addresses the three system patterns we identified and quickly adapts to six different applications and various performance requirements settings.

Compared to the built-in Linux approach, the derived governor is able to leverage performance slack, save more energy, and place only a very small inference overhead burden.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

- [1] M. Taneja, N. Jalodia, P. Malone, J. Byabazaire, A. Davy, C. Olariu, Connected cows: Utilizing fog and cloud analytics toward data-driven decisions for smart dairy farming, *IEEE Internet Things Mag.* 2 (4) (2019) 32–37.
- [2] D. Brunelli, A. Albanese, D. d'Acunto, M. Nardello, Energy neutral machine learning based iot device for pest detection in precision agriculture, *IEEE Internet Things Mag.* 2 (4) (2019) 10–13.
- [3] P.M. Chintanpalli, S. Yenuganti, M. Guizani, IoMT and DNN-enabled drone-assisted COVID-19 screening and detection framework for rural areas.
- [4] R. Togneri, C. Kamienski, R. Dantas, R. Prati, A. Toscano, J.-P. Soininen, T.S. Cinotti, Advancing IoT-based smart irrigation, *IEEE Internet Things Mag.* 2 (4) (2019) 20–25.
- [5] Y. Sahraoui, C.A. Kerrache, A. Korichi, B. Nour, A. Adnane, R. Hussain, DeepDist: A deep-learning-based IoV framework for real-time objects and distance violation detection, *IEEE Internet Things Mag.* 3 (3) (2020) 30–34.
- [6] F. Reghenzani, A. Bhuiyan, W. Fornaciari, Z. Guo, A multi-level DPM approach for real-time DAG tasks in heterogeneous processors, in: 2021 IEEE Real-Time Systems Symposium, RTSS, IEEE, 2021, pp. 14–26.
- [7] B. Ranjbar, T.D. Nguyen, A. Ejlaei, A. Kumar, Power-aware runtime scheduler for mixed-criticality systems on multicore platform, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 40 (10) (2020) 2009–2023.
- [8] D. Zhu, R. Melhem, B.R. Childers, Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems, *IEEE Trans. Parallel Distrib. Syst.* 14 (7) (2003) 686–700.
- [9] A. Bhuiyan, Z. Guo, A. Saifullah, N. Guan, H. Xiong, Energy-efficient real-time scheduling of DAG tasks, *ACM Trans. Embed. Comput. Syst. (TECS)* 17 (5) (2018) 1–25.
- [10] A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, N. Guan, Z. Guo, Energy-efficient parallel real-time scheduling on clustered multi-core, *IEEE Trans. Parallel Distrib. Syst.* 31 (9) (2020) 2097–2111.
- [11] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, N. Guan, Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, IEEE, 2019, pp. 156–168.
- [12] A. Saifullah, S. Fahmida, V.P. Modekurthy, N. Fisher, Z. Guo, CPU energy-aware parallel real-time scheduling, in: *Leibniz International Proceedings in Informatics*, Vol. 165, 2020.
- [13] A. Paolillo, J. Goossens, P.M. Hettiarachchi, N. Fisher, Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies, in: 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE, 2014, pp. 1–10.
- [14] J. Huang, H. Sun, F. Yang, S. Gao, R. Li, Energy optimization for deadline-constrained parallel applications on multi-ECU embedded systems, *J. Syst. Archit.* 132 (2022) 102739.
- [15] Z. Li, S. Ren, G. Quan, Energy minimization for reliability-guaranteed real-time applications using DVFS and checkpointing techniques, *J. Syst. Archit.* 61 (2) (2015) 71–81.
- [16] M. Qiu, Z. Ming, L. Jiayin, S. Liu, B. Wang, Z. Lu, Three-phase time-aware energy minimization with DVFS and unrolling for chip multiprocessors, *J. Syst. Archit.* 58 (10) (2012) 439–445.
- [17] H. Sobhani, S. Safari, J. Saber-Latibari, S. Hessabi, REALISM: Reliability-aware energy management in multi-level mixed-criticality systems with service level degradation, *J. Syst. Archit.* 117 (2021) 102090.
- [18] H. Jung, M. Pedram, Supervised learning based power management for multicore processors, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 29 (9) (2010) 1395–1408.
- [19] J.L.C. Hoffmann, A.A. Fröhlich, Online machine learning for energy-aware multicore real-time embedded systems, *IEEE Trans. Comput.* 71 (2) (2022) 493–505, <http://dx.doi.org/10.1109/TC.2021.3056070>.
- [20] J.-G. Park, N. Dutt, S.-S. Lim, An interpretable machine learning model enhanced integrated CPU-GPU DVFS governor, *ACM Trans. Embed. Comput. Syst. (TECS)* 20 (6) (2021) 1–28.
- [21] A. Das, G.V. Merrett, M. Tribastone, B.M. Al-Hashimi, Workload change point detection for runtime thermal management of embedded systems, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 35 (8) (2015) 1358–1371.
- [22] Y. Tan, W. Liu, Q. Qiu, Adaptive power management using reinforcement learning, in: 2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers, IEEE, 2009, pp. 461–467.
- [23] W. Liu, Y. Tan, Q. Qiu, Enhanced Q-learning algorithm for dynamic power management with performance constraint, in: 2010 Design, Automation & Test in Europe Conference & Exhibition, DATE 2010, IEEE, 2010, pp. 602–605.

- [24] Y. Wang, M. Pedram, Model-free reinforcement learning and bayesian classification in system-level power management, *IEEE Trans. Comput.* 65 (12) (2016) 3713–3726.
- [25] F.M.M. ul Islam, M. Lin, Hybrid DVFS scheduling for real-time systems based on reinforcement learning, *IEEE Syst. J.* 11 (2) (2015) 931–940.
- [26] D. Ramegowda, M. Lin, Energy efficient mixed task handling on real-time embedded systems using freertos, *J. Syst. Archit.* 131 (2022) 102708.
- [27] Y. Wang, W. Zhang, M. Hao, Z. Wang, Online power management for multi-cores: A reinforcement learning based approach, *IEEE Trans. Parallel Distrib. Syst.* 33 (4) (2021) 751–764.
- [28] R.A. Shafik, S. Yang, A. Das, L.A. Maeda-Nunez, G.V. Merrett, B.M. Al-Hashimi, Learning transfer-based adaptive energy minimization in embedded systems, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 35 (6) (2015) 877–890.
- [29] S.K. Panda, M. Lin, T. Zhou, Energy efficient computation offloading with DVFS using deep reinforcement learning for time-critical IoT applications in edge computing, *IEEE Internet Things J.* (2022).
- [30] T. Zhou, M. Lin, Deadline-aware deep-recurrent-q-network governor for smart energy saving, *IEEE Trans. Netw. Sci. Eng.* (2021).
- [31] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.
- [32] S. Kaxiras, M. Martonosi, Computer architecture techniques for power-efficiency, *Synth. Lect. Comput. Archit.* 3 (1) (2008) 1–207.
- [33] Perf wiki, 2022, https://perf.wiki.kernel.org/index.php/Main_Page. (Accessed by 15 April 2022).
- [34] R. Wysocki, CPU performance scaling, 2022, <https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html>. (Accessed by 21 May 2022).
- [35] Using DVFS on raspberry pi, 2022, <https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#using-dvfs>. (Accessed by 03 May 2022).
- [36] C.J.C.H. Watkins, *Learning from Delayed Rewards*, King's College, Cambridge United Kingdom, 1989.
- [37] H. Van Hasselt, A. Guez, D. Silver, Deep reinforcement learning with double q-learning, in: *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30, no. 1, 2016.
- [38] T. Schaul, J. Quan, I. Antonoglou, D. Silver, Prioritized experience replay, 2015, *arXiv preprint arXiv:1511.05952*.
- [39] G. Bradski, *The OpenCV Library*, Dr. Dobb's J. Softw. Tools (2000).
- [40] T. Giannakopoulos, PyAudioAnalysis: An open-source python library for audio signal analysis, *PLoS One* 10 (12) (2015).
- [41] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*, IEEE, 2001, pp. 3–14.
- [42] D. Ramegowda, M. Lin, Can learning-based hybrid DVFS technique adapt to different Linux embedded platforms? in: *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation*, IEEE, 2021, pp. 170–177.
- [43] V.W. Freeh, D.K. Lowenthal, Using multiple energy gears in MPI programs on a power-scalable cluster, in: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 164–173.
- [44] M. Gupta, L. Bhargava, S. Indu, Dynamic workload-aware DVFS for multicore systems using machine learning, *Computing* 103 (2021) 1747–1769.