Deadline-Aware Deep-Recurrent-Q-Network Governor for Smart Energy Saving

Ti Zhou, Man Lin
Department of Computer Science
St. Francis Xavier University, Nova Scotia, Canada
Email: x2019cwm@stfx.ca, mlin@stfx.ca

***Abstract*—Complex cyber-physical-social systems (CPSS) consist of battery-supplied devices with low energy consumption requirements. It is essential to maintain the timing performance of computing or communication tasks while saving the device energy. Linux OS provides built-in frequency governors for power management. However, these governors are not able to incorporate the timing requirements of the application for decision-making and cannot adapt the power management decision to the specific application on target devices. This paper presents an intelligent Linux frequency Deep-Recurrent-Q-Network (DRQN) governor for dedicated applications with deadline requirements running on CPSS devices through machine learning. Although machine learning algorithms have made considerable breakthroughs in recent years, deploying them to real small devices is challenging because of the computational overhead. To tackle the computation overhead problem, an interactive learning framework is designed where the DRQN model performs only the lightest inference in the kernel while using the online data (time-series) to learn and update itself in real-time at the user level. The governor is tested on both standalone devices and networked devices. The experiment shows that DRQN can self-develop tradeoff policy to meet the user's need with low overhead. The energy saved by DRQN ranges from 7% to 33% for various deadlines.**

***Index Terms*—Deep-Recurrent-Q-Network, Reinforcement Learning, Linux Kernel, Mobile Devices, Power Management, CPU.**

I. Introduction

CYber-physical system-social systems (CPSS) are systems composed of interconnected computing devices, machinery, and digital machines to provide personalized services. A variety of recent research focuses on various aspects such as energy, security, and response time for CPSS systems, with different perspectives and different methods [1], [2], [3], [4], [5], [6], [7], [8], [9].

With the widespread of CPSS systems, reducing the power consumption of the CPU for the battery-powered devices in CPSS systems is one of the hottest topics in CPSS. Response time of services is also essential in CPSS systems. Response time requirements can be used in CPSS when making a decision on energy saving. For example, a vision assist system running on a car or a drone needs to regularly capture the surrounding scene with a camera and then analyze it using some algorithms. This timed task only needs to end before the subsequent request is triggered, so the speed performance of some tasks can be reduced to save energy. A speech analysis system involves devices for audio sampling and devices for analyzing audio data. The device used for audio sampling may send new data to the processing device at regular intervals. The processing device only needs to finish processing the previous batch of data when the next task comes, so there is an opportunity to reduce speed to save energy.

Among the CPSS hardware devices such as processor, memory, sensors, Bluetooth, camera, etc., the processors, normally designed to have multiple performance levels, consume a large part of the energy. Dynamic Voltage/Frequency Scaling (DVFS) [10] is a common technique for power management in modern processors to save energy by reducing the frequency of the CPU. The question is how to find a good tradeoff between energy consumption and timing requirements of the application tasks (computing or networking) to achieve energy saving as much as possible while satisfying the timing requirement of CPSS. This paper focuses on incorporating the timing QoS requirement of the application into a CPSS device managed by Linux to find an intelligent DVFS governor through machine learning.

A. Motivation: Intelligent Governor

Linux kernel supports DVFS governors [11]. The built-in Linux dynamic governors evaluate the system utilization and adjust CPU frequencies based on pre-defined rules. With the default settings, the operation of these dynamic governors can be seen as an attempt to conserve energy without sacrificing performance too much. However, the dynamic built-in governors are not able to incorporate the QoS requirement of the CPSS requirement to make decisions for power management. Energy will be wasted if the deadline for the computing or communication tasks is far away.

The built-in dynamic governors do provide some parameters for the user to fine-tune to customize the tradeoff between energy and performance. For example, the $up\ threshold$, one of the parameters provided by $Ondemand$, is an integer between 0 and 100. $Ondemand$ will set the frequency to the maximum value if the CPU load is above $up\ threshold$. However, the timing and energy behavior of the application on a device is determined by the underlying device architecture, network capacity, and the application itself. It is challenging for an application developer to set these parameters provided by the governor to satisfy the QoS requirements and save energy as much as possible as it is, in general, unclear how these parameters change the timing performance and energy

behavior of the application on the target device. To find a good tradeoff, users will have to spend a long time for trial and test. This is cumbersome if the timing requirement or the target device is changed, the whole process of trial and test will have to redo.

Therefore, there is a need to design an intelligent governor that can incorporate the deadline requirement of an application to save the most energy when running the application on the target device. The intelligent governor's decision-making should be able to self-adapt itself (fundamentally different from the built-in Linux governors, which use scaling policy based on human-defined rules) to the response time requirement, the application workload, the device architecture, and network capability. We assume no prior information about the device system architecture is explicitly available, which means that the governor needs to learn from data collected during device execution to find policies beyond human-defined rules, which are traditionally set based on assumptions.

B. Challenges and the Proposed Method

In order to be able to access the system performance counters as easily as existing governors at the kernel level, we choose to implement our proposed intelligent governor at the kernel level.

Reinforcement learning (RL) [12] is an area of machine learning concerned with how software agents ought to take actions in an environment to maximize the notion of cumulative reward. Reinforcement learning has been proved useful for training an agent [13], [14].

Reinforcement learning can shine in the game area because the game software supports unlimited trial and error, which reinforcement learning relies heavily on upon. Real environments often do not provide such support. Some control strategies may put the system environments (such as autopilot or operating system schedulers) into an irreversible state of damage. For DVFS modules, it is unlikely to damage the system with any control strategy ($Performance$ and $Powersave$ are already the two extreme cases). We take advantage of this by using reinforcement learning as the main body of the model.

We use the execution time of a task to express the CPU performance for training the model. The cause of a timeout does not simply lie in the current execution step but spreads over a long sequence of historical execution. In order to deal with time-series data (kernel execution traces), we choose to combine Q reinforcement learning [12] with Recurrent Neural Network as the machine learning model. The proposed intelligent governor is thus a Deep-Recurrent-Q-Network Governor (DRQN). The reason for using a recurrent network is that RNN can extract features of the time sequences. If otherwise extracted manually by humans, the features of the time sequence could be inaccurate and may not reflect the dynamics of sequences of other applications or devices.

Although machine learning algorithms have made considerable breakthroughs in recent years, deploying them to small devices is challenging. One of the main challenges in applying AI to system design is the computational overhead. Online learning models are necessary to provide the ability to self-adapt to unknown environments. In many scenarios, models need to perform inference and updates at a high rate. Unlike running in a simulated environment, such as a game, the real system does not pause to wait for the model to finish running. So excessive computational latency is unacceptable. We observe that the training part takes up the majority of space and computational overhead for deep reinforcement learning. We design a training framework that keeps only the leanest code in the kernel for inference while using the data for training at the user level in real-time. In this framework, only a small amount of computational overhead is introduced in the kernel.

C. Contributions

The paper makes the following contributions.

- We proposed and designed an intelligent DVFS Governor operating in the Linux kernel that saves energy through online learning with the input of time-series of system workload and CPU operating frequency. The DRQN DVFS Governor can adapt itself to the deadline requirement and the workload of the application on the target device.
- We designed an interactive framework to deploy deep reinforcement inference with low overhead at the kernel level, which interacts with the user level that performs learning.
- We tested the governor and the framework on real-time applications mixed with CPU-intensive calculation, storage I/O, and networking communication tasks.

II. Related Works

A. Improving/Optimizing QoS for CPSS

Cyber-physical-social systems, which consist of complex system components and provide services for computing, communication, and sensing, et al., have gained a lot of attention in the last decade. A variety of recent research focuses on various QoS factors such as energy, security, response time, and location-based service for CPSS systems, with different perspectives and different methods. [2] proposed a method for conserving position confidentiality of position-based services (PBSs) for roaming users and achieving timely services. Machine learning methods used includes decision tree, KNN, and hidden Markov models. [3] addresses cybersecurity and energy consumption by proposing an energy-aware green adversary model. [4] reduces the energy consumption of operating sensors by transferring gathered data from the supernode to the sink using Bat Algorithm (BA) to select optimum sensor nodes and paths. [5] proposed a partially computation offloading method in Mobile-Edge Computing using a metaheuristic algorithm to produce a near-optimal solution to achieve low energy consumption. [6] proposed an optimization method to maximize the profit of collaborative computation on a cloud and edge computing system based on many factors like CPU, memory and bandwidth resources, load balance of nodes, max energy, etc., demonstrated by realistic data-based simulation.

[15] proposed strategies to address energy-latency tradeoff in task overloading problems in vehicular fog computing.

Our work focus on saving energy for CPSS using dynamic frequency scaling (DVFS) with a known end-to-end response time requirement. Most of the above optimization problems need to have knowledge or assumption about the tasks and the execution environment. Our method assumes no prior knowledge about the underlying device architecture and network bandwidth, and our method is an online approach that learns the frequency scaling decision based on experience replay.

B. Machine Learning DVFS Methods

There has been much research on using machine learning-based power management by learning from the external environment patterns. [16] proposed a supervised learning-based method using the Bayesian classifier to reduce the overhead of the power manager. A power manager normally runs at a high frequency to control the state of CPUs, which may consume unnecessary energy. Chen et al. treated a DVFS problem as a pattern classification problem [17]. They use counter propagation networks to read in some system counters, classify the task behavior, and then predict the system's next frequency. These methods focus on building a prediction model which can understand the current system state but still makes decision greedily.

Different from supervised learning, reinforcement learning learns from the feedback of actions taken. It aims at an overall score of a serial of actions. Shen et al. proposed an approach that applies Q-learning to achieve autonomous power management [18]. [19] proposed using hybrid DVFS scheduling for real-time systems based on reinforcement learning where the system learns to choose the best DVFS technique depending on the system state. Hui et al. use double Q-learning to reduce Q values' overestimation, which shows better performance than Q-learning [14]. However, these methods are based on table-stored parameters. With the breakthroughs in many areas in recent years, deep neural networks showed great potential in the ability of non-linear learning and handling complex features. Online learning also has drawbacks in a real environment for the latency and cost of training. It will worsen if the model requires floating-point online training, which is an extra burden in the Linux kernel. In this work, we propose Q-Net power management, which uses a neural network as a function approximation, and trained offline in userspace. Q-Net power management was also used in [20]. However, the corresponding algorithm was not implemented as a governor in the Linux Kernel.

Many energy-aware scheduling methods only show the effectiveness of using simulator [21] and/or with an implementation that requires scheduler support [22], [23]. In this work, we proposed a learning-based DVFS governor that considers deadline constraints and does not need cooperation from any kernel scheduler module.

C. Compared to Conventional DVFS Governors

Linux kernel implements a $CPUFreq$ driver and supports static DVFS governors ($Performance$ and $PowerSave$) and dynamic DVFS governors ($Ondemand$, $Schedutil$, and $Conservative$) [11]. The static governors set the frequency of CPU to a fixed value ($Performance$ sets the frequency to Max, and $PowerSave$ sets the frequency to Min). Linux also implements a $Userspace$ Governor, which provides interfaces for the user to set the CPU frequency. Since the user level does not have access to the system performance counters as easily as the kernel level, $Userspace$ cannot achieve the same high speed and low overhead scaling as other Governors.

The Dynamic governors evaluate the utilization of the system and set future CPU frequencies based on greedy strategies. In $Ondemand$ and $Conservative$ governors, the utilization metric indicates the number of instructions being executed by the processor during a given period. $Schedutil$ uses the utilization metric calculated by the scheduler's Per-Entity Load Tracking (PELT) mechanism that adds up each processor's load not only in the last sampling rate but also the history load with discounted value. $Conservative$ governor behaves similarly as $Ondemand$ governor, except that the frequency is increased or decreased gracefully rather than jumping to the desired level.

Fig.1 shows the difference between $DRQN$ governor and $Ondmeand$ governor when selecting frequency.

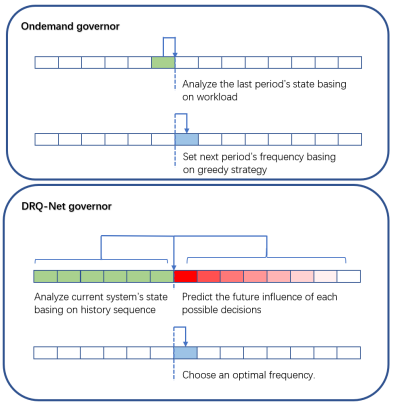

Fig. 1. Difference between $Ondmeand$ and $DRQN$ governor.

Next, we summarize the differences between our Learning-based governor and conventional governors.

- Conventional governors assume that the current state will reflect future profiles. However, the workload pattern of the tasks to be executed varies.
- Conventional governors set the frequency to a high level if the current CPU utilization is high, even the task could run at a low speed to save energy since its deadline is far away; thus, the performance constraint of the task, e.g., deadline of the tasks are not taken into account by the existing governors;
- The execution behavior (time and energy consumed) of the same workload on a different device with different architecture is different. Thus, it is hard to set the threshold parameters in conventional governors as a suitable threshold value may be different for different tasks and devices.

III. DRQN Power Management

In this section, we describe the design of Deep-Recurrent-Q-Network for our Governor (DRQN Governor) and the interactive training framework to overcome the computation overhead.

Before introducing the DRQN network, we first describe the observable of the device that the governor controls.

We define the interval between two frequency updates as a period. A tuple $s_t = (frequency, workload)_t$ is used to describe the observation at time t, including the chosen frequency and the system workload in period t. There are many algorithms for predicting workloads in cloud systems or data centers [24], [25]. In this work, we do not predict workload. Instead, it can be observed in the Linux kernel. In the Linux kernel, $workload$ indicates CPU usage in the previous duration, and it is an unsigned integer ranging from one to one hundred.

$Workload$ is the main metric used by $Ondemand$ Governor. In this work, we do not introduce more performance counters than what is used in conventional Governor, thus demonstrating that it is possible to make intelligent decision through machine learning with the existing performance counter used in a conventional governor.

Our model normalizes $workload$ value between 0 and 1.

$$workload = \frac{active\ time\ of\ CPU}{total\ time} \tag{1}$$

ϕ_t is used to indicate the sequence of the observations up to t, that is, $\phi_t = s_0, s_1, \ldots, s_t$. Note that $\phi_t = \phi_{t-1} + s_t$.

A. The DRQN Network Configuration

Like game playing in Atari [13], a frequency scaling decision at a time point affects the subsequent observation of the device, and the sequence of decisions all contribute to the final reward. Note that a good decision at t relies on not only the observation at t: s_t , but also the sequence of observations up to t: ϕ_t. The intelligent governor thus needs to use the running history of the device (the past system load and frequencies) to make frequency scaling decisions and consider future rewards when evaluating a decision at a time point. Since recurrent network (RNN) is suitable for modeling sequential data, a deep recurrent Q network (DRQN) is chosen as the learning model.

The detailed structure of the deep recurrent Q network (DRQN) is shown in Fig. 2. The agent contains a gated recurrent unit (GRU) ceil and a Q-Net (neural network) with one hidden layer. We chose GRU-based RNN because it is powerful in extracting features of time sequences yet has a small computation overhead. The size of the hidden matrix for the ceil is 8, and the input size for the ceil is 2. The neural network structure is 8-16-15, and the activation function for the NN is ReLU.

The advantage of GRU is that it can encode the sequence of historical observations with an internal vector. We denote this internal history vector as h_t, which represents the internal GRU system state. Note that h_t encodes the history information up to time t.

Table III-A summarizes the symbols used in the paper.

TABLE I
SYMBOLS

t	time instance t
s_t	observation of (workload, frequency) at t
ϕ_t	$s_0, s_1, \ldots s_t$: a sequence of observations up to t
a_t	action selected at t
f_t	frequency at t: the corresponding frequency for a_t
h_t	internal history vector in GRU
$Q(h_t, a)$	Q-value for for action a given history vector h_t
r_t	reward at t
y_t	expected Q-value at t
G	GRU function
D	transition table
γ	future discount factor, constant

B. Inference

The DRQN governor performs inference in period t by taking in s_t as input, deriving the new system vector h_t based on h_{t-1} and s_t. Then the governor produces the Q-value for each frequency (each action) based on the system vector and the neural network parameters. The frequency level (f_t) (the output) is chosen by looking up the frequency table T and is sent to the hardware to regulate the frequency. The details can be found below.

DRQN Inference:

Obtain input s_t

$h_t = G(h_{t-1}, s_t)$

$a_t = max_a Q^*(h_t, a),\ f_t = T[a_t]$

The output for the governor at time t is the chosen action a_t, which is the frequency level (f_t) to send to the hardware to regulate the frequency.

C. Learning (Training)

The goal of training the DRQN model is to derive a model that can produce a good frequency scaling decision for the application to run on the target device. The frequency scaling decision will change the device's frequency level and, in turn, change the execution dynamics (the execution speed and the workload level) of the device. The *reward* function plays a vital role in guiding the training of the model. The immediate reward at period t, r_t, is set to reflect our objectives: low energy consumption and not missing the deadline. When the deadline is satisfied, the immediate reward is define as $r_t = 1.0 - \frac{f_t}{freq_{max}}$, otherwise, $r_t = penalty\ value$ if deadline is missed. In our experiment, the $penalty\ value$ is -5.

The model is updated through back propagation (see Fig. 2). The *loss function* for the back propagation is calculated as difference of the derived Q-value $Q^*(h_t, a_t)$ of the inference and the target Q-value y_t. The target Q-value y_t takes into account the immediate reward r_t and the discounted future Q-value:

$$y_t = r_t + \gamma \max_{a'} Q(h_{t+1}, a')$$

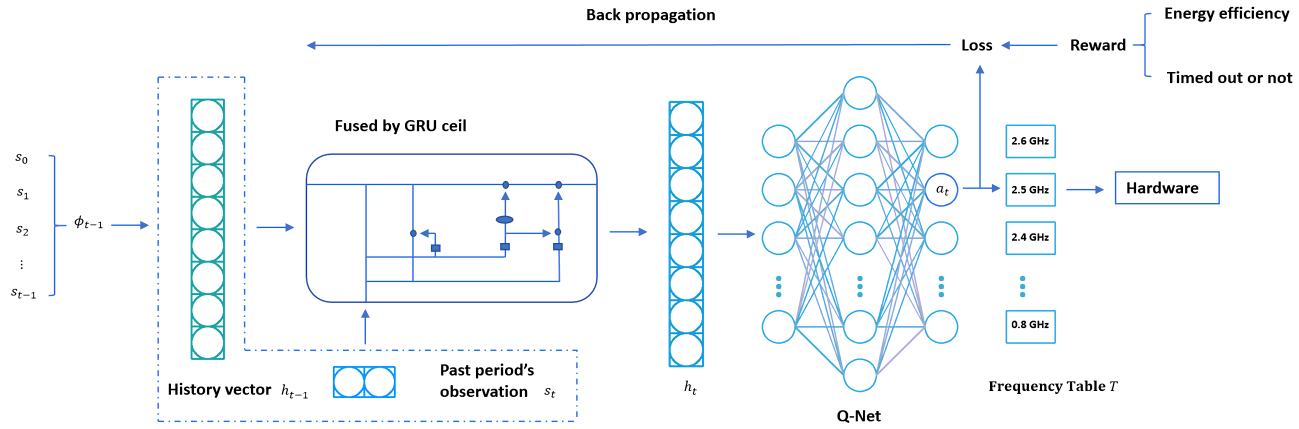

Fig. 2. DRQN Network.

if t is not at the terminal state; otherwise, $y_t = r_t$. Here, γ is the future discounted rate, The $\varepsilon - greedy$ strategy is used when selecting an action to allow model exploration. When $\epsilon = 0$, the action a_t is selected based on the Max Q-value. Otherwise, random action will be chosen based on probability ϵ.

Besides the optimizing and fine-tuning tools that support backpropagation, training for deep reinforcement learning models often includes an experience pool for supporting the experience replay mechanism. The purpose of adopting experience replay is to remove correlations in the observation sequence, as done in [13]. The replay memory table D stores transitions $(\phi_{t-1}, a_{t-1}, \phi_t, y_{t-1})$. At each inference step, an entry of the transition will be added to D. Once a certain amount of experience has been collected, the training component will update the model parameters.

D. Why Using an Interactive Inference-Training Framework

The experience collection must actually run in the device environment to gather feedback for the next training, so complete offline learning is not feasible.

Now, the question is whether a complete online kernel-level DRQN governor is feasible.

Algorithm 1 shows an an intuitive implementation of such integrated inference and training governor in kernel. Because many parameter in the network needs to be updated, learning (model update) will not be done in every inference step because of the computation overhead. In each learning step, a mini-batch of transitions sampled from the experience replay table D is used to update the network model through back propagation.

As we can see, with the history vector h_{t-1} that stores the encoded information of the historical data up to $t-1$, the inference does not introduce a high overhead as only one computation step is required: $h_t = G(h_{t-1}, s_{t-1})$ to produce the new system state h_t. The training phase, on the other hand, requires a huge amount of computation. First, any sample from

Algorithm 1: Kernel Level Online Learning DRQN Governor

Input: frequency table T
Initialize replay memory D to capacity N
Initialize gated recurrent unit G with random weights
Initialize action-value function Q with random weights
Initialize history vector h_0 with zeroes
Initialize sequence $\phi_0 = \{\}$
for period $t = 1$ to $t = END$ do **do**
  Obtain the current observation s_t
  Perform Inference as in section III-B
  $h_t = G(h_{t-1}, s_t)$
  if $\epsilon = 0$ **then**
    $a_t = max_a Q^*(h_t, a)$
  else
    Randomly select action a_t based on probability ϵ
  end if
  $f_t = T[a_t]$
  Set f_t to hardware
  Calculate reward r_t based on observed feedback
  $\phi_t = \phi_{t-1} + s_t$
  if s_t is the terminal **then**
    $y_t = r_t$
  else
    $y_t = r_t + \gamma \max_{a'} Q(h_{t+1}, a')$
  end if
  if $t > 1$ **then**
    Store transition $(\phi_{t-1}, a_{t-1}, \phi_t, y_{t-1})$ into D
  end if
  if training step **then**
    Perform backprorogation and update network parameters using samples from D.
  end if
end for

the transition table $(\phi_{k-1}, a_{k-1}, \phi_k, y_{k-1})$ can be selected. The G function needs to be applied k (the length of the sequence) times to derive h_k when processing the sample. Second, the update of many network parameters is costly due to the computation of gradients, and the many parameters need to be updated.

We tested the latency difference of one inference and one learning step on an Intel I5-7200U CPU. With our implementation, a DRQN Governor inference takes only 0.004 seconds at the highest CPU frequency and 0.01 seconds at the lowest CPU frequency. Fig. 3 shows that $\frac{Training\ time}{Inference\ Time}$ has an approximately linear relationship with the sequence length. When the sequence length is 1000, the average time required for training is almost 600 times longer than that required for inference.

Assume that the system makes ten CPU frequency adjustments per second. For 1000 frequency adjustments, the time span is 100 seconds. The process results in an observation sequence with length = 1000. Each inference takes only 0.004 to 0.01 seconds. But learning from the collected experience pool with transitions of length ranging from 1 to 1000, each learning step may take 0.004*600 = 2.4 seconds to 0.01*600 = 6 seconds. Such a high training overhead is not acceptable at the kernel level as a kernel-level task is designed to run high-priority instructions.

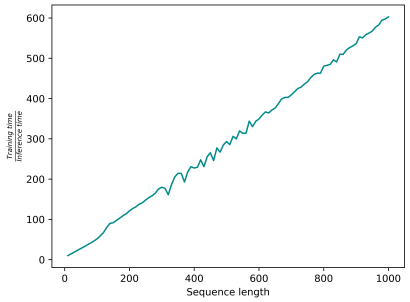

Fig. 3. Training time divided by inference time.

Therefore, we design a user-kernel Interleaving offline training framework.

E. User-kernel Interleaving Offline Training Framework

We separate the training module from the inference module, where the inference module collects the experience data and sends it to the training module. Then the training module uses the data to update the model parameters and then passes the parameters to the inference module. With this design, the training module can be implemented in the user level and therefore does not put computation pressure on the kernel level. Fig. 4 and Fig. 5 shows the idea.

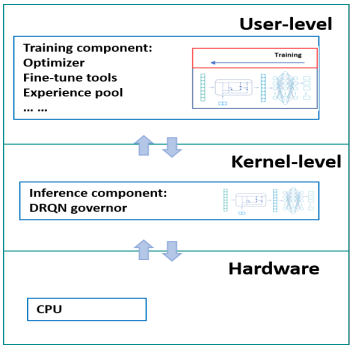

Fig. 4. User-kernel Interleaving offline training framework

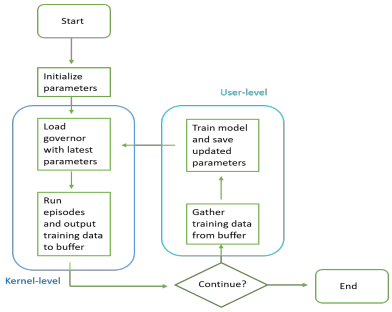

Fig. 5. User-kernel Interleaving offline training flow chart

Algorithm 2 shows DRQN inference running at the kernel level. Algorithm 3 shows how the DRQN governor is trained at the user level.

Note that only necessary computing and data observations are kept at the kernel to reduce computation and memory overhead at the kernel level. The user-level can derive the needed values for computing the loss function from the sequence of transitions (s_{t-1}, a_t, s_t, t) stored in D.

IV. Evaluations and Analysis

A. Experimental Settings

In this section, we describe the evaluation experiments. The experiment environment uses devices powered by Ubuntu 18.04 with Linux kernel 5.4.45. The CPU is Intel Core i5 7200U that supports 15 frequencies in the range from 400Mhz to 2500Mhz. Intel Turbo Boost is off, and the sampling rate is set to 100000 us for each tested governor. Table IV-A shows the environment setting.

The setting for the algorithm is as follows: The optimizer for backpropagation is Adam. The learning rate is 1e-3. For reinforcement learning, the discounted factor is 0.9. The chance of randomly selecting an action is initially 0.5 and gradually decreases thereafter. The total number of episodes trained is 600. At the end of each training session, we test the performance of the model by adjusting the chance of randomly selected actions to zero.

We tested the governor with three applications, where the workloads are mixed with CPU-intensive calculation, storage, I/O requests, and networking communication requests. Two of the applications run on a single device, and one application runs on connected devices communicating via Bluetooth.

Algorithm 2: Interleaving Learning-based DRQN inference at the kernel-level

Initialize transition table D
Initialize action-value function Q with weights loaded from buffer
Initialize gated recurrent unit G with weights loaded from buffer
Initialize frequency table T
Initialize history vector h_0 with zeroes
Initialize state s_0
for period $t = 1$ to $t = END$ **do**
  $h_t = G(h_{t-1}, s_t)$
  if $\epsilon = 0$ **then**
    $a_t = max_a Q^*(h_t, a)$
  else
    Randomly select action a_t based on probability ϵ
  end if
  $f_t = T[a_t]$
  Set f_t to hardware
  observe CPU load l_t during this period
  Set $s_t = f_t, l_t$ and normalize s_t
  Store transition (s_{t-1}, a_t, s_t, t) in D
end for
Output D to buffer

Algorithm 3: Interleaving Learning-based Training DRQN agent at the user-level

Initialize action-value function Q with random weights
Initialize gated recurrent unit G with random weights
Initialize frequency table T
Initialize task set S
for episode = 1 to M **do**
  Export Q and G's parameters to buffer
  Execute S
  Read transition table D from buffer
  for each mini-batch of transitions (s_{t-1}, a_t, s_t, t) from D **do**
    Restore time series ϕ_t from s_0 to s_{t-1} from D
    Initialize history vector h_0 with zeroes
    $h_t = G(h_0, \phi_t)$
    $h_{t+1} = G(h_t, s_t)$
    if t within the deadline **then**
      $r_t = 1 - T[a_t]/T_{max}$
    else
      $r_t = penalty\ value$
    end if
    if s_t is the terminal **then**
      $y_t = r_t$
    else
      $y_t = r_t + \gamma \max_{a'} Q(h_{t+1}, a')$
    end if
    Perform a gradient descent step on $(y_t - Q(h_t, a_t))^2$
  end for
end for

TABLE II
EXPERIMENT SETTING

Processor	Intel Core i5-7200U @ 2.60GHz
Core Count	2
Thread Count	4
Extensions	SSE 4.2 + AVX2 + AVX + RDRAND + FSGSBASE
Cache Size	3 MB
Microcode	0xde
Core Family	Kaby/Coffee/Whiskey Lake
OS	Ubuntu 18.05 with Linux kernel 5.4.45
Energy Measurement	Intel RAPL
DVFS Governor Sampling Rate	100000 us
Number of Frequency Levels	15

We use the same Deep-Recurrent-Q-Network and learning algorithm across the three different applications. The kernel-level governor only keeps the parameters for inferring the frequency level, and the user-level takes the feedback from the kernel level and performs training. The Intel RAPL (Running Average Power Limit) driver [26] is used to measure the CPU on-core energy consumption. We will compare our algorithm with the built-in dynamic governor in the Linux kernel in this section.

B. Benchmark description

Benchmark 1 is a computer-vision-related real-time application: The system runs the following periodic task ten times. The period for the tasks is 6.0 seconds. Every 6.0 seconds, the system sends a request to take a picture via camera, then classifies the image taken by the camera via a ResNet 50 model, and subsequently saves the result to local storage. This benchmark contains external device calling (camera), CPU intensive calculation (ResNet 50), and storage I/O.

Benchmark 2 is a networking application. This experiment is tested on two devices communicating via Bluetooth. The system sends a request every five seconds. In each request, device one first transcodes two 20M MP3 files to Wav format and then sends a string to inform device two when it completes the transcoding via Bluetooth. Thus, this benchmark contains a CPU-intensive task and a networking task.

Benchmark 3 is an application that integrates a computation-intensive computing task and an IO-intensive computing task. It contains a fast Fourier transform (FFT) computation task and a large amount of small (4KB) file writing task. The period for the application is 6.0 seconds.

Table III shows the average performance of the three dynamic governors in Linux default in terms of execution time for the three benchmarks.

C. Deadline-Awareness

What we are interested in most is the ability of the DRQN model to sense and adapt to the deadline requirements. We trained three sets of models for benchmark 1 with three

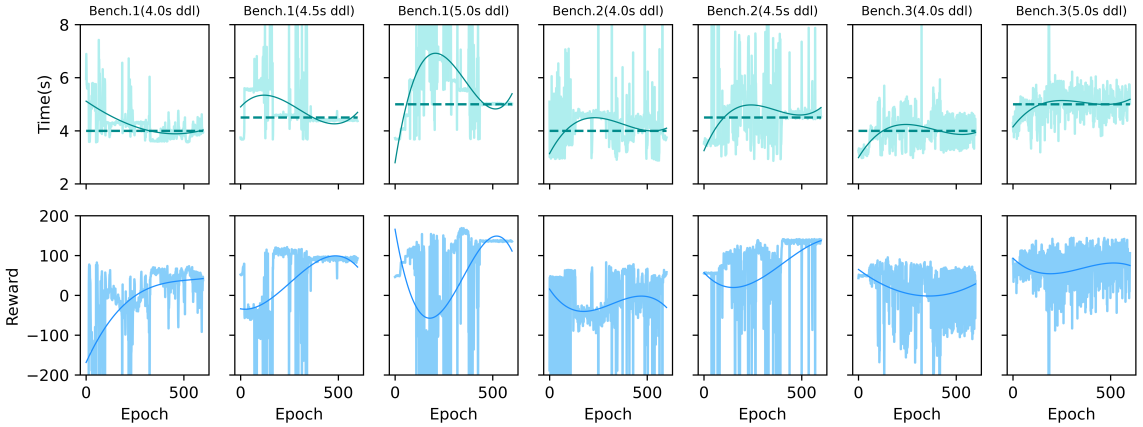

Fig. 6. Changes in reward values and target task execution times harvested during training.

TABLE III
EXECUTION TIME OF BENCHMARKS WITH CONVENTIONAL BUILT-IN GOVERNORS .

Governor	Benchmark 1	Benchmark 2	Benchmark 3
Ondemand	3.75s	3.12s	3.20s
Schedutil	3.86s	3.58s	3.33s
Conservative	4.24s	4.01s	3.96s

deadlines = 4.0s, 4.5s, and 5.0s; two sets of models for benchmark 2, with two deadlines = 4.0s and 4.5s; and two sets of models for benchmark 3 with two deadlines = 4.0s and 5.0s, respectively.

Fig. 6 shows the variation of the reward values and target task execution times during the training process at the end of each epoch. At the end of each epoch training, the system runs the benchmark three times with the newly trained DRQN governor. The average task execution time is then calculated, and a reward is obtained. We can see that as the number of training epochs increases, the model's control will guide the application's execution time closer to the given deadline.

With both the reward value and the task execution time at the end of each epoch in the same figure, Fig. 6 allows us understand the decisions made by the governor during the training course. Initially, the parameters of the model are random, so the initial control strategy is also random. Thus, in each of the seven sets of graphs, the model's performance (execution time) control at the beginning is different. Note that in our setting, the lower the frequency the model selects, the higher the reward harvested (if there is no timeout). Considering benchmark 1 with deadline = 5.0s, we notice that the reward values keep increasing in the first 100 epochs. This suggests that in the first 100 epochs, the model steadily tries to reduce the frequency selection as we can observe that the running time of the application increases in the first 100 epochs. However, a timeout may occur when the selected frequency is too low because the task runs too slow. If a timeout is triggered, the reward value suddenly becomes very low because of the penalty. At this point, the learner will adjust the model not to select a further reduced frequency to satisfy the deadline requirements and start learning to do a tradeoff. For benchmark 1 (4.0s deadline), the timeout penalty is triggered by the policy at the beginning of the model.

D. Comparison of DRQN and Linux built-in Governors

To compare the DRQN governor with the built-in governors [11], we first show the charts that depict the power varied with execution time for the DRQN and the Ondemand Governors when running different applications. To save space, we only show the charts (see Fig. 7 and Fig. 8) for benchmarks 1 (period = 6.0s) and benchmark 2 (period = 5.0s). As CPU performance decreases (execution time becomes longer), the peak value of power consumption decreases. Ending tasks more quickly allows the CPU to enter an idle state sooner. Taking benchmark 1 as an example (Fig. 7), Ondemand governor finishes execution in less than four seconds, so the CPU can get about two seconds of idle time, which is shown in the chart as a power spike followed by a rapid decay to a trough. DRQN governors adapt to the deadline. For governors that have a longer deadline, such a trough in power consumption will take longer to come and may not even come. DRQN (5.0s) working on benchmark one barely got much idle time. Charts in Fig. 8 for benchmark 2 also confirm similar pattern.

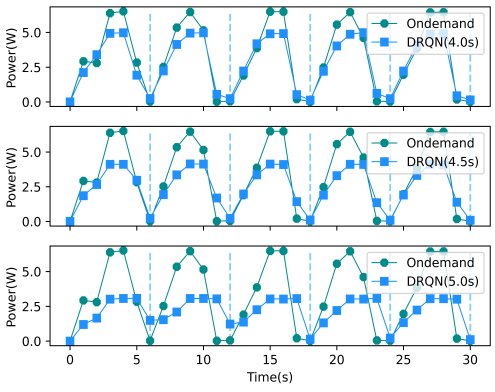

Fig. 7. Comparison of DRQN and Ondemand energy consumption per second on benchmark 1

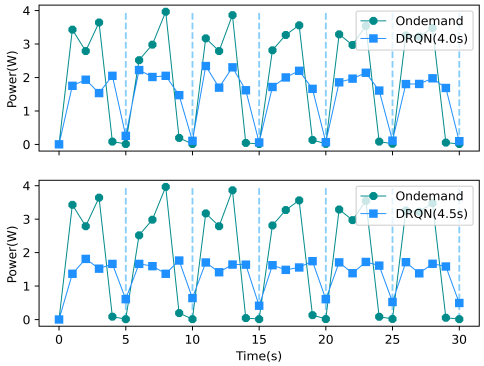

Fig. 8. Comparison of DRQN and Ondemand energy consumption per second on benchmark 2

Next, we compare the energy saving for different governors with respect to the Ondemand governor (see Fig. 9, Fig. 10 and Fig. 11). The figures record the energy consumption of the system with ten requests of each application. Because CPU energy consumption is highest under Ondemand's control, we use Ondemand as a baseline to compare the effect of energy savings of other governors.

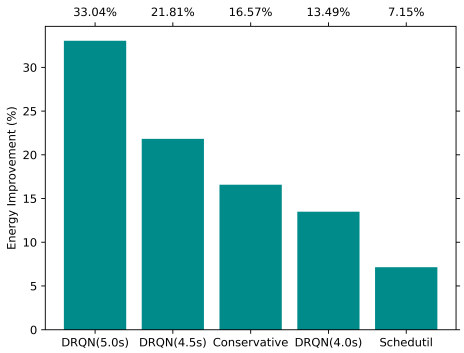

Fig. 9. Energy savings compared to Ondemand on Benchmark 1

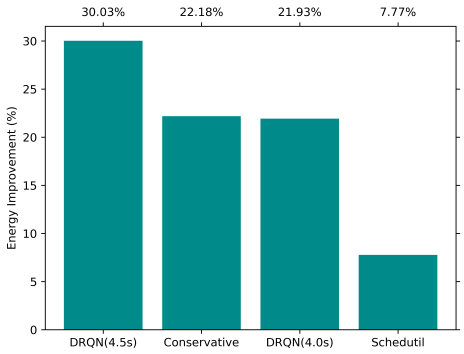

Fig. 10. Energy savings compared to Ondemand on Benchmark 2

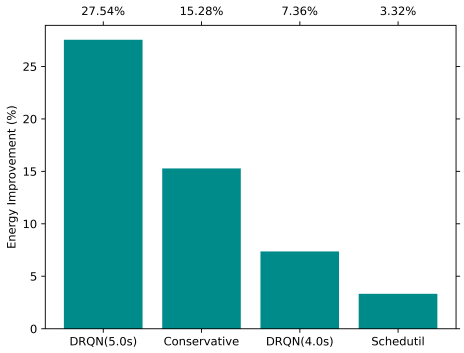

Fig. 11. Energy savings compared to Ondemand on Benchmark 3

The energy saved by DRQN compared to Ondemand governor ranges from 13% (deadline = 4.0s) to 33% (deadline = 5.0s) for benchmark 1, 22% (deadline = 4.0s) to 30% (deadline = 4.5s) for benchmark 2 and 7% (deadline = 4.0s) to 28% (deadline = 5.0s) for benchmark 3. Basically, the results are in line with the principle that the further away the deadline, the lower the energy consumption. This means that DRQN can successfully adapt to the deadline and slow down the task to save energy. Such performance/energy tradeoff is found intelligently through machine learning.

The benefits gained by entering idle status early cannot be ignored. For DRQN(5.0s) on benchmark 1, although the peak energy consumption is less than half of Ondemand, the total energy consumption is only 33% less than Ondemand.

V. Conclusion

No existing works have implemented a kernel-level intelligent energy-saving DVFS governor that can adapt to the application deadline requirement on a target device. This paper has proposed a Deep-Recurrent-Q-Net-based (DRQN) power management method that uses deep reinforcement learning to self-explore energy performance tradeoff policy to meet users' needs. The DRQN model contains a feature extractor to extract feature vectors from historical sequences as input to Q-Net. In essence, the framework learns the DVFS policy from the collected experience replay when running the CPSS applications. We have studied the time needed for DRQN inferences and training and observed that the time needed for training with long sequences of experience replay is much higher than that for inference. Therefore, to effectively deploy DRQN to CPSS devices that require low overhead, we developed an interactive learning framework that distributes the model's training and inference to the user level and kernel level, respectively. The framework satisfies the training requirements for deep reinforcement learning while significantly reducing the burden of the model on system operation. The governor is tested on both standalone devices and networked devices with various applications that have mixed computing, IO, storage, and network communication tasks. The energy saved by DRQN compared to Ondemand governor ranges from 7% to 33% depending on the deadline requirement, which shows that our method can save dramatic

energy while satisfying system needs on standalone devices and networked devices. Using machine learning models to explore system performance/energy tradeoff control strategies can significantly reduce human labor. However, models that can handle complex problems often require large amounts of computation, which is challenging to operate in an operating system environment. With our implementation, the kernel retains only the lightest inference code. A single inference takes 0.004s-0.01s, which is still higher than the latency of existing heuristics. A device may have less computation power in a real industrial environment, and overhead requirements are stricter. Thus, exploring methods including quantification and pruning to reduce kernel overhead further is one of the future directions. Other future directions include exploring other learning structures combining with optimization methods that can make a better decision. We will also work on interpreting the policies learned by the DRQN model, which is a problem that explainable AI strives to solve.

ACKNOWLEGEMENT

The authors thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for supporting this research.

REFERENCES

[1] J. Zeng, L. Yang, M. Lin, and et.al., "A survey: Cyber-physical-social systems and their system-level design methodology," Future Generation Computer Systems, vol. 105, pp. 1028–1042, 2020.

[2] A. K. Sangaiah, D. V. Medhane, T. Han, M. S. Hossain, and G. Muhammad, "Enforcing position-based confidentiality with machine learning paradigm through mobile edge computing in real-time industrial informatics," IEEE Transactions on Industrial Informatics, vol. 15, no. 7, pp. 4189–4196, 2019.

[3] A. K. Sangaiah, D. V. Medhane, G.-B. Bian, A. Ghoneim, M. Alrashoud, and M. S. Hossain, "Energy-aware green adversary model for cyber-physical security in industrial system," IEEE Transactions on Industrial Informatics, vol. 16, no. 5, pp. 3322–3329, 2020.

[4] A. K. Sangaiah, M. Sadeghilalimi, A. A. R. Hosseinabadi, and W. Zhang, "Energy consumption in point-coverage wireless sensor networks via bat algorithm," IEEE Access, vol. 7, pp. 180 258–180 269, 2019.

[5] J. Bi, H. Yuan, S. Duanmu, M. Zhou, and A. Abusorrah, "Energy-optimized partial computation offloading in mobile-edge computing with genetic simulated-annealing-based particle swarm optimization," IEEE Internet of Things Journal, vol. 8, no. 5, pp. 3774–3785, 2021.

[6] H. Yuan and M. Zhou, "Profit-maximized collaborative computation offloading and resource allocation in distributed cloud and edge computing systems," IEEE Transactions on Automation Science and Engineering, pp. 1–11, 2020.

[7] M. Lin, Y. Pan, L. T. Yang, M. Guo, and N. Zheng, "Scheduling co-design for reliability and energy in cyber-physical systems," IEEE Transactions on Emerging Topics in Computing, vol. 1, no. 2, pp. 353–365, 2013.

[8] G. Xie, G. Zeng, J. Jiang, C. Fan, R. Li, and K. Li, "Energy management for multiple real-time workflows on cyber-physical cloud systems," Future Generation Computer Systems, vol. 105, pp. 916 – 931, 2020.

[9] H. Aydin, V. Devadas, and D. Zhu, "System-level energy management for periodic real-time tasks," in 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06), 2006, pp. 313–322.

[10] M. Weiser, B. Welch, A. J. Ddmers, and S. Shenker, "Scheduling for reduced CPU energy," in Proceedings of the 1st OSDI, 1994, pp. 13–23.

[11] R. J. Wysocki, "Cpu performance scaling," 2017, accessed: 26-04-2021, kernel version = 5.12.0. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html

[12] R. Sutton and A. Barto, Reinforcement Learning. MIT Press, 2018.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in NIPS Deep Learning Workshop, 2013.

[14] H. Huang, M. Lin, L. T. Yang, and Q. Zhang, "Autonomous power management with double-q reinforcement learning method," IEEE Transactions on Industrial Informatics, vol. 16, no. 3, pp. 1938–1946, 2020.

[15] R. Yadav, W. Zhang, O. Kaiwartya, H. Song, and S. Yu, "Energy-latency tradeoff for dynamic computation offloading in vehicular fog computing," IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14 198–14 211, 2020.

[16] H. Jung and M. Pedram, "Supervised learning based power management for multicore processors," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29, no. 9, pp. 1395–1408, Sept 2010.

[17] Y.-L. Chen, M.-F. Chang, C.-W. Yu, X.-Z. Chen, and W.-Y. Liang, "Learning-directed dynamic voltage and frequency scaling scheme with adjustable performance for single-core and multi-core embedded and mobile systems," Sensors, vol. 18, p. 3068, 09 2018.

[18] H. Shen, Y. Tan, J. Lu, Q. Wu, and Q. Qiu, "Achieving autonomous power management using reinforcement learning," ACM Transactions on Design Automation of Electronic Systems, vol. 18, no. 2, pp. 1–32, April 2013.

[19] F. M. M. u. Islam and M. Lin, "Hybrid DVFS scheduling for real-time systems based on reinforcement learning," IEEE Systems Journal, vol. 11, no. 2, pp. 931–940, 2017.

[20] Q. Zhang, M. Lin, L. T. Yang, Z. Chen, and P. Li, "Energy-efficient scheduling for real-time systems based on deep q-learning model," IEEE Transactions on Sustainable Computing, vol. 4, no. 1, pp. 132–141, 2019.

[21] A. Saifullah, S. Fahmida, V. P. Modekurthy, N. Fisher, and Z. Guo, "CPU Energy-Aware Parallel Real-Time Scheduling," in 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020), vol. 165, Dagstuhl, Germany, 2020, pp. 2:1–2:26.

[22] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware scheduling for real-time systems: A survey," ACM Transactions of Embedded Computing System, vol. 15, no. 1, Jan. 2016.

[23] C. Scordino, L. Abeni, and J. Lelli, "Energy-aware real-time scheduling in the linux kernel," in Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018, pp. 601–608.

[24] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization, 2007.

[25] J. Gao, H. Wang, and H. Shen, "Machine learning based workload prediction in cloud computing," in 2020 29th International Conference on Computer Communications and Networks (ICCCN), 2020, pp. 1–9.

[26] S. Desrochers, C. Paradis, and V. Weaver, "A validation of DRAM RAPL power measurements," 10 2016, pp. 455–470.

Ti Zhou Ti Zhou is currently a graduate student in Computer science at St. Francis Xavier University. His research interests include IoT, Green Computing, Machine Learning, and Operating Systems.

Man Lin Man Lin is an IEEE senior member. She received the B.E. degree in computer science and technology from Tsinghua University, Beijing, China, and the Ph.D. degree from Linköping University, Sweden. She is currently a Professor in Computer Science at St. Francis Xavier University, Canada. Her research interests include Cyber-physical Systems, Real-Time Scheduling, Power-aware Computing, Machine Learning, System Analysis and Optimization Algorithms.