A Deep Q-Learning Approach for Dynamic Management of Heterogeneous Processors

Ujjwal Gupta, Sumit K. Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y. Ogras

Abstract—Heterogeneous multiprocessor system-on-chips (SoCs) provide a wide range of parameters that can be managed dynamically. For example, one can control the type (big/little), number and frequency of active cores in state-of-the-art mobile processors at runtime. These runtime choices lead to more than 10× range in execution time, 5× range in power consumption, and 50× range in performance per watt. Therefore, it is crucial to make optimum power management decisions as a function of dynamically varying workloads at runtime. This paper presents a reinforcement learning approach for dynamically controlling the number and frequency of active big and little cores in mobile processors. We propose an efficient deep Q-learning methodology to optimize the performance per watt (PPW). Experiments using Odroid XU3 mobile platform show that the PPW achieved by the proposed approach is within 1 percent of the optimal value obtained by an oracle.

Index Terms—Heterogeneous multi-cores, Power management, Deep reinforcement learning

1 Introduction

Dynamic thermal and power management (DTPM) has been prevalent in computing systems, especially battery powered mobile platforms. The complexity of DTPM decisions increase with the number of controllable variables, such as the number and types of cores. For example, Snapdragon 810 and Exynos 5422 platforms allow controlling the performance and power states of four little and four big cores at runtime. Power management drivers built into the Linux kernel utilize these knobs in a variety of DTPM algorithms.

Power management decisions are taken by the device drivers, which are executed periodically every 50 ms-100 ms. This means, more than 600 power management actions are taken every minute of operation. Deriving the optimal power management decisions is not tractable at runtime due to large decision space, uncertainty in workloads, and imperfections of the power-performance models used for decision making. Reinforcement learning (RL) is a promising approach, since it can exploit abundant data collected at runtime to learn the optimal decisions without relying on an explicit model [11].

Among different RL algorithms, the most popular choice has been table-based Q-learning, i.e., Q-Table, due to its simplicity and strong convergence properties. It is used for maximizing performance of CPUs under a power budget [3] and thermal constraint [4]. More recently, a Q-Table technique is used to select the best DVFS policy among multiple candidate policies to minimize the CPU power consumption [13]. But, the tabular nature of Q-Table requires dividing the state space into discrete bins with different granularities [3], [4]. Each state-action pair is represented by a separate row in the table. For the case when F features are divided into B bins, the number of rows for each action becomes B^F. Similarly, K knobs and C possible actions per knob leads to C^K actions. As a result, the total number of rows in Q-Table becomes $B^F C^K$. State-of-the-art SoCs have 4 or more knobs (number and frequencies of big/little cores). Even when we limit the actions to 3 (up, down, no change), there are more than 3^4 actions per state. To represent a large application space, a rich set of features need to be used. If 15 features are employed as in our setup, then even with as few as 4 bins, the total number of rows in the Q-Table becomes $4^{15} \times 3^4$. Assuming 64 bit floating point representation, a Q-Table requires about 650 GB of memory. Hence, the Q-Table approach is not practical for dynamic power management of state-of-the-art SoCs.

The scalability problem of Q-Table is addressed using function approximation [11]. For example, a linear model or a Deep Q-network (DQN) is used to approximate the values stored in the Q-Table. Since function approximation uses continuous state values, it eliminates the need for discretization of the states, as required in Q-Table approach. In this work, we present a deep Q-learning approach to dynamically manage the type, number and frequency of active cores in SoCs. The state space consists of CPU core utilizations, power consumption, and CPU core and frequency configurations in addition to six hardware performance counters (detailed in Section 3.1). The proposed DQN takes the state as the input and produces four different actions that control 1) the number of little cores, 2) frequency of little cores, 3) the number of big cores, 4) frequency of big cores. The proposed approach evaluates each action in terms of the resulting performance per watt using a novel reward function. Then, the reward is used to update the weights of the DQN. It has been shown that the performance of the different DQN approaches depends on the training set [8]. Since large training sets are prohibitive at runtime due to memory requirements, we also propose a novel *prioritized* experience replay mechanism that allows effective training with a small memory footprint.

The major contributions of this paper are as follows:

- We present the first DQN approach which can dynamically control both the frequency and core configuration of SoCs,
- We develop a novel prioritized experience replay technique and a reward function for effective training of the DQN,
- We present comprehensive experimental evaluations on Odroid XU3, and compare our approach to Q-Table and double DQN approaches.

2 Related Research

Several Q-Table techniques have been proposed to manage the resources of multicore systems. For instance, the authors in [4] propose a controller for maximizing the performance under thermal constraints for multimedia applications by applying DVFS on a CPU. Similarly, the technique presented in [3] maximizes the performance of a multicore system under a power budget constraint using a Q-Table approach. A recent technique reduces the Q-Table size significantly by adaptively transferring information with a smaller number of bins as workloads change [9]. As we detail in Section 4.2, Q-Table does not scale to modern SoCs even with small number of bins due to large memory requirements.

DQNs have recently been applied to a large variety of problem domains ranging from computer games to mastering the game of Go [10], [11]. The main advantages of DQN over Q-Table are eliminating the need for discretizing the states and smaller storage requirements. It has been shown that double deep Q-learning (DDQN) can alleviate potential overestimation of Q value targets, and provide higher stability in learning [14]. A recent work applies this idea successfully for dynamic power management by approximating Q-Tables with two stacked auto-encoders [15]. Our work

- *The authors are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281. E-mail: {ujjwal, skmandal, mmao7, chaitali, umit}@asu.edu.*

Manuscript received 31 July 2018; revised 10 Dec. 2018; accepted 3 Jan. 2019. Date of publication 9 Jan. 2019; date of current version 7 Feb. 2019.
(Corresponding author: Sumit K. Mandal.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2019.2892151

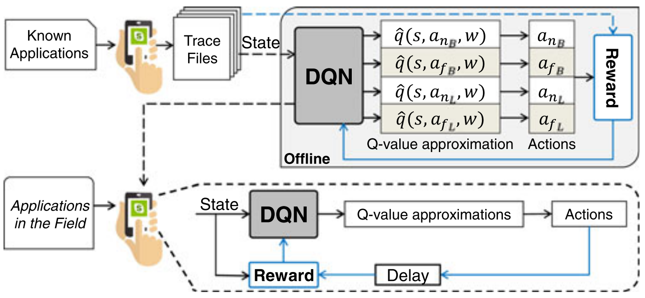

Fig. 1. Overview of the DQN technique for DPTM.

differs from this DDQN approach in two major aspects. First, the technique presented in [15] targets randomly generated periodic real-time tasks, whereas our approach uses standard benchmarks on real platforms that do not make any assumptions on the underlying workloads. Second, it *controls only the frequency of a single resource, i.e., one variable*. In strong contrast, our approach controls four parameters (the number of little and big cores and their frequencies) simultaneously. Also, we design a fundamentally new type of reward function to handle multiple processing elements and configuration knobs. We empirically compare the Q-Table and double DQN approach with our technique in Section 4.

RL approaches require a reward function, which can be avoided by imitation learning (IL). Recent work has shown that IL can outperform RL for known applications [7]. IL approximates the policy using linear function and nonlinear functions in the form of trees, whereas we employ a nonlinear function in the form of a neural network. We plan to explore a combination of offline IL policy and online learning in our future work.

3 Deep Q-Learning Methodology for DTPM

This section starts with an overview of our DQN framework. Then, we present offline training of the DQN in Section 3.2, online learning using experience replay in Section 3.3, and design of the reward function for multiple actions in Section 3.4. We focus on the novel aspects of this work. Further details for standard DQN can be found in [10], [11].

3.1 Overview and Preliminaries

In RL, an agent (DTPM algorithm) reads the environment (platform) state and takes actions, such as controlling the operating frequency. The objective of the agent is to learn a policy that maximizes a cost function, such as, performance per watt (PPW), by taking optimal actions.

Actions. We control four knobs: the number of active little cores n_L, operating frequency of active little cores f_L, number of active big cores n_B, and operating frequency of active big cores f_B. Each action can take three values, which are encoded as 0 (decrease), 1 (no change), and 2 (increase).

State. We define the state as the core utilizations, sum of little core utilizations, big and little core frequencies, number of big and little cores, total power consumption, and five *normalized* performance counters listed below. Thus, we have a total of 15 state variables in our experiments.

Performance Counters. The behavior of the workload is captured through the following hardware performance counters: *CPU Cycles, Branch Miss Prediction, L2$-misses, Data Memory Access, and Noncache External Memory Requests, all normalized by the number of Instructions Retired*. Using a universal set of counters that capture the workload behavior, we ensure that our policy is not application-dependent. Hence, it can learn new applications efficiently at runtime, as shown by our experimental results.

Policy. The policy takes the state and outputs the Q-values (quality) of each possible action. Then, the action with the maximum Q-value, i.e., the one that maximizes the PPW, is applied by the agent to the environment. After each action, we obtain a new state and a reward from the environment. The reward is then used to improve the policy of the agent. Since it is useful to explore the rewards due to all types of actions, we use an epsilon-greedy policy [11].

Fig. 1 shows the overview of the offline training and runtime use of the DQN. The policy is modeled with a DQN where the Q-values are a function of state S, action A and neural network weights $\mathbf{w}$, $Q = q(S, A, \mathbf{w})$. The state vector is used to predict Q-values for each possible action. There are 12 Q-values, since we have four configurations knobs $\{n_L, f_L, n_B, f_B\}$ and three possible actions for each knob. At each control interval, the policy chooses the action that leads to the maximum Q-value. Finally, the weights of the DQN are updated using current state S_t, current action A_t, next state S_{t+1}, and the reward R_{t+1}:

$$\Delta\mathbf{w} = \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w}). \quad (1)$$

In this equation, α is the learning rate, γ is the discount factor, t is the control interval.

3.2 Training Data and Oracle

Training the DQN requires the optimum configuration, i.e., the set of active cores and their operating frequency that give the largest PPW for a given state action pair. To design an Oracle, we first instrument the applications known at design time such that they are partitioned into basic blocks called snippets. We achieve this by inserting PAPI calls at snippets within the application using LLVM compiler [5]. After instrumenting 16 single- and multi-threaded benchmarks from MI-Bench [6], Cortex [12], and PARSEC [2] suites, we obtain snippets with an average length of 16.5 ms. The instrumented code records the performance counters listed in Section 3.1.

We run each application while sweeping the number of big and little cores from one to four, and their respective frequencies in steps of 0.2 GHz from 0.6 GHz to 2 GHz. Besides the performance counters, we also measure the power consumption using the built-in sensors in the target Odroid-XU3 platform. Using this data, we compute the PPW for each snippet-configuration pair, and construct a table that gives the optimal PPW for each snippet. We perform an iterative search on this table to find the sequence of configurations that give the maximum PPW for each snippet. We note that finding the optimal Oracle would have required a combinatorial search. The collected data and Oracle are also used to compute the reward function and train the DQN using Equation (1).

3.3 Prioritized Experience Replay

The quality of DQN training may suffer from rare or correlated data updates [8], but these problems can be eliminated by using experience replay [10]. Experience buffer stores the most recent current action, current state, next state, and reward. Then, a mini-batch is sampled uniformly from the experience buffer and the DQN is trained on it using Equation (1).

Deep Q-learning on high-end computers can utilize experience replay buffer having size in order of megabytes [10]. However, the memory overhead has to be minimized when implementing DQN in the power management drivers of SoCs. Consequently, the experience replay buffers can store only a limited number of data points. This, in turn, can lead to an unbalanced representation of configurations. To avoid this, we propose a priority scheme that stores only representative samples in the experience buffer. For example, if configuration $\{1\,\mathrm{L}, 0.8\,\mathrm{GHz}, 2\,\mathrm{B}, 1\,\mathrm{GHz}\}$ is seen more often than others, we balance the representation of both types of data samples. Our technique achieves this by inserting new data points to the experience buffer by using the following priority function:

TABLE 1
Illustration of Generating the Reward Sign

Optimal PPW & actions ($\mathbf{x}^*$)					Current PPW & actions ($\mathbf{x}_a$)					Cos Sim.
PPW	a_{n_L}	a_{f_L}	a_{n_B}	a_{f_B}	PPW	a_{n_L}	a_{f_L}	a_{n_B}	a_{f_B}	c
1.400	1	0	1	2						0.87
1.395	2	1	2	2	1.200	1	1	0	2	0.82
1.390	1	2	0	2						0.96

The unit of PPW is Instr./nJ. The action 0, 1, 2 imply decrease, remain same, and increase, respectively, for a configuration knob.

$$p_i = 1 - e^{k\frac{n_i}{n_T}}, \forall i \in [1, N], \tag{2}$$

where n_T is the total number of configurations in the optimal PPW set, and n_i is the number of occurrences of configuration i. k is a gain parameter that changes the shape of the distribution and N is the total number of configurations supported by the platform. This function enables us to increase the priority of data points that have less occurrence in the dataset and vice versa. After computing the priorities, we find the probability of inserting configuration i to the experience replay buffer as:

$$Pr(i) = \frac{p_i}{\sum_{i=1}^{N} p_i}, \forall i \in [1, N]. \tag{3}$$

It is also common practice to balance the positive and negative reward samples. Therefore, we monitor the positive and negative reward samples added to the experience buffer and then use this information to prioritize the addition of new samples. For example, if the number of positive reward samples is larger than the negative samples, we prioritize the negative samples.

3.4 Design of the Reward Function

We construct a new reward function for two reasons. First, the PPW is the combined effect of *four distinct actions*. Hence, these actions cannot be evaluated individually in isolation. Second, the PPW achieved by multiple configurations may be within the measurement error. Suppose the maximum observed PPW is given as PPW^*. To avoid arbitrary decisions, we define the set of actions that achieve $PPW \geq (1-\epsilon)PPW^*$ as optimum ($\epsilon = 0.01$ in this work). For instance, Table 1 illustrates a scenario with three sets of optimal actions. With these considerations, the sign and magnitude of the reward is computed using the set of optimal actions, the current action, and their PPWs.

The Sign of the Reward. There may be multiple actions with very close PPWs, as illustrated by the first column in Table 1. Thus, we must first choose the PPW-action pairs that needs to be compared against our policy. To achieve this, we first form a vector that combines the PPW and actions. For example, the PPW and action of our policy is given as $\mathbf{x}_a = [1.200, 1, 1, 0, 2]$ in Table 1. Similarly, we construct the PPW^*-action vector $\mathbf{x}_i^*$ for each set of optimal actions. Then, we compute the cosine similarity c_i between the current PPW-action vector and each set of optimal actions as:

$$c_i = \frac{\mathbf{x}_a . \mathbf{x}_i^*}{|\mathbf{x}_a||\mathbf{x}_i^*|} \tag{4}$$

The vector with the largest cosine similarity is used to generate the reward sign. According to the illustration in Table 1, the closest set to our PPW-action vector is $\mathbf{x}_{i=3}^* = [1.390, 1, 2, 0, 2]$. Then, the sign of the reward is obtained by comparing each action in $\mathbf{x}_a$ with the corresponding action in $\mathbf{x}_{i=3}^*$. The reward is positive, if the actions match, and it is negative otherwise. In our example, the actions for a_{n_L}, a_{n_B}, and a_{f_B} match with the optimal set, while a_{f_L} does not. Thus, the signs of the reward for each action are obtained as $[+, -, +, +]$.

The reward magnitude. We modulate the magnitude based on the distance between PPW^* and the PPW achieved by the current actions. When the policy produces a PPW close to its optimal values, the magnitude of the reward function should be small. To this end, we first compute the relative difference between the current and optimal value PPW as $\Delta = \frac{|PPW^* - PPW|}{PPW^*}$. If Δ is less than a margin, such as 1 percent, the magnitude is suppressed. Otherwise, it is amplified to increase the rate of convergence. We achieve this by computing the magnitude of the reward as:

$$R_m = \frac{e^d}{e^d + e^{-d}}, d = \frac{\Delta - \mu}{\sigma} \tag{5}$$

where μ and σ are parameters used to control the shape of the reward. We empirically find that $\mu = 0.004$ and $\sigma = 0.01$ give the best results for our training workloads.

4 Experimental Evaluation

4.1 Experimental Setup

We employ Odroid-XU3 platform running Ubuntu OS with kernel 3.10. It contains Samsung Exynos 5422 application processor with four little (A7) and four big (A15) cores, whose frequencies and power states can be managed at runtime. We run 16 single- and multi-threaded workloads from Mibench, SPEC, and CORTEX suites, as explained in Section 3.2. We implement a DQN with two hidden layers of 14 neurons each in Python with Keras APIs running Tensorflow-backend. We employ Adam optimizer with an initial learning rate of 0.004. We define the model efficiency as the ratio of total number of correct actions taken and the total number of possible actions. The learning rate is reduced to 0.001 and eventually 0, when the model efficiency exceeds 80 and 90 percent, respectively.

4.2 Overhead Analysis and Comparison to Q-Table

We implemented the proposed approach on the target platform. The whole experience buffer takes up 334 kB, while size of the replay buffer is 38 kB. The runtime overhead of our implementation is about 2.96 μs, which is negligible compared to the 10-200 ms control interval.

We emphasize that the proposed DQN approach can select *both the frequency and core configuration, since it is scalable*. The size of comparable Q-Table implementation with 15 state variables is prohibitive even if the Q-Table is minimized using novel transfer techniques [9], as we explain in Section 1. It exceeds 100 GB even with 4 bins for each state variable. To demonstrate accuracy comparisons, we employ the PCA application, for which 1L (little) – 1B (big) is the dominant optimal configuration. Hence, the Q-Table approach assumes 1L–1B core configuration, and *manages only the frequency levels, while our approach discovers both the core configuration and frequency*. The PPW obtained by using the Oracle, proposed DQN and Q-Table approaches are shown Fig. 2a. The PPW is within 1.5 percent for both DQN and Q-Table compared to the Oracle. This means that function approximation using DQN works as good as Q-Table. Figs. 2b and 2c show the big core frequency and number of active cores for the three algorithms. The accuracy of correct actions using DQN is 91.2 percent and the corresponding configurations match closely with the Oracle. There is only 3 percent difference in PPW of the Oracle and DQN for the regions with spikes in the Oracle in Fig. 2c.

4.3 Evaluation Using Workload Mixtures

Offline Model Evaluation. We train the model offline with 13 workloads out of 16 available workloads. Then, we generate workload mixes by randomly choosing 5 workloads out of 16 workloads. For instance, the first mix 1 (W-mix 1) contains Patricia, Basicmath, Spectral, Motion Estimation, and FFT, that are run one after the other. Out of these 5 workloads, two (Spectral and Motion Estimation) were not present at the time of training. Figs. 3a and 3b

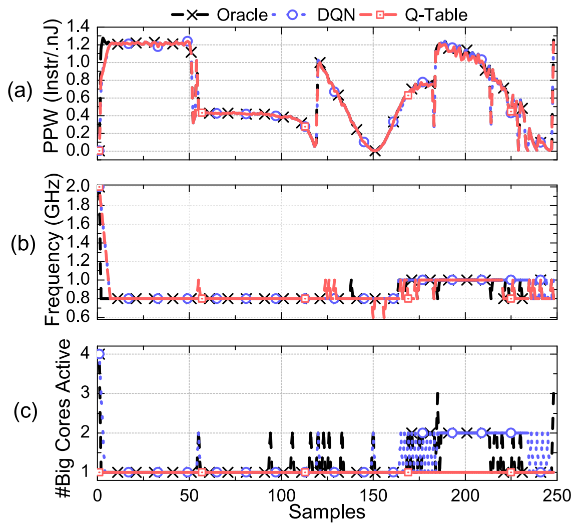

Fig. 2. Oracle, DQN, Q-Table results for the PCA benchmark: (a) PPW, (b) big core frequency, (c) number of active big cores.

compare the results of DQN against the Oracle by showing the mean absolute percentage error (MAPE) of PPW and relative number of correct actions, respectively. On average, the PPW achieved by our offline model is within 1 percent of the optimal PPW obtained by the Oracle. Moreover, actions taken by offline model are correct on average 95 percent of the time with respect to the Oracle. Even the worst accuracy, which is observed for W-mix 3 is 88 percent, and the corresponding PPW is within 1.8 percent of the Oracle.

Online Learning with Unknown Workload-Mix. We also perform online learning of the DQN, such that its weights can be updated at runtime. In this case, the applications in the workload mixes were not seen by the model during offline training. The Oracle actions are not known for the unseen workloads. Hence, we use power and performance models developed offline using linear regression [1] to compute the PPW achieved by all possible actions. Finding the reward takes 1.53 μs using the PPW obtained by these models. Fig. 3a shows that the PPW achieved by model learned online is within 1 percent of the Oracle. Similarly, the actions are correct 93 percent of the time, as shown by the blue bars. We achieve good accuracy for unseen applications since multiple training applications enable the policy to see diverse types of system states, which are representative of a broader set of potentially unseen applications. Features obtained with a new application (e.g., instructions retired, CPU cycles) provide useful insight because of two reasons. First, we employ 15 universal features. Second, training set exercises these features over a large number of application snippets. Even when two applications are different, they have similar phases, such as loading the data, similar loops, etc.

Comparison Against Double DQN. We also implemented a DDQN policy [14], [15] using our methodology. The double DQN achieves comparable action accuracy and PPW *across all the workload mixes*. The DDQN approach results in a negligible difference (≈ 1 percent) since potential overestimations of our DQN policy do not necessarily lead to negative performance [14]. We conclude that the proposed prioritized experience replay policy is able to provide good accuracy with a single DQN.

Summary of the Results. We have shown that the proposed DQN policy performs well for mix of known and unknown, and completely unknown workloads. Hardware implementation shows that runtime overhead and storage requirements of our approach is significantly lower than Q-Table approaches. Finally, the proposed approach generates comparable results to recently proposed DDQN techniques with lower overhead.

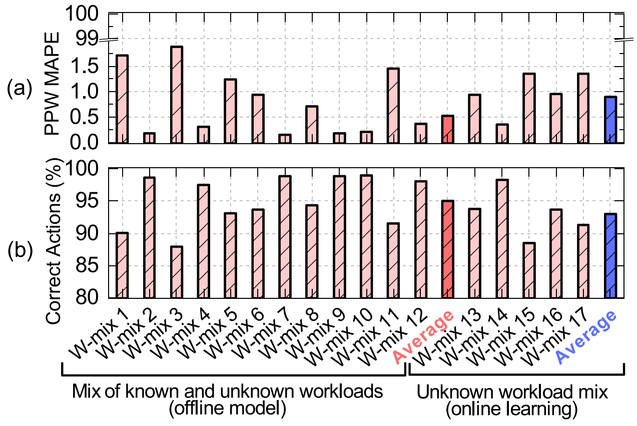

Fig. 3. Comparison of DQN technique with Oracle on known and unknown test workload mixtures.

5 Conclusions

We presented a deep reinforcement learning approach for dynamic resource management of SoCs. We employed a DQN to maximize the PPW by optimally controlling the number of active heterogeneous CPU cores and their frequency. To achieve this, we developed a novel prioritized experience replay technique and a reward function for effective training of the DQN. The proposed DQN approach achieves a PPW within 1 percent for 17 different application mixes with high action accuracy of 95 percent.

Acknowledgments

This work was supported by NSF grant CNS-1526562 and Semiconductor Research Corp. task 2721.001.

References

[1] G. Bhat, et al., "Online learning for adaptive optimization of heterogeneous SoCs," in *Proc. ICCAD*, 2018, p. 61.

[2] C. Bienia, et al., "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Arch. Compilation Tech.*, 2008, pp. 72–81.

[3] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *Proc. DATE*, 2015, pp. 1521–1526.

[4] Y. Ge and Q. Qiu, "Dynamic thermal management for multimedia applications using machine learning," in *Proc. 48th ACM/EDAC/IEEE Des. Autom. Conf.*, 2011, pp. 95–100.

[5] U. Gupta, et al., "DyPO: Dynamic pareto-optimal configuration selection for heterogeneous MpSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, 2017, Art. no. 123.

[6] M. R. Guthaus, et al., "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. Int. Workshop Workload Char.*, 2001, pp. 3–14.

[7] R. G. Kim, et al., "Imitation learning for dynamic VFI control in large-scale manycore systems," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 25, no. 9, pp. 2458–2471, 2017.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," [Online]. Available: https://arxiv.org/abs/1511.05952, 2015.

[9] R. A. Shafik, et al., "Learning transfer-based adaptive energy minimization in embedded systems," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 35, no. 6, pp. 877–890, Jun. 2016.

[10] D. Silver, et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction, 2nd ed.*, vol. 1, no. 1. Cambridge, MA, USA: MIT Press, 2017.

[12] S. Thomas, et al., "CortexSuite: A synthetic brain benchmark suite," in *Proc. Int. Symp. Workload Characterization*, 2014, pp. 76–79.

[13] F. M. M. ul Islam and M. Lin, "Hybrid DVFS scheduling for real-time systems based on reinforcement learning," *IEEE Syst. J.*, vol. 11, no. 2, pp. 931–940, Jun. 2017.

[14] H. Van Hasselt , A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI*, vol. 2, 2016, Art. no. 5.

[15] Q. Zhang, et al., "A double deep Q-learning model for energy-efficient edge scheduling," *IEEE Trans. Serv. Comput.*, 2018, doi: 10.1109/TSC.2018.2867482.