An Energy-aware Online Learning Framework for Resource Management in Heterogeneous Platforms

SUMIT K. MANDAL and GANAPATI BHAT, Arizona State University, USA
JANARDHAN RAO DOPPA and PARTHA PRATIM PANDE, Washington State University, USA
UMIT Y. OGRAS, Arizona State University, USA

Mobile platforms must satisfy the contradictory requirements of fast response time and minimum energy consumption as a function of dynamically changing applications. To address this need, systems-on-chip (SoC) that are at the heart of these devices provide a variety of control knobs, such as the number of active cores and their voltage/frequency levels. Controlling these knobs optimally at runtime is challenging for two reasons. First, the large configuration space prohibits exhaustive solutions. Second, control policies designed offline are at best sub-optimal, since many potential new applications are unknown at design-time. We address these challenges by proposing an online imitation learning approach. Our key idea is to construct an offline policy and adapt it online to new applications to optimize a given metric (e.g., energy). The proposed methodology leverages the supervision enabled by power-performance models learned at runtime. We demonstrate its effectiveness on a commercial mobile platform with 16 diverse benchmarks. Our approach successfully adapts the control policy to an unknown application after executing less than 25% of its instructions.

CCS Concepts: • **Hardware** → **On-chip resource management**; • **Computing methodologies** → *Machine learning algorithms*;

Additional Key Words and Phrases: Dynamic power management, imitation learning, reinforcement learning, online learning

ACM Reference format:
Sumit K. Mandal, Ganapati Bhat, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y. Ogras. 2020. An Energy-aware Online Learning Framework for Resource Management in Heterogeneous Platforms. *ACM Trans. Des. Autom. Electron. Syst.* 25, 3, Article 28 (May 2020), 26 pages.
https://doi.org/10.1145/3386359

1 INTRODUCTION

Over a billion people use various types of electronic devices including mobile phones, tablets, and personal computers [37]. As the processing and sensing capabilities of these devices expand, we see exponential growth in the number and types of applications. Common examples include graphics-intensive games, communication-intensive social media apps, health monitoring, and traditional

This work was supported partially by USA Army Research Office grant W911NF-17-1-0485, National Science Foundation grants CNS-1526562 and OAC-1910213, and Semiconductor Research Corporation (SRC) task 2721.001.

Authors' addresses: S. K. Mandal, G. Bhat, and U. Y. Ogras, Arizona State University, School of Electrical, Computer, and Energy Engineering, Tempe, AZ 85287, USA; emails: {skmandal, gmbhat, umit}@asu.edu; J. R. Doppa and P. P. Pande, Washington State University, School of EECS, Pullman, WA 99164, USA; emails: {jana.doppa, pande}@wsu.edu.

1084-4309/2020/05-ART28 $15.00
https://doi.org/10.1145/3386359

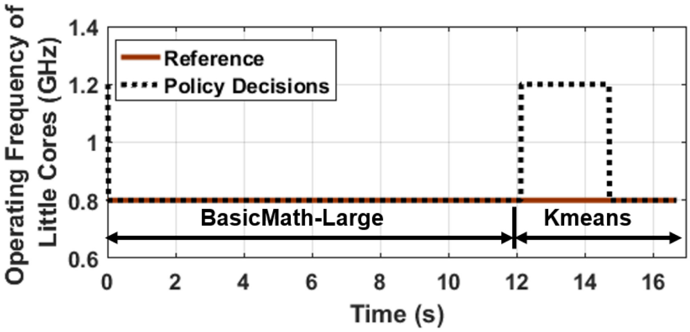

Fig. 1. Illustration of the offline policy constructed using a training set that includes BasicMath-Large. The policy decisions for BasicMath-Large are perfect for entire execution, while decisions for Kmeans are different 50% of the time with respect to the reference.

compute-intensive applications. Delivering the required performance on-demand and maximizing the battery life are two common goals independent of the application. However, the runtime configurations to achieve these goals can vary dramatically for different application scenarios. For example, high-performance CPU cores (e.g., big cores) are preferred over low-power cores while running compute-intensive applications. Furthermore, the voltage and frequency levels should be controlled optimally at runtime, since the highest (i.e., most power-hungry) levels are not needed continuously. Therefore, heterogeneous SoCs must orchestrate the utilization of the available resources optimally at runtime as the composition of active applications evolve.

Determining the optimal SoC configuration at runtime is challenging for two reasons. First, the parameter space is prohibitively large to explore at runtime. Even with two types of cores (e.g., n_B big and n_L little cores) with f_B frequency levels for big cores and f_L frequency levels for little cores, there are thousands of possible configurations ($f_B^{n_B} X f_L^{n_L}$, $f_B, f_L > 1$). Second, and more importantly, one cannot find the optimal configuration offline, since many potential applications are unknown at design-time. Even if a set of key applications, i.e., key performance indicators (KPI), are available, we do not know the precise composition of these KPIs and background applications in advance. Furthermore, the type and number of applications grow continuously. Therefore, there is a strong need for approaches that adapt to new applications by learning optimal SoC configurations at runtime.

Existing governors in mobile platforms use simple metrics for runtime power management decisions. For example, interactive and on-demand governors in Android phones control the operating frequency as a function of the utilization. These policies maximize performance, but they are not energy-efficient. Several power management techniques have recently been proposed to overcome these limitations [25, 30, 31]. These policies are typically built offline using prior knowledge of few known applications. Therefore, they may not perform well for new applications encountered at runtime. For example, Figure 1 illustrates an offline policy that performs poorly while running applications outside the training set. Reinforcement learning (RL) [40] methods, such as Q-learning, can be employed for online learning. However, RL methods are not efficient to learn the optimal policy for new applications, since they learn via trial-and-error using very weak training signals and require a large number of data samples.

In this article, we propose a novel online imitation learning (*Online-IL*) approach to learn optimal policies for new applications at runtime. The Online-IL approach leverages an offline control algorithm, constructed at design-time, as the initial policy to effectively bootstrap the learning process. This policy may not perform well on new applications seen at runtime as demonstrated by Figure 1 and our experimental results. Hence, the proposed Online-IL approach synergistically combines the benefits of power/performance models and machine learning techniques. The power/performance models provide strong supervision via low-overhead and accurate evaluation of candidate configurations. Our machine learning techniques leverage this supervision to efficiently adapt the policy to new applications. In each training epoch, we perform the following four steps: (1) Execute the current policy and record the workload metadata, such as the number of instructions and memory accesses; (2) evaluate the candidate configurations using the metadata and analytical power/performance models to determine the best action; (3) update the policy parameters such that evaluation of policy at each system state matches the best configuration found via power/performance models; and (4) if required, update the parameters of analytical power/performance models based on real measurements. We repeat these four steps until the policy convergences.

Experimental evaluations on Odroid-XU3 board show that offline policies achieve almost 100% accuracy with respect to a golden reference while running applications from the training set. However, they perform poorly under new applications seen at runtime. Our proposed online-IL methodology adapts the offline policy online to new applications within a few seconds.

Contributions: The main contributions of this work are as follows:

- A novel approach to utilize online power/performance models at runtime to generate training data,
- A runtime adaptive control algorithm that combines optimal offline policies and online learning,
- Comprehensive experimental evaluation on a commercial mobile platform using 16 diverse applications.

2 BACKGROUND AND RELATED WORK

Increasing mobile phone usage and performance demands have led to significant research in exploring power-performance tradeoffs in heterogeneous SoCs. Mobile platforms employ heterogeneous SoCs that integrate different types of cores (big/little), GPUs, and application-specific processors. They come with power management governors, such as on-demand [29], that use simple metrics including core utilization to make power management decisions, although the increasing complexity of heterogeneous SoCs necessitates new algorithms for their dynamic resource management.

The majority of prior power management techniques utilize control policies designed offline [32, 44]. For example, two recent studies propose CPU/GPU frequency selection techniques using decision tree [30, 31]. These techniques are designed offline and verified at runtime under gaming applications. Purely offline methodologies are impractical, since they cannot adapt to new applications with unknown characteristics. Therefore, a practical dynamic power management technique for mobile platforms should *automatically* adapt to new application applications seen at runtime.

One class of techniques choose between multiple pre-designed dynamic power management policies by characterizing the incoming applications online [1, 8, 9, 36]. For example, the technique presented in Reference [1] updates the parameters online, when a new uncharacterized application begins. These methods incur the cost of storing multiple control policies and switching among them online. The accuracy of these approaches depends critically on the availability of an appropriate policy, which is infeasible due to the rapidly growing number of new applications. A recent study updates the policy online with the help of a proportional-integral (PI) controller that estimates the workload behavior [45]. However, this technique controls only one knob, the uncore frequency.

Modeling power-performance with system-level measurements is usually an integral part of dynamic resource management techniques. Multiple studies in the literature discuss power modeling for heterogeneous platforms [6, 38, 46]. A comprehensive discussion about power modeling techniques is presented in References [6, 38]. One class of power models rely on the parameters obtained through circuit simulations [2, 46]. Since typical circuit simulations are slow, obtaining power models though simulations are time-consuming. Another class of techniques use empirical data to obtain the power models. Since our technique uses empirical data to obtain the power model, we discuss this class in more detail. An empirical power modeling technique is proposed in Reference [4]. In this technique, the authors collect power data from the platform itself by executing it at different fixed temperatures, voltages, and frequencies. Several researchers also proposed offline and online performance models for heterogeneous platforms. The authors of References [16, 21] proposed machine learning-based performance models as a function of hardware counters to estimate GPU performance. Auto-regressive models are used to predict the performance in heterogeneous processors both offline [10] and online [11]. Adaptive performance models for heterogeneous SoCs are proposed in References [3, 17]. In these techniques, the authors perform offline feature selection and use Recursive Least Square (RLS) to learn model coefficients online. In our proposed work, we first collect power and execution time data for different applications. Since total power is a function of dynamic capacitance (C_{dyn}) and leakage current (I_{leak}), we model these two as a linear combination of hardware counters. However, these models might be inaccurate for the new applications seen at runtime. Therefore, we update these linear models through RLS with the help of the hardware counters and the power values obtained at runtime. We apply a similar technique to estimate execution time, too.

Recent studies proposed applying Reinforcement Learning (RL) to dynamic power management to enable online adaptation [7, 43, 47]. For instance, Q-Table based approaches are used for energy-aware DVFS scheduling [7, 43]. These approaches are not scalable for heterogeneous SoCs, since the number of state-action pairs is increasing with the number of cores and supported power states. To address this problem, Zhang et al. [47] proposed a deep queue learning network (DQN) technique for heterogeneous SoCs. A common drawback of all RL methods is their inefficiency to learn the optimal policy for new applications. More specifically, RL techniques require a large number of data samples to converge, since they learn via trial-and-error using weak training signals.

In this work, we employ the principles of imitation learning (IL) [34] due to significant advantages of IL methods over RL: require fewer data and time to learn near-optimal policies. IL leverages the optimal sequential-decision making behavior (referred to as the Oracle) and employs supervised learning techniques to mimic this behavior. However, exact imitation learning algorithms suffer from error propagation. Advanced IL algorithms use additional training data collected during policy evaluation to recover from mistakes to address this problem [33]. This idea has been used to control voltage-frequency islands in a multiprocessor SoC [22]. In their approach, the control policy is created offline using known applications and evaluation is performed for homogeneous SoCs using only simulation studies. Recently, another IL-based dynamic power management technique is proposed in Reference [24] for heterogeneous platforms. In this approach, the authors design the Oracle policy using the applications known at design-time. Subsequently, they use the offline designed policy at runtime. Therefore, these methodologies cannot improve the control policy for new applications.

In contrast to prior work, we propose a novel low-overhead *Online Imitation Learning* approach that can learn to control multiple configuration knobs simultaneously as new applications arrive. Our proposed solution utilizes power/performance models to generate strong online supervision that enables us to tailor the policy to new applications seen at runtime. Finally, we evaluate the effectiveness of our solution with experiments on a hardware platform.

Table 1. Notations and Important Parameters

$\mathcal{E}$	Sequence of epochs of an application ($\mathcal{E} = \{E_1, E_2, \ldots, E_T\}$)
E_i	ith epoch of an application
C	Set of candidate configurations
C_j	Configuration on which jth epoch of an application runs
n_B	Number of active big cores
n_L	Number of active little cores
f_B	Frequency of big cores
f_L	Frequency of little cores
π	A power management policy that maps E_i to $C_i \in C$

3 PRELIMINARIES AND PROBLEM SETUP

We consider a heterogeneous platform with k different core types. Each type can contain a single core or multiple cores. For example, the platform may integrate multiple CPU cores while having a single GPU. Given such a platform, our goal is to design a power management policy to control the number of active cores and their respective frequencies. Each possible combination of the number of cores and their frequencies is a unique runtime configuration for the platform. Using this definition, we can represent all possible configurations in the system by $C = \{C_1, C_2, \ldots, C_N\}$, as summarized in Table 1. For our experimental platform, Odroid-XU3, a configuration consists of four control knobs: the number of active big cores (n_B), the number of active little cores (n_L), the frequency of big cores (f_B), and the frequency of little cores (f_L). Therefore, the goal of the power management policy is to determine the tuple (n_B, n_L, f_B, f_L) for each control interval.

The default power management governors used on mobile platforms apply decisions in periodic intervals ranging from 10 to 100 ms [29]. These decisions typically involve determining the number of active cores and their respective frequencies. Suppose that we run an application repeatedly at different frequencies. Decision intervals observed in each execution will be different from the other runs. This means that we cannot execute the same application at different configurations and collect the optimal configuration for each interval. Thus, periodic intervals impede construction of optimal oracles even for known applications.

To facilitate Oracle policy construction, we segment applications into *repeatable* decision epochs, which are clusters of macro-blocks. To this end, we use the methodology proposed by Gupta et al. [14]. We start by inserting PAPI API [28] calls within consecutive epochs using LLVM [23] and clang compiler framework. Specifically, we first find all the available basic blocks in the source code. Then, we instrument at all possible basic blocks in the application. Finally, we remove PAPI calls from basic blocks at the lower hierarchy of call graph to prune the number of epochs. As a result of the instrumentation, the number of instructions in each epoch ranges from 10–100 million instructions with a median of about 7.5 million instructions. This translates to about 50–1000 epochs per workload as a function of the total execution time of the workload. This instrumentation enables us to collect the performance counters listed in Table 2 and power consumption for each *repeatable* epoch. We note that the repeatability here means that the number of instructions executed for a given epoch are always the same regardless of the configuration they are being executed on. At the same time, the other performance counters, such as LLC misses, memory accesses, can be different each time the application is executed. Consequently, an application with T decision epochs can be expressed as a sequence of epochs $\mathcal{E} = \{E_1, E_2, \ldots, E_T\}$. Each epoch has a fixed set of macro-blocks (e.g., a while loop) ranging from 10 to 100 millions of instructions. Using these definitions, a policy $\pi : \mathcal{E} \rightarrow C^T$ is a function that maps a given epoch to

Table 2. Performance and Power Consumption Data Collected in Each Epoch

Instructions Retired	Noncache External Memory Request
CPU Cycles	Total Little Cluster Utilization
Branch Miss Prediction	Per Core Big Cluster Utilization
Level 2 Cache Misses	Total Chip Power Consumption
Data Memory Access	

one of the supported configurations. For instance, $\pi(E_i) = C_i$ means that epoch E_i should run on configuration $C_i \in C$ based on policy π.

The standard approach to obtain a policy is to first obtain data with a number of applications and then apply supervised learning methods to train a policy. However, as we show in Figure 1, policies trained offline may not perform well for unseen applications. Therefore, there is a need for methodologies that continue to learn online and adapt to unseen applications. To this end, we first present an approach using reinforcement learning to solve this problem, since it is commonly used for online learning. However, RL methods require the design of a reward function and exploration of large state space. This can lead to a slow convergence for RL policies. To overcome this, we present a novel low-overhead *Online Imitation Learning* approach that can learn to control multiple configuration knobs simultaneously as new applications arrive online.

4 CANONICAL BASELINE APPROACH: REINFORCEMENT LEARNING (RL)

Reinforcement Learning (RL) is a commonly used framework for solving sequential decision-making problems. In RL setting, a controller takes an action as a function of the current system state. In our power management problem, we define the state as the hardware counters observed in a given decision epoch. The controller takes an action as a function of the hardware counters. After the controller takes the action, it interacts with the environment. The environment provides feedback (reward) to the controller about the quality of the action taken by the controller. The learned utility of actions at states is represented in the form of Q-values. Each state and action pair corresponds to a Q-value. Depending on the reward, the controller gets to know how good or bad the previously taken action was and the corresponding Q-value gets updated. Using the reward and the current state information, the controller performs an update to its parameters such that the actions in future states can be improved. After a sufficient number of observations, the controller learns to take near-optimal action for a given state. Algorithm 1 delineates the canonical Q-learning algorithm that is popularly employed in the design automation community. There are different methodologies in the literature to implement this learning algorithm, especially to store Q-values. One of them employs a Q-table to *explicitly* store the values for each state and action pair, and another one uses a function approximator to *implicitly* store the Q-values.

Table-based approach: A well-known methodology to implement Q-learning is the table-based approach. In this setting, the Q-values represent the value of taking an action in a given state. The Q-values of all the states and the corresponding actions are stored in a table. When the controller takes an action, the environment serves the reward using a hand-designed reward function. Subsequently, it calculates the Q-value for the current state-action pair and populates the table. It has been proved that table-based RL approach will asymptotically converge to take optimal actions [40]. Table-based RL is typically used to perform power management for SoCs in existing literature [7, 12, 35, 43].

ALGORITHM 1: Q-learning Algorithm with ϵ Greedy Exploration

```
Input: Number of learning epochs ($K$), Reward function ($R$), Exploration probability ($\epsilon$)
for $k = 1, 2, \ldots, K$ do
    if rand() < $\epsilon$ then
        Choose $C_k$ randomly from $C$ // Exploration
    else
        Evaluate $Q$-value for different configurations as a function of the features of the current epoch
            ($E_k$)
        Select the configuration ($C_{pol}$) with highest $Q$-value // Exploitation
        Obtain reward ($R_k$) from the environment for the chosen configuration ($C_k$)
        Update $Q$ function following Equation (1)
        $C_k = C_{pol}$
    end
end
```

Function approximation approach: Although table-based RL approach is very efficient to learn an optimal controller, this approach scales poorly as the number of states grows beyond a small number. If there are S possible states and A candidate actions, then the number of entries in the Q-table is $O(S \times A)$. Therefore, if S is very high, then the number of entries in the Q-table will grow. Moreover, in the table-based approach, the state has to be discretized into a fixed number of bins when the state space is continuous, as is the case in our problem. If the discretization is coarse, then the learned model can suffer from low accuracy. However, if the discretization is very fine, then storing a large table will incur additional overhead. To solve these challenges with table-based policy representation, function approximation based Q-learning algorithm is proposed in the literature [27]. In this methodology, the Q-values are learned using a function approximator, such as a neural network or regression tree. If a neural network is used as a function approximator, then the Q-learning methodology is often referred to as deep Q-Learning (DQN). Deep Q-Learning has been used to manage control knobs of an SoC in prior work [15, 42, 48]. We refer the reader to [27] for more details on deep Q-learning.

The power management problem described in Section 3 has a continuous state space and a discrete action space. Therefore, we apply Deep Q-Learning [27] to obtain a policy π. The Q-values are updated following Equation (1):

$$Q(E_k, C_k) = (1 - \alpha)Q(E_k, C_k) + \alpha \left[R_k + \gamma \max_{c \in C} Q(E_{k+1}, c)\right] \tag{1}$$

where $Q(E_k, C_k)$ is the Q-value for kth epoch executed on configuration C_k, α is the learning rate and γ is the discount factor.

Reward function: The performance of RL algorithms critically depends on the design of a good reward function. We define the reward function (R_k) at kth decision epoch as follows:

$$R_k = -\frac{P(E_k, C_k) \times t(E_k, C_k)^\beta}{P_{min}(E_k) \times t_{min}(E_k)^\beta} \tag{2}$$

where $P(E_k, C_k)$ and $t(E_k, C_k)$ are the power consumption and execution time respectively at decision epoch E_k with configuration C_k. $P_{min}(E_k)$ is the minimum possible energy consumed at decision epoch E_k. Typically, when the powersave governor is active, energy consumption is minimum. $t_{min}(E_k)$ is the minimum possible execution time at decision epoch E_k. With performance governor, the platform delivers minimum execution time. Further, we can change the value of parameter β to fine-tune the optimization objective. For example, with $\beta = 0$, the reward function results in control behavior to minimize power, and with $\beta = 1$, the reward function results in control behavior to minimize energy. In our experiments, we employ $\beta = 1$, as this minimizes energy. However, we argue that deep Q-Learning based RL is not suitable for controlling heterogeneous processors when compared to our proposed online imitation learning approach. This can be attributed to the following reasons:

- *First,* the feedback obtained by the controller from the environment through the reward function is indirect. Therefore, the controller has to sufficiently explore the large state space to be able to learn a near-optimal policy. This can result in a long convergence time for the RL policy, as shown later in Section 6.4. Such a long convergence time is not desirable, since the workload composition can change frequently in mobile devices.
- *Second,* the accuracy of policy learning in RL critically depends on the reward function. However, designing a good reward function that can achieve the desired convergence is challenging.

To overcome these challenges, we propose a novel online imitation learning methodology in the next section.

5 ONLINE IMITATION LEARNING METHOD

Imitation learning (IL) is a supervised machine learning framework that aims to imitate an Oracle for sequential decision-making problems. The Oracle policy provides expert-level feedback to the policy being learned by IL. In classical IL settings, the Oracle is provided as training data from expert demonstrations (such as humans performing a task in robot learning [34]). In the case of dynamic management of heterogeneous systems, the Oracle provides the best configuration for a given epoch of the application. The IL-based controller receives the correct control action as feedback when it makes incorrect decisions. However, RL-based controllers can only observe the reward from the environment, which is weaker supervision compared to IL. It has been shown that IL policies converge exponentially faster than RL [39]. Due to this advantage, we choose IL to train our resource management policies.

Construction of an Oracle is one of the most important steps to train an IL policy. In an offline setting, this can be achieved by collecting characterization data and using advanced algorithms like dynamic programming to obtain the Oracle [24]. In literature, there are techniques to construct an offline policy using IL for dynamic management of SoCs [22, 24]. However, constructing an Oracle policy is non-trivial for an online learning setting because of two reasons. *First,* Oracle construction needs a large amount of data that are not readily available in an online setting. *Second,* the methodology to construct Oracle policy may involve expensive computation, which can be prohibitive at runtime. Therefore, there is a strong need for methodologies that can construct high-quality Oracle policies at runtime with minimal overhead. To address this challenge, we propose a four-step methodology for online imitation learning as depicted in Figure 2. These steps are outlined below and detailed in the following sub-sections.

1. Construct Control Policy Offline (Section 5.1) – The first step of the proposed methodology is to construct an offline control policy $\pi_{offline}$ using the set of training applications available at the design-time. Prior approaches use similar policies constructed offline to make runtime power management decisions [14, 30]. However, they do not learn the characteristics of new applications online and adapt the policy at runtime. In contrast, the proposed approach uses this offline policy as a starting point and employs online imitation learning to adapt it to new applications.

2. Online Execution and Power-Performance Modeling (Section 5.2) – At runtime, we use the most up to date policy trained until that point to make power management decisions.

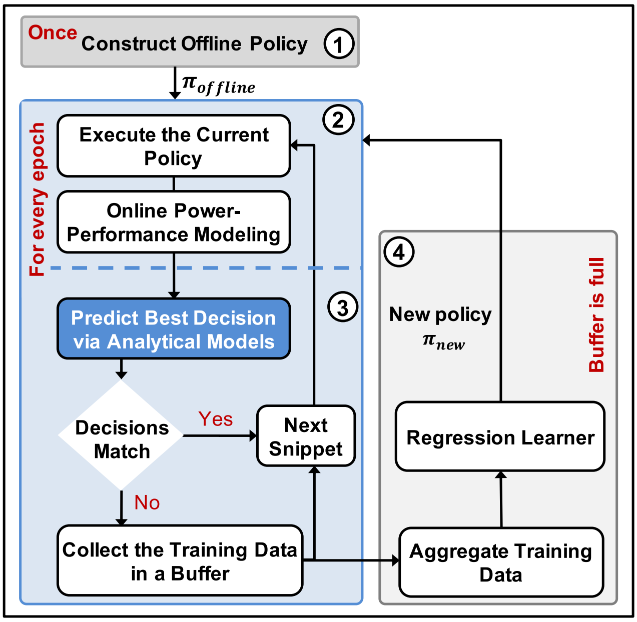

Fig. 2. Overview of the proposed framework.

Meanwhile, we also continuously collect the workload metadata summarized in Table 2. Subsequently, we employ this data for maintaining accurate runtime adaptive power-performance models with negligible overhead.

3. Online Oracle Policy Construction (Section 5.3) – The fundamental challenge in online learning is to construct a good Oracle (or reference) policy that can provide strong supervision needed to update the policy and quickly learn an optimal policy for new applications. We overcome this challenge by using our power-performance models maintained online, as shown in Figure 2. More specifically, we use these models at the end of each control interval to determine the best action with less than 0.1% runtime overhead. This retrospective view enables us to compare this action against the actual policy decision. If they do not match, then we record the metadata and optimal decision as future training data to update the policy.

4. Online Training (Section 5.4) – After a set of metadata and corresponding optimal configuration are collected, we use them as supervised training examples to incrementally re-train the policy using imitation learning. Hence, the policy adapts to new applications via supervision from the oracle constructed online.

5.1 Offline Policy Construction

We can employ any existing method to construct a control policy offline using the applications known at the design time. Without loss of generality, we use imitation learning for this purpose. Segmenting the applications into repeatable epochs, i.e., microbenchmarks, enables us to execute each epoch of an application at each supported configurations. Hence, we collect power consumption and performance data for each epoch-configuration pair at design-time. These data allow us to determine the configuration that optimizes a given metric (e.g., energy) for each epoch known at design-time. More specifically, we use this data to construct an Oracle policy $\pi^*(\cdot) : \mathcal{E} \rightarrow C^T$ that maps each epoch to the configuration that minimizes the energy consumption.

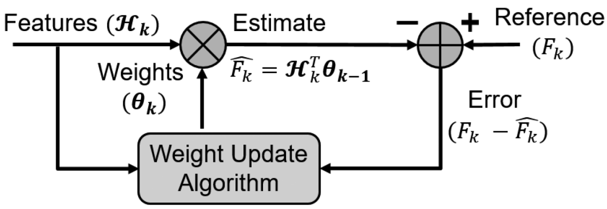
Features ($\mathcal{H}_k$)
Estimate
$\widehat{F_k} = \mathcal{H}_k^T \theta_{k-1}$
Weights (θ_k)
−
+
Reference (F_k)
Error ($F_k - \widehat{F_k}$)
Weight Update Algorithm

Fig. 3. Illustration of the online recursive least square algorithm to update the analytical models of power consumption and execution time.

The Oracle cannot be used directly at runtime due to excessive (more than 1 MB) storage and computational requirements. Therefore, we need to construct a policy $\hat{\pi} : \mathcal{E} \rightarrow C^T$ that determines the best power management decisions for each epoch E_k. In other words, we want the policy $\hat{\pi}$ to mimic the behavior of the Oracle policy. To this end, we apply imitation learning to first construct an offline policy using the training data for all epochs $\mathcal{E} = \{E_1, E_2, \ldots, E_T\}$ and the corresponding labels for best configuration obtained from the Oracle policy. The features for each epoch are the performance counters and power consumption values listed in Table 2. We use a neural network to approximate the Oracle using this training data. Advantages of neural networks include the ability to learn complex decision functions and perform online updates via backpropagation. After learning a policy using exact imitation, we further improve its robustness and accuracy using state-of-the-art IL techniques [33] to closely approximate the behavior of Oracle policy. At the end of this design-time step, we obtain the offline policy $\pi_{offline}$ in the form of four functions to predict the configuration for knobs n_B, n_L, f_B, and f_L.

5.2 Online Execution and Power-Performance Modeling

At runtime, we start with the offline policy ($\pi_{offline}$) created at design time. Subsequently, we run the incoming applications using the most up to date policy obtained so far, as shown in Figure 2. Our online learning objective is to update the parameters of policy to minimize the overall energy consumption. Meeting this goal requires a reliable supervision, but constructing an Oracle similar to one presented in Section 5.1 is not practical at runtime. Therefore, we also collect power consumption and performance data listed in Table 2 and maintain accurate analytical power-performance models during the regular execution at runtime. Then, we use these models to achieve the online learning objective.

We begin with the power consumption and execution time models built offline with the help of the same applications used for the design-time Oracle construction. For example, the total power in kth epoch is expressed as:

$$\begin{aligned} P_{CPU} &= P_{dyn} + P_{leak} \\ &= \left(C_{dyn}(\mathbf{h_k}, \theta_\mathrm{k}) V^2 f \right) + V I_{leak}(\mathbf{h_k}, \theta_\mathrm{k}) \end{aligned} \tag{3}$$

where the dynamic switching capacitance C_{dyn} and leakage current I_{leak} are modeled as a function of the performance counters ($\mathbf{h_k}$) shown in Table 2. We employ a linear model using these counters and weight vector θ_k, as depicted in Figure 3. Note that our data also includes measured power consumption enabled by current sensors [19]. At the end of each epoch, we employ our model to estimate the power consumption in the previous interval. Subsequently, we compare it to the actual measurement, as illustrated in Figure 3 and Algorithm 2. We feed this error to the Recursive Least Square (RLS) [26] algorithm to update the parameters of the power consumption model.

ALGORITHM 2: Online power/performance modeling

```
Input: Features listed in Table 2 ($\mathbf{h_k}$), Measured power/performance ($F_k$)
for each epoch $E_k$ do
    Obtain weights at the end of the previous epoch ($\theta_{\mathbf{k-1}}$)
    Estimate power/performance through a linear model: $\widehat{F_k} = \mathbf{h_k^T}\theta_{\mathbf{k-1}}$
    Calculate the error between the measured value and the estimated value: $F_k - \widehat{F_k}$
    Apply weight update algorithm to obtain weights ($\theta_{\mathbf{k}}$) for the next epoch.
end
```

Similarly, we maintain an execution time model as a function of the performance counters. In this case, the reference is simply the length of the epoch, which can be tracked easily at runtime.

In summary, we maintain accurate power-performance models by using the data available at the end of each epoch. These models are used to generate strong supervision required by our online learning algorithm as described in the following section. We also note that updating the power and performance models have less than 0.1% overhead with respect to the execution time of an epoch.

5.3 Online Oracle Construction

In general, providing online supervision in terms of the best configuration to update the policy is a challenging problem. We overcome this challenge by leveraging the power and performance models, described in the previous section. At the end of each epoch (E_k), we have the metadata listed in Table 2 and up to date power-performance models. These models and the inputs enable us to evaluate the energy at candidate configurations in C other than the one chosen by the current policy. Specifically, we use the hardware counters measured on the current configuration to estimate the power and performance when using other configurations. In general, the hardware counters do not remain the same when we estimate the power consumption of other configurations. Indeed, predicting the change in the counters is a complex problem that involves modeling of system dynamics and the state transition probability. This can potentially add additional overhead to the Oracle estimation step of our approach. Therefore, to avoid additional overhead, we reuse the hardware counter values that we observe at runtime (of the current configuration) to estimate the power consumption of other configurations. After obtaining the energy consumption values at candidate configurations in C, we mark the configuration consuming the least energy as the optimal configuration (C_k^*).

We note that exhaustive search in the whole configuration space is resource expensive. For example, Samsung Exynos 5422 processor supports up to 4940 different configurations. Our implementation shows that evaluating each configuration can take up to 5% of the execution time of an epoch, which is prohibitive at runtime. To overcome this difficulty, we exploit the fact that the current policy ($\hat{\pi}$) is reasonably accurate and makes decisions that are not too far from those made by the Oracle policy. Therefore, instead of searching in the whole configuration space, we search within a local neighborhood of the configuration predicted by the current policy as described in Algorithm 3.

Resource-bounded Online Oracle Construction: We start the local search in the neighborhood of $\hat{C_k}$ by setting a maximum allowable time limit for the search, such as 5 μs. We initialize the optimal configuration C_k^* to the current policy decision ($\hat{C_k}$) and optimal cost (J_k^*) to the cost of $\hat{C_k}$ (line 3 in Algorithm 3). After this step, we start performing a search around the configuration chosen by the current policy. We first evaluate the metrics obtained by configurations that are

ALGORITHM 3: Online Estimation of Oracle Configuration

```
1  Input: Power and performance models; $\hat{C_k}$, predicted configuration; Timeout period
2  Output: Oracle configuration ($C_k^*$) at decision epoch $E_k$
3  Initialize: $C_k^* \leftarrow \hat{C_k}, C_{min} \leftarrow \hat{C_k}, J_k^* \leftarrow J(\hat{C_k})$
4  while !Timeout do
5      $\mathcal{N}(C) \leftarrow$ Configurations with control knob values one step away from $C_{min}$ (e.g., $n_B \pm 1$)
6      Evaluate cost of configurations in $\mathcal{N}(C)$
7      $C_{min} \leftarrow$ Configuration with minimum cost in $\mathcal{N}(C)$ // Greedy search step
8      $J_{min} \leftarrow J(C_{min})$
9      if ($J_{min} < J_k^*$) then
10         $J_k^* \leftarrow J_{min}$
11         $C_k^* \leftarrow C_{min}$
12     end
13 end
```

ALGORITHM 4: Policy Update via Online Training

```
Input: Current Policy $\pi$, Online Oracle $\pi^*$ via power-performance models
for each epoch $E_k$ do
    if $\pi(E_k)$ != $\pi^*(E_k)$ then
        Record hardware counters and the Oracle configuration $\pi^*(E_k)$ as a new training example in
            the buffer
    end
    if Buffer is full then
        Perform online training to update the parameters of policy (i.e., neural network) via
            backpropagation algorithm using the new training examples
        return the new policy $\pi_{new}$
    end
end
```

one step away from the policy decision. Specifically, we get configurations to be evaluated by changing the frequency of the cores by 200 MHz (i.e., ± 200 MHz) or changing the number of active cores by 1 (i.e., ± 1). After evaluating all the configurations that are one step away, we find the configuration with the minimum cost from $\mathcal{N}(C)$ as shown in Algorithm 3. We update the optimal configuration (C_k^*) whenever a configuration in the neighborhood search outperforms the best configuration found so far (line 10 in Algorithm 3). If the configuration chosen by the current policy is different than C_k^*, then we store C_k^* along with the features of the system state in a buffer, as illustrated in Figure 2. We continue the search in this way until the time limit is exceeded. The data in this buffer serve as the supervised training data from Oracle that is used for online learning, as detailed in the next section. This flexible methodology can be easily extended to different objectives. For example, we can perform a gradient-based search in the direction of increasing value if the objective function is to maximize performance.

5.4 Policy Update via Online Training

The final step in the proposed online-IL methodology is to incrementally update the parameters of the policy as a function of the new unseen applications and new training examples. The online training method to update the policy is shown in Algorithm 4. As described in the previous section,

Table 3. List of Applications Instrumented from Different Benchmark Suites

Set 1	Set 2
BML	AES
Dijkstra	Kmeans
FFT	Spectral
Patricia	Motion Estimation
Qsort	PCA
SHA	Blackscholes-2T
Blowfish	Blackscholes-4T
String Search	
ADPCM	

the Oracle policy (i.e., $\pi^*(E_k) = C_k^*$) is stored in a buffer. The size of this buffer is important, since it determines the training accuracy and implementation overhead. Our experimental evaluations show that metadata and label for 100 epochs easily provide close to 100% accuracy in adapting to new applications. Furthermore, the corresponding storage overhead is less than 20 KB. Therefore, we use a buffer to store 100 Oracle entries in our experiments. We aggregate the training examples until the buffer is full. Subsequently, we update the parameters of the policy, which is represented as a neural network using this training data and backpropagation algorithm [20]. Once the online training procedure to update the policy parameters is completed, we replace the current policy with the new policy π_{new} and continue the next iteration of online-IL approach.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Platform: We implement the proposed online IL algorithm on Odroid-XU3 [19] board running Ubuntu 15.04 OS. The board integrates a Samsung Exynos 5422 SoC that consists of a A15 (big) core cluster with 4 cores, a A7 (little) cluster with 4 cores, ARM Mali GPU and other components. To facilitate power measurements, the board also provides sensors to measure the power consumption of the big cores, little cores, main memory, and the GPU. We sample these sensors every 50 ms to record the power consumption of each component. These measurements are used in the online learning algorithm to update the power/performance models and to evaluate the performance of the proposed online learning algorithm. The constructed policies are implemented as user-space governors on the Odroid-XU3 board. Specifically, the frequency of operation is specified in a *sysfs* entry that is read by the kernel to set the frequency. The number of cores for the little and big cluster are set using the dynamic hotplugging feature of the Linux kernel.

Benchmarks: We evaluate the proposed methodology on a total of 16 applications from Mibench [18], Cortex [41], and PARSEC [5] benchmark suites. Applications from these benchmark suites exhibit a wide range of characteristics, such as compute intensiveness, memory intensiveness, and parallelism. Therefore, these applications represent a broad group of applications commonly executed in mobile platforms. We divide the benchmarks into two sets, as shown in Table 3. At any time, one of the sets is used for the offline learning phase while the other set is reserved for online learning.

Table 4. Different Parameters for Neural Network

Model Hyper-Parameters	No. of Hidden Layers	2
	No. of Neurons	20 in each layer
	Activation	ReLu
	Optimizer	Adam
	Learning Rate	0.001
	Loss Function	Categorical Cross-entropy
Training Parameters for Offline Learning	Batch Size	150
	Epochs	500
Training Parameters for Online Learning	Buffer Size	100
	Batch Size	20
	Epochs	20

Data Collection: Construction of an Oracle for offline policy requires characterization of the applications while running at different configurations. Therefore, we perform extensive data collection on the Odroid-XU3 platform while running the benchmarks. Specifically, we sweep the core configuration from *1 big–1 little* to *4 big–4 little*. Within each configuration, we change the frequency of big cores from 600 MHz to 2.0 GHz and little cores from 600 MHz to 1.4 GHz in steps of 200 MHz. We do not go below 600 MHz as lower frequencies do not provide better energy-efficiency [1]. This characterization data is used to construct the Oracle policies in the offline training phase.

6.2 Evaluation of the Policy Trained Offline

To obtain a baseline, we first design the initial offline policy using only the applications known at design-time. We employ IL to construct the policy using a neural network with two hidden layers. The parameters of the policy and the hyperparameters used in training are listed in Table 4. We adopt a neural network implementation over other regression techniques, such as linear regression and regression trees, since neural networks allow us to learn non-linear decision functions and facilitate online updates through the backpropagation algorithm [20]. We use a smaller number of training epochs and smaller batch size for online learning, because the policy updates at runtime are incremental with a relatively small number of training examples. We observe that a training buffer size of 100, and 20 epochs with a batch size of 20 are sufficient to update the policy.

We evaluate the accuracy of the policy for each configuration knob by measuring the distance between the policy decision and a golden reference, which is constructed to evaluate the proposed policies. If the total number of available levels for a configuration knob is L, the level chosen by the policy is L_π, and the reference level is L_{ref}, then accuracy can be expressed as:

$$\text{Accuracy}(\%) = 100 \times \left(1 - \frac{|L_\pi - L_{ref}|}{L - 1}\right). \tag{4}$$

After finding the accuracy of each epoch, we take the average across all epochs of an application to compute the overall accuracy for each control knob. We note that 100% accuracy denotes that the decisions of the underlying policy match with the golden reference, and no policy can perform better than that. We first train the policy offline using applications from the Mibench suite. Then, we evaluate the trained policy with applications from all three benchmark suites, as shown in Figure 4. Since the Mibench applications are included in the training set for this experiment, the

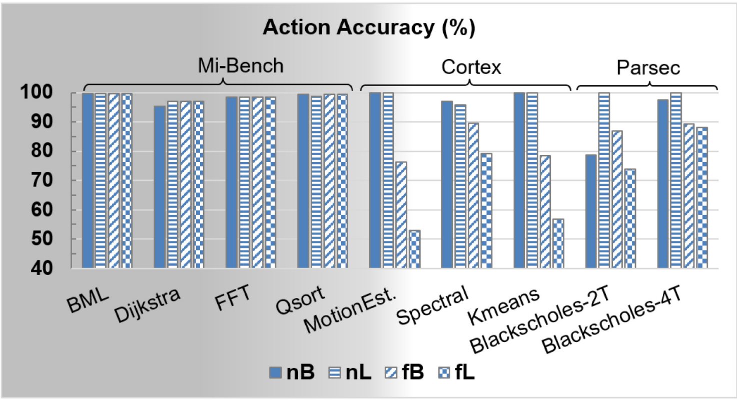

Fig. 4. Accuracy comparison for all knobs for applications from Mibench, Cortex, and PARSEC benchmark suites *when the policy is trained with only Mibench applications.*

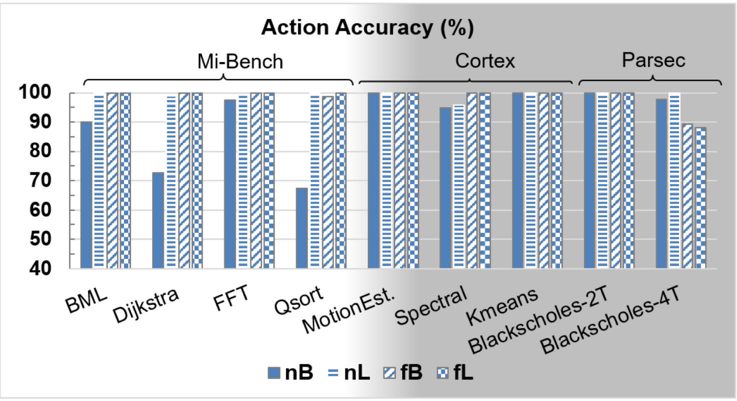

Fig. 5. Accuracy comparison for all knobs for applications from Mibench, Cortex, and PARSEC benchmark suites *when the policy is trained with Cortex and PARSEC applications.*

offline policy always reaches more than 95% accuracy compared to the Oracle for all four configurations knobs. However, the accuracy is significantly lower for the applications from the Cortex suite that are not included in the training set. For example, MotionEstimation shows only 77% and 53% accuracy for big and little core frequencies, respectively. Similarly, the accuracies for the Blackscholes application from the PARSEC suite are lower than 80% for the number of big cores and frequency of little cores. *This shows that the offline policy is not good enough to capture the characteristics of applications not seen at design-time.*

We also construct an offline policy by using only the applications from Cortex and PARSEC suites. This time the offline policy performs well when running applications from these two benchmark suites, but it shows poor performance for Mibench suite, as shown in Figure 5. More precisely, we observe above 95% accuracy for all knobs when running applications from Cortex and PARSEC suites. In contrast, the accuracy for applications from the Mibench suite are significantly lower. For example, Qsort has only 67.5% accuracy for number of big cores. *The degradation of accuracy with unseen applications demonstrates the need for online learning.*

Next, we compare the performance of IL and RL policies that are trained offline. Figure 6 shows the accuracy of the offline policies trained with IL and RL, respectively. In this case, we construct

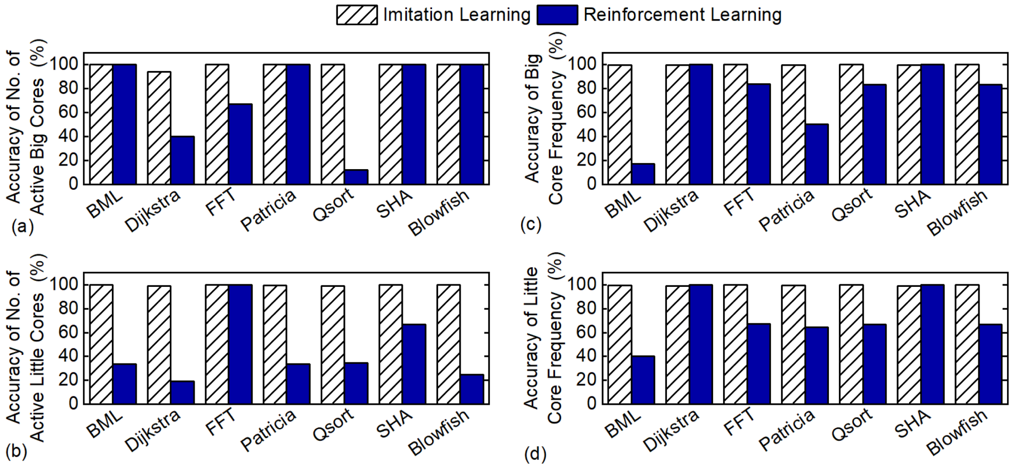

Fig. 6. Accuracy comparison *when the policy is trained with only Mibench applications (Set 1 in Table 3).*

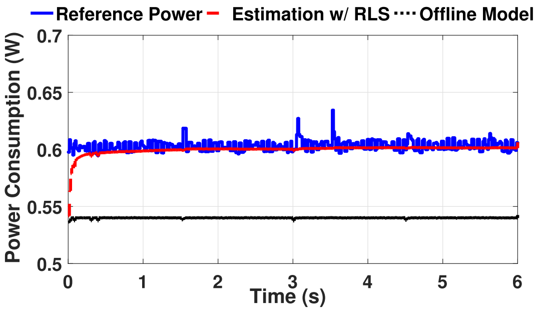

Fig. 7. Comparison of estimated power for Kmeans application. While estimating with RLS, the power estimation accuracy increases from 90% to 99% within 0.6 s.

two offline policies through IL and RL by considering all nine applications from Mibench suite as training set. Then, we test these policies individually on all applications in Mibench suite. We observe that the IL policy achieves close to 100% accuracy for all the control knobs for all applications. That means IL policy can achieve performance similar to that of the Oracle policy. However, the offline RL policy has much lower accuracy when compared to the IL policy. For example, number of big cores has 17% accuracy and number of little cores has 33% accuracy for Qsort application. This shows that even in the offline setting, the IL policy achieves a superior performance when compared to the RL policy. Moreover, the offline RL policy requires more than 10× time to converge than the offline IL policy.

6.3 Evaluation of Power-Performance Models

Similarly to the offline policy, we construct offline power and performance models using the applications available at design-time. These power and performance models may not be valid for new applications encountered only at runtime if they are not represented by the training data. For example, Figure 7 shows that power models constructed using Mibench applications cannot track the power consumption while running Kmeans application from the Cortex suite. In contrast, our online power models based on RLS converge to the measured power in about 200 ms, as depicted

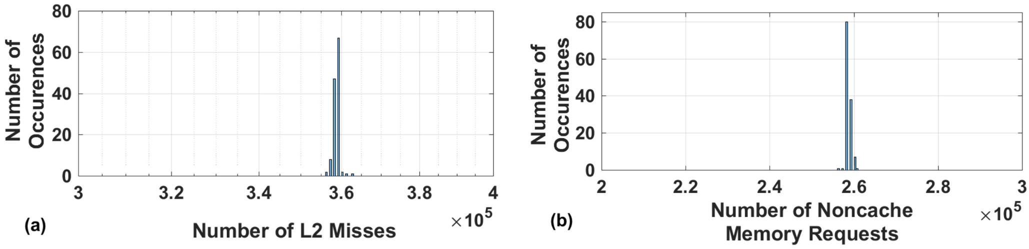

Fig. 8. Histogram showing (a) number of L2 misses and (b) number of noncache memory request for a particular epoch of Spectral application.

by the dashed line in Figure 7. Although the Kmeans application is not seen during training, we achieve less than 1% error compared to the reference value. Similarly, the performance models converge to reference values in 100–300 ms, which amounts to fewer than 10 epochs. This shows that the proposed technique adapts the offline model to unseen applications at runtime with high efficiency. Overall, the power models have an average error of 2.6%. These models enable us to construct the accurate runtime supervision required for adapting the offline policy. From our experiments, we observe that generating runtime supervision and updating the power and performance models at each epoch take approximately 8.5 μs. This is negligible compared to the execution time of an epoch that is in the order of 10 ms.

Accuracy of Power Estimation across Configurations: We use the performance counters measured at a given configuration to estimate the power consumption at other configurations. Figure 8(a) shows the histogram of L2 misses for different configurations for a particular epoch of Spectral application. The number of L2 misses for the applications we consider is in the range of 500–2.5×10^8. However, the number of L2 misses (for the particular epoch of Spectral application) of 122 configurations of a total of 128 configurations (96%) is between 3.57×10^5 and 3.59×10^5, i.e., within 0.6% of each other. Therefore, the number of L2 misses is nearly the same for all configurations for that particular epoch. The histogram for the number of non-cache memory requests is shown in Figure 8(b). The number of non-cache memory requests for the applications we consider is in the range of 20–7.6×10^7. However, the number of non-cache memory requests (for the particular epoch of Spectral application) of 98% of all configurations is between 2.58×10^5 and 2.6×10^5, i.e., within 0.8% of each other. We observe similar behavior for other hardware counters and for other applications, too. For this reason, we can safely take the hardware counters of the current configuration as the features to estimate the power of other configurations. Since we also consider the configuration itself as a feature to estimate the power/performance of the configuration, we obtain different power/performance estimation for different configurations. To validate the accuracy of this power estimation, we compared the estimated power of all the configurations with the actual power consumption of the configuration. That is, we use the hardware counters observed for a single configuration to estimate the power consumption of all the configurations and compared it with the measured value. For the applications used in online training, we observed that the estimation error is around 10%. This error is acceptable, since the online policy converges to the optimal for unseen applications.

6.4 Evaluation on Stream of Unseen Applications

In this section, we evaluate the effectiveness of the proposed online-IL methodology to new applications observed only at runtime. We start with a policy that is trained offline with applications from Mibench suite. Then, we run a random sequence of seven applications from the Cortex and

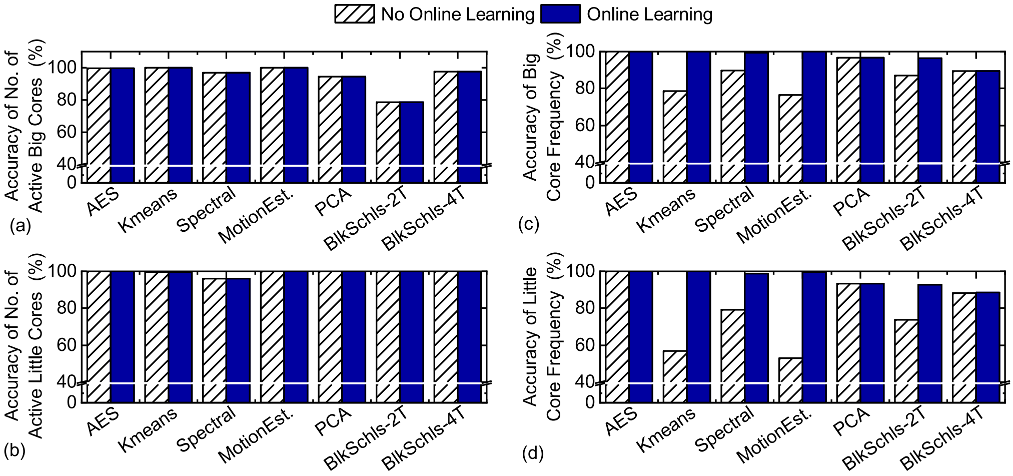

Fig. 9. Comparison of accuracy of (a) number of big cores, (b) number of little cores, (c) big core frequency, and (d) little core frequency for applications in Cortex and PARSEC benchmark suites.

PARSEC suites (AES, Kmeans, Spectral, Motion Estimation, PCA, 2-threaded Blackscholes, and 4-threaded Blackscholes) back to back. Each complete run of this set of applications takes around 33 seconds. We repeat this experiment for five different random sequences. After running the application sequences, we evaluate the effectiveness of the proposed approach on each application individually. To achieve this, we average the accuracy of each control knob over all random sequences. We note that here accuracy measures the distance of the policy decision to the golden reference (Equation (4)). Figure 9 compares the accuracy between the offline policy and the online policy after it converges for all four configurations for the random sequences. Figure 9(a) and Figure 9(b) show that our offline policy provides good accuracy for the number of cores. The online policy maintains this accuracy without providing significant improvement. In contrast, Figure 9(c) and Figure 9(d) show that the proposed online learning technique achieves a significant increase in the accuracy of little and big core frequencies. The biggest improvements (20–47%) are observed for Kmeans, Spectral, and MotionEstimation applications. In particular, the action accuracy of the little core frequency for MotionEstimation application increases from 53% to 99%.

We also analyze how quickly the online policy converges to the highest accuracy for different application sequences. As stated before, we randomly construct five different application sequences. To illustrate the convergence properties, we pick two representative application sequences. *Sequence 1* consists of Kmeans, AES, Spectral, MotionEstimation, PCA, BlackScholes-2T, and BlackScholes-4T applications one after another. Figure 10(a) and Figure 10(b) shows that the accuracy of big (f_B) and little (f_L) core frequencies start at 52% and 71%, respectively. Our online-IL technique increases both of these accuracies to 100% within 0.9 s, i.e., after running less than 3% of the sequence. Moreover, we also use the offline RL policy to perform online learning for this application sequence. To perform experiment with RL, we take the offline RL model trained with applications from Mibench suite as described in Section 6.2. From Figure 10(a) and Figure 10(b), it is observed that RL-based online policy does not achieve similar performance as Oracle policy most of the time. For example, the accuracy of choosing the frequency of big cores (f_B) and frequency of little cores (f_L) are 100% only 16% of the entire execution with RL-based online learned policy. Moreover, we observe that the online RL policy could not converge to optimal configuration even after the sequence is run 14 times back to back. In *Sequence 2*, we run Motion-

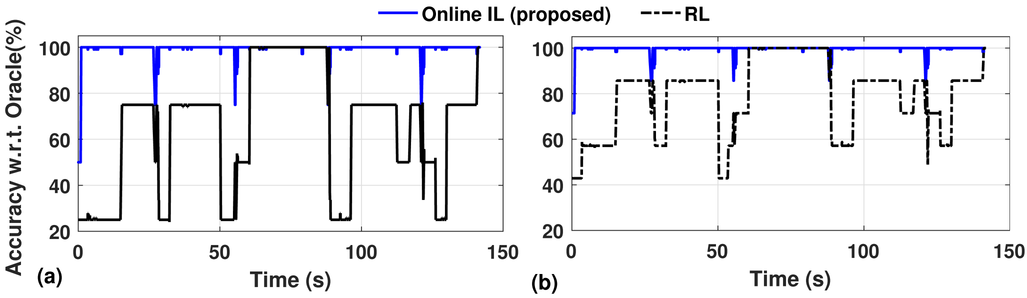

Fig. 10. Comparison between proposed online IL and reinforcement learning (RL) with application sequence 1 for (a) frequency of big cores (f_B) and (b) frequency of little cores (f_L).

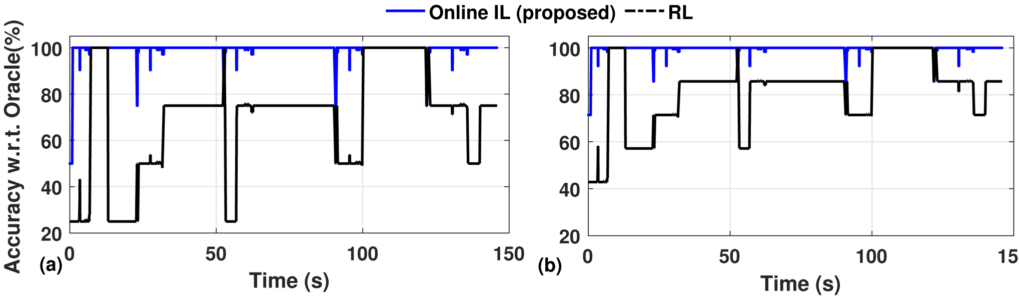

Fig. 11. Comparison between proposed online IL and reinforcement learning (RL) with application sequence 2 for (a) frequency of big cores (f_B) and (b) frequency of little cores (f_L).

Estimation, BlackScholes-4T, AES, BlackScholes-2T, Kmeans, Spectral, and PCA applications one after another. Similarly to the previous example, our initial policy converges to the optimal policy only 1.2 s, as shown in Figure 11(a) and Figure 11(b). RL-based online training is unable to converge as fast as online IL policy for this application sequence also. For both sequences, the initial offline IL model is retrained once before converging. However, the offline RL model is not able to obtain high accuracy ever after retraining five times. We also compare the energy consumption of the applications upon applying the RL policy and the proposed online IL policy. Figure 12 and Figure 13 show this energy comparison with respect to the Oracle for application sequence 1 and application sequence 2, respectively. For both sequences, we observe that the energy consumption with the proposed online IL policy is always within 2% of the energy consumption with Oracle policy. However, the energy consumption due to the RL policy is 10–40% higher than the energy consumption with the Oracle policy. This demonstrates the efficiency of our proposed online IL technique to adapt to new unseen applications.

Evaluation of the Proposed Approach without Offline Learning: We also evaluate the proposed online learning framework when a pre-trained offline model is not available. In this case, we start with a neural network model initialized with random weights. At runtime, with incoming applications, the model is updated using the proposed approach. Figure 14(a) and Figure 14(b) show how fast the online policy converges when the pre-trained offline model is not provided for frequency of big cores (f_B) and frequency of little cores (f_L) respectively. In this case also the online IL policy converges within 1.5 s of execution that is around 5% of the sequence. Similar to the results with offline policy, policy learned through RL does not obtain high accuracy when compared to the proposed online IL methodology. Figure 15(a) and Figure 15(b) show the

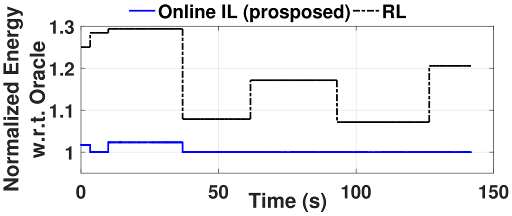

Fig. 12. Energy comparison with application sequence 1.

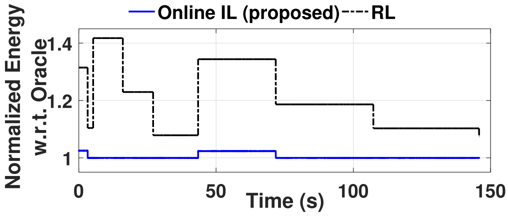

Fig. 13. Energy comparison with application sequence 2.

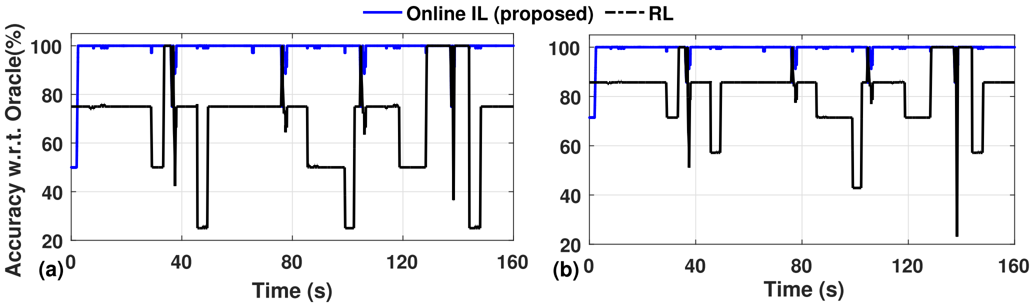

Fig. 14. Comparison between proposed online IL and reinforcement learning (RL) with application sequence 1 for (a) frequency of big cores (f_B) and (b) frequency of little cores (f_L). In this experiment, no pre-trained offline model is used.

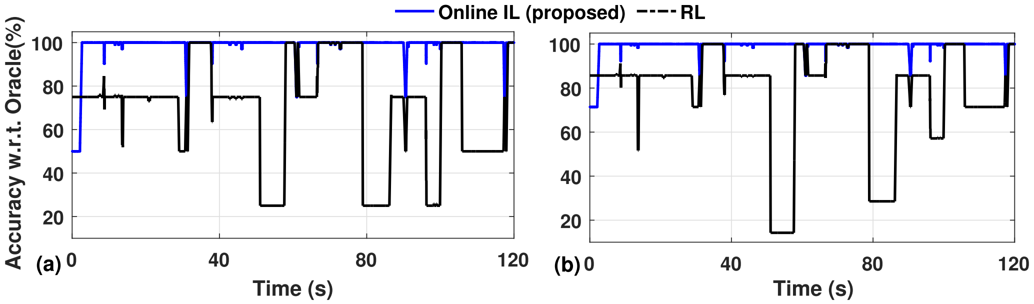

Fig. 15. Comparison between proposed online IL and reinforcement learning (RL) with application sequence 2 for (a) frequency of big cores (f_B) and (b) frequency of little cores (f_L). In this experiment, no pre-trained offline model is used.

convergence result with application sequence 2. In this case also the proposed online IL policy converges faster than RL policy. The RL policy is unable to achieve optimal performance even after the application sequence is run 5 times. Figure 16 and Figure 17 show the energy comparison of the application sequences with RL and proposed online IL method. In both cases, online IL achieves similar energy consumption as the reference Oracle policy. In contrast, energy consumption with RL policy is always 1.1×–1.6× higher than the energy consumption by the reference Oracle policy.

Performance of Online Learned Policy on Initial Applications: In online learning forgetting offline learned policy is a common issue [13]. To show that the proposed online IL methodology does not suffer from forgetting, we apply the online learned policy to the applications with which the offline training was performed. Figure 18 shows the accuracy of choosing action knobs for

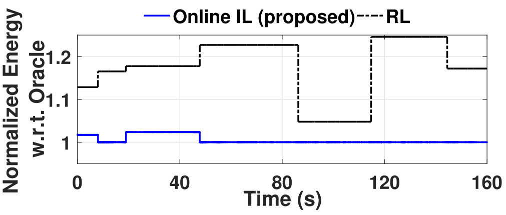

Fig. 16. Energy comparison with application sequence 1 without offline model.

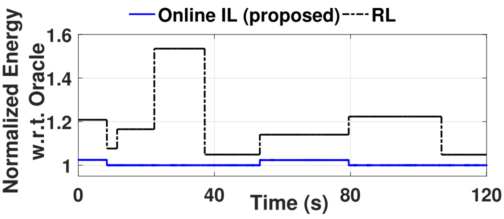

Fig. 17. Energy comparison with application sequence 2 without offline model.

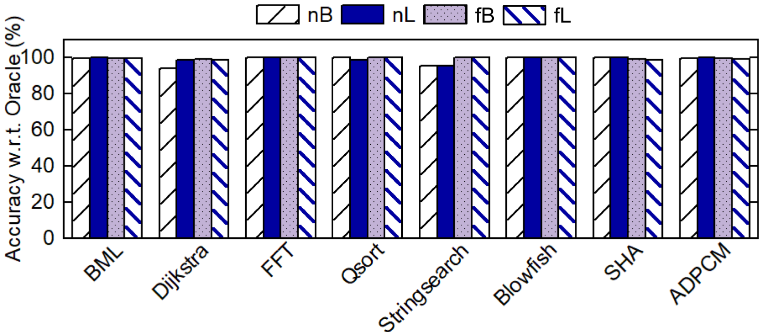

Fig. 18. Performance of online learned policy on initial applications.

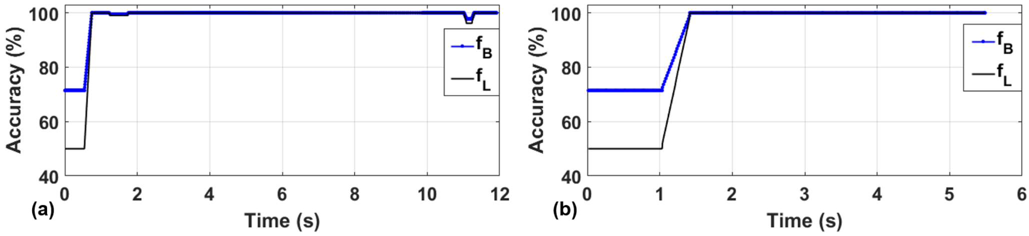

Fig. 19. Convergence of f_B and f_L for (a) Spectral and (b) Motion Estimation application.

Mibench applications with the policy obtained after online learning is performed on the applications from Cortex and Parsec benchmark suites. It can be seen that the accuracy is close to 100% for all applications and for all control knobs. This shows that the policy obtained after performing several rounds of online updates does not suffer from unwanted catastrophic forgetting.

6.5 Convergence for Single Application

Figure 9(c) and Figure 9(d) show that Kmeans, Spectral, and MotionEstimation applications have large improvements when we perform an online update of the policy. Therefore, we execute these applications standalone to measure the time required for each of the applications to converge to the optimal Oracle policy. From Figure 19(a), we see that to converge to optimal policy Spectral requires 0.7 s, which is 6% of the total application execution time. Also, we observe that within this time our online learning methodology aggregated 100 new samples that are then used to update the policy once. Similarly, Figure 19(b) shows that MotionEstimation requires 1.4 seconds to converge. In this case, 149 new samples are collected as training data to retrain the policy twice. Finally, we

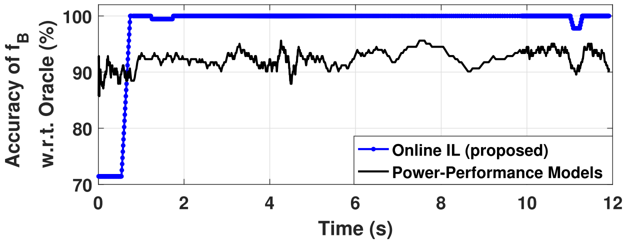

Fig. 20. Analysis of a Policy using only Power-Performance Models.

note that each online update of the policy incurs 2 ms of execution time. Since this is less than 0.5% of the time interval between successive updates, the runtime overhead is small.

Analysis of a Policy Using Only Power-Performance Models: We perform another experiment to specifically highlight the contribution of online-IL to provide an optimal policy. In this experiment, we do not use online-IL to estimate the configuration. Instead, we only use the power-performance models to estimate the configuration. Specifically, in the current epoch, we search around the configuration that was found optimal through the search performed in the previous epoch (same as in Algorithm 3). Figure 20 shows the accuracy comparison for frequency of big cores (f_B) between proposed online-IL technique and the case where we use only power-performance models. This experiment is performed for the Spectral application. We observe that, if only power-performance models are used, the initial accuracy is more than the proposed method. This happens because, initially, the IL policy is not suitable for the application. However, after learning on a sufficient number of samples, the IL policy reaches to the highest accuracy. This IL policy uses an online Oracle that is obtained through a local search around the current configuration. We note that, if there is no learned policy, the local search guided only by the power-performance models would not be effective. The online learned policy provides a good starting configuration to bootstrap the search process. In other words, only a small amount of search is sufficient to reach the optimal configuration if we perform local search around the configuration suggested by the learned policy. However, if only power-performance models are used to take the control decision, then local search will require evaluating a significantly large number of configurations (as much as 3× than online-IL) to construct an equally good online Oracle policy, which is not practically feasible. Therefore, if we perform local search guided only by the power-performance models, then the resulting policy never converges to the optimal as shown in Figure 20. Since local search requires a significantly large number of evaluations of configurations using power-performance models, it is not practical to use only power-performance models for online control. The combination of learned policy for getting good initial configuration and a small amount of local search guided by power-performance models to generate better configurations (supervision to improve policy) makes our overall approach effective. Once we converge to a near-optimal policy, we can apply the policy with very little search, but the baseline solution (search guided by power-performance models) will still require a lot of search and is impractical.

6.6 Comparison with the State-of-the-Art Power Management Techniques

Powersave governor is commonly used in mobile platforms to operate in low power mode. Therefore, we compare the energy consumption of individual applications when we use the proposed online-IL approach and the Powersave governor. We note that these comparisons include energy and execution time overheads, since the evaluations are based on hardware measurements. Similar to the previous sections, we start with an offline policy trained with applications from Mibench

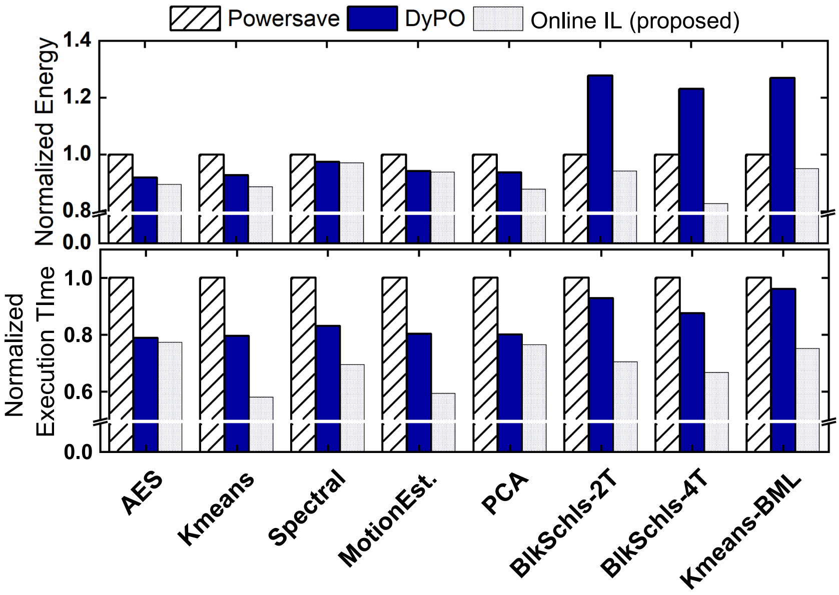

Fig. 21. Comparison of energy consumption and execution time with respect to the DyPO [14] and Powersave governor.

suite. Then, we use the proposed technique to update the initial policy as we run applications from Cortex and PARSEC suites.

Figure 21 shows the energy consumption and execution time comparisons. We normalize the energy and execution time of all applications with respect to the Powersave governor, since absolute numbers vary widely across different applications. We observe that the proposed online learning methodology achieves *both lower energy and execution time* compared to the Powersave governor for all applications. More specifically, we achieve on average 10% lower energy consumption and 24% faster execution time. In particular, we improve the execution time for Kmeans by 42% while also reducing the energy consumption.

The proposed policy achieves 48.4% less energy consumption on average when compared to the interactive governor. We omit the detailed comparison with the interactive governor, since it targets performance optimization. We also compared the proposed online learning methodology with a recent offline learning technique [14]. The DyPO approach first finds the Pareto-optimal configurations for energy optimal operation and then chooses a configuration among them at runtime. However, the DyPO approach does not perform any online learning. We observe that our proposed methodology on average achieves 6.39% improvement in energy consumption and 15.24% improvement in execution time for unseen applications when compared to the DyPO approach.

Furthermore, to validate the robustness of the proposed online IL technique, we apply the technique while executing Kmeans and BML applications concurrently (Kmeans-BML). We choose Kmeans as the foreground application and BML as the background application. We observe that the proposed online IL technique results in 5% less energy consumption than powersave governor for Kmeans-BML. Moreover, the execution time of Kmeans-BML with proposed online IL technique is 25% and 22% less than the powersave governor and DyPO, respectively. This experiment proves that the proposed online IL technique can efficiently manage the platform while executing multiple applications concurrently.

Table 5. Runtime and Implementation Overhead Summary

Metric	Module	Overhead	Time between successive calls	% Overhead
Exe. time	RLS	6 μs	10–20 ms	0.06–0.12
	Local search	2.42 μs	10–20 ms	0.02–0.05
	Backprop.	2 ms	0.5–1.5 s	0.4–1.0

In summary, the proposed online imitation learning methodology efficiently adapts to unseen applications and provides significant improvements over state-of-the-art methods with minimal overhead.

6.7 Implementation Overhead Analysis

We conclude our experimental evaluation by summarizing the runtime and storage overhead of our approach measured on Odroid XU3 board. The online power-performance models are updated in each epoch, which ranges between 10 and 20 ms. Since the model updates take 6 μs, the runtime overhead is only 0.06–0.12%, as summarized in Table 5. Online policy update via backpropagation takes longer (2 ms) than the other operations, as expected. However, we note that it runs every 100 epochs, i.e., in the order of seconds. Therefore, its runtime overhead is less than 1% as shown in Table 5. Finally, the storage overhead of the buffer to store new training examples is only 18 KB. In summary, the proposed approach incurs minimal runtime and storage overheads.

Overhead Comparison with RL: We start the overhead comparison with the overhead of retraining neural network with backpropagation algorithm. For each backpropagation step, we use 100 training samples for both online-IL and RL algorithms. Furthermore, the network structure used in both the algorithms is similar. Therefore, the execution time overhead for both algorithms for a single backpropagation step is same (2 ms). However, RL needs more backpropagation steps than the proposed online-IL to learn a policy that performs as good as online-IL. As a result, the cumulative overhead of RL in terms of both execution time and training samples required is much higher than online-IL. For an example, to obtain the optimal policy for Spectral application, online-IL performs backpropagation step only once (total 2 ms overhead). But, even after RL performs 8 backpropagation steps (total 16 ms overhead), it is not able to learn the optimal policy for Spectral. Overall, for Spectral application, the performance overhead incurred by RL is twice of the overhead incurred by IL. This further proves the efficiency of proposed online-IL approach over RL.

7 CONCLUSIONS AND FUTURE WORK

Designing an optimal runtime power management policy that achieves fast response time and minimum energy consumption is challenging due to the large space of available control knobs and diversity of applications. Policies constructed at design-time may fail to deliver optimal execution for applications encountered at runtime. This article presented a novel online imitation learning methodology to efficiently learn the optimal policy for new applications at runtime. Experimental evaluation on Odroid XU3 platform shows that we achieve both more than 10% lower energy consumption and 24% faster execution time with respect to the Powersave governor. The proposed approach is also applicable to other sequential decision-making problems in mobile systems, such as, prefetcher, cache replacement, branch prediction. However, extending the proposed IL framework to these domains is beyond the scope of this article. Therefore, it is left as future work.

REFERENCES

[1] Ali Aalsaud et al. 2016. Power–aware performance adaptation of concurrent applications in heterogeneous many-core systems. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 368–373.

[2] Frank Bellosa. 2000. The benefits of event: Driven energy accounting in power-sensitive systems. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. 37–42.

[3] Ganapati Bhat, Sumit K. Mandal, Ujjwal Gupta, and Umit Y. Ogras. 2018. Online learning for adaptive optimization of heterogeneous socs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 61.

[4] Ganapati Bhat, Gaurav Singla, Ali K. Unver, and Umit Y. Ogras. 2017. Algorithmic optimization of thermal and power management for heterogeneous mobile platforms. *IEEE Trans. VLSI Syst.* 26, 3 (2017), 544–557.

[5] Christian Bienia et al. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 72–81.

[6] David Brooks, Robert P. Dick, Russ Joseph, and Li Shang. 2007. Power, thermal, and reliability modeling in nanometer-scale microprocessors. *IEEE Micro* 27, 3 (2007), 49–62.

[7] Zhuo Chen and Diana Marculescu. 2015. Distributed reinforcement learning for power limited many-core system performance optimization. In *Proceedings of the Design Automation in Test in Europe Conference (DATE'15)*. 1521–1526.

[8] Ryan Cochran et al. [n.d.]. Pack & Cap: Adaptive DVFS and thread packing under power caps. In *Proceedings of the 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 175–185.

[9] Gaurav Dhiman et al. 2009. System-level power management using online learning. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 5 (2009), 676–689.

[10] Benedikt Dietrich and Samarjit Chakraborty. 2014. Lightweight graphics instrumentation for game state-specific power management in android. *Multimedia Syst.* 20, 5 (2014), 563–578.

[11] Benedikt Dietrich, Swaroop Nunna, Dip Goswami, Samarjit Chakraborty, and Matthias Gries. 2010. LMS-based low-complexity game workload prediction for DVFS. In *Proceedings of the 2010 IEEE International Conference on Computer Design*. 417–424.

[12] Yang Ge and Qinru Qiu. 2011. Dynamic thermal management for multimedia applications using machine learning. In *Proceedings of the 48th Design Automation Conference*. 95–100.

[13] Ian J. Goodfellow et al. 2013. An empirical investigation of catastrophic forgetting in gradient-based neural networks. In *Proceedings of International Conference on Learning Representations (ICLR'14)*.

[14] Ujjwal Gupta et al. 2017. DyPO: Dynamic pareto-optimal configuration selection for heterogeneous MpSoCs. *ACM Trans. Embedd. Comput. Syst.* 16, 5s (2017), 123.

[15] Ujjwal Gupta et al. 2019. A deep q-learning approach for dynamic management of heterogeneous processors. *IEEE Comput. Arch. Lett.* 18, 1 (2019), 14–17.

[16] Ujjwal Gupta, Manoj Babu, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, Suat Gumussoy, and Umit Y. Ogras. 2018. An online learning methodology for performance modeling of graphics processors. *IEEE Trans. Comput.* 67, 12 (2018), 1677–1691.

[17] Ujjwal Gupta, Manoj Babu, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, and Umit Y. Ogras. 2018. STAFF: Online learning with stabilized adaptive forgetting factor and feature selection algorithm. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[18] Matthew R. Guthaus et al. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*. 3–14.

[19] Hardkernel. 2014. ODROID-XU3. Retrieved November 24, 2018 https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3.

[20] Robert Hecht-Nielsen. 1992. Theory of the backpropagation neural network. In *Neural Networks for Perception*. Elsevier, 65–93.

[21] David Kadjo, Raid Ayoub, Michael Kishinevsky, and Paul V. Gratz. 2015. A control-theoretic approach for energy efficient CPU-GPU subsystem in mobile platforms. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 62.

[22] Ryan Gary Kim et al. 2017. Imitation learning for dynamic VFI control in large-scale manycore systems. *IEEE Trans. VLSI Syst.* 25, 9 (2017), 2458–2471.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. 75.

[24] Sumit K. Mandal et al. 2019. Dynamic resource management of heterogeneous mobile platforms via imitation learning. *IEEE Trans. VLSI Syst.* 27, 12 (2019), 2842–2854.

[25] Jose F. Martinez and Engin Ipek. 2009. Dynamic multicore resource management: A machine learning approach. *IEEE Micro* 29, 5 (2009).

[26] Jerry M. Mendel. 1995. *Lessons in Estimation Theory for Signal Processing, Communications, and Control.* Pearson Education.

[27] Volodymyr Mnih et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[28] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, Vol. 710.

[29] Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the Linux Symposium*, Vol. 2. 215–230.

[30] Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. 2017. ML-Gov: A machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming. In *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia.* 12–21.

[31] Anuj Pathania et al. 2015. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *Proceedings of the Design Automation Conference.* 201.

[32] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. 2014. Integrated CPU-GPU power management for 3D mobile games. In *Proceedings of the Design Automation Conference.* 1–6.

[33] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics.* 627–635.

[34] Stefan Schaal. 1999. Is imitation learning the route to humanoid robots? *Trends Cogn. Sci.* 3, 6 (1999), 233–242.

[35] Rishad A. Shafik, Sheng Yang, Anup Das, Luis A. Maeda-Nunez, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2015. Learning transfer-based adaptive energy minimization in embedded systems. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 6 (2015), 877–890.

[36] Amit Singh, Karunakar Reddy Basireddy, Alok Prakash, Geoff Merrett, and Bashir M. Al-Hashimi. 2020. Collaborative adaptation for energy-efficient heterogeneous mobile SoCs. *IEEE Trans. Comput.* 69, 2 (2020), 185–197.

[37] Statista. [n.d.]. Mobile App Usage—Statistics & Facts. Retrieved November 24, 2018 from https://www.statista.com/topics/1002/mobile-app-usage/.

[38] Hameedah Sultan, Gayathri Ananthanarayanan, and Smruti R. Sarangi. 2014. Processor power estimation techniques: A survey. *Int. J. High Perf. Syst. Arch.* 5, 2 (2014), 93–114.

[39] Wen Sun, Arun Venkatraman, Geoffrey J. Gordon, Byron Boots, and J. Andrew Bagnell. 2017. Deeply AggreVaTeD: Differentiable imitation learning for sequential prediction. In *Proceedings of the 34th International Conference Machine Learning*, Vol. 70. 3309–3318.

[40] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* MIT Press.

[41] Shelby Thomas et al. 2014. CortexSuite: A synthetic brain benchmark suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'14).* 76–79.

[42] Zhongyuan Tian, Zhe Wang, Haoran Li, Peng Yang, Rafael Kioji Vivas Maeda, and Jiang Xu. 2018. Multi-device collaborative management through knowledge sharing. In *Proceedings of the 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC'18).* 22–27.

[43] F. M. M. ul Islam and Man Lin. 2017. Hybrid DVFS scheduling for real-time systems based on reinforcement learning. *IEEE Syst. J.* 11, 2 (2017), 931–940.

[44] N. Vallina-Rodriguez and Jon Crowcroft. 2012. Energy management techniques in modern mobile handsets. *IEEE Comm. Surv. Tutor.* 15, 1 (2012), 1–20.

[45] Jae-Yeon Won et al. 2014. Up by their bootstraps: Online learning in artificial neural networks for CMP uncore power management. In *Proceedings of the 2014 IEEE 20th International Symposium on HPCA.* 308–319.

[46] Wu Ye, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. 2000. The design and use of simplepower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference.* ACM, 340–345.

[47] Qingchen Zhang et al. 2017. Energy-efficient scheduling for real-time systems based on deep q-learning model. *IEEE Trans. Sust. Comput.* 4, 1 (2017), 132–141.

[48] Qingchen Zhang, Man Lin, Laurence T. Yang, Zhikui Chen, Samee U. Khan, and Peng Li. 2018. A double deep Q-learning model for energy-efficient edge scheduling. *IEEE Trans. Serv. Comput.* (2018).

Received September 2019; revised February 2020; accepted February 2020