Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning

Sumit K. Mandal, *Student Member, IEEE*, Ganapati Bhat, *Member, IEEE*, Chetan Arvind Patil, Janardhan Rao Doppa, *Member, IEEE*, Partha Pratim Pande, *Senior Member, IEEE*, and Umit Y. Ogras, *Senior Member, IEEE*

***Abstract*—The complexity of heterogeneous mobile platforms is growing at a rate faster than our ability to manage them optimally at runtime. For example, state-of-the-art systems-on-chip (SoCs) enable controlling the type (Big/Little), number, and frequency of active cores. Managing these platforms becomes challenging with the increase in the type, number, and supported frequency levels of the cores. However, existing solutions used in mobile platforms still rely on simple heuristics based on the utilization of cores. This paper presents a novel and practical imitation learning (IL) framework for dynamically controlling the type (Big/Little), number, and the frequencies of active cores in heterogeneous mobile processors. We present efficient approaches for constructing an Oracle policy to optimize different objective functions, such as energy and performance per Watt (PPW). The Oracle policies enable us to design low-overhead power management policies that achieve near-optimal performance matching the Oracle. Experiments on a commercial platform with 19 benchmarks show on an average 101% PPW improvement compared to the default interactive governor.**

***Index Terms*—Heterogeneous computing, imitation learning (IL), multicore architectures, multi-processor systems-on-chip (SoCs).**

I. Introduction

THERE are over two million smartphone and tablet apps as of the third quarter of 2018 [1]. Designers of mobile platforms must deliver both competitive performance and energy efficiency while running these applications to remain in the market. Hence, mobile platforms are designed to provide peak performance for both computation and memory-intensive applications (e.g., games) and high energy efficiency for low-intensity workloads (e.g., email).

Modern heterogeneous mobile systems-on-chip (SoCs) support multiple types of cores (e.g., Big/Little) and different voltage/frequency levels for each core. The full potential of these architectures can be realized only if these knobs are managed optimally at runtime as a function of active applications. This is a challenging problem for two reasons. First, the joint space of control actions grows exponentially with the number of cores and their frequency levels. For example, Samsung Exynos 5422 SoC features four Little and four Big cores whose power states (on/off) and frequency levels can be controlled adaptively. Consequently, power management drivers need to choose 1 out of 4940 feasible solutions at every control interval, which typically ranges from 50 to 100 ms [2]. Second, it is impractical to perform application-specific optimization due to the exponential growth of potential applications. Therefore, there is a great need for runtime techniques that can manage mobile SoCs optimally for a wide variety of applications.

Despite the impressive progress in hardware support, the majority of existing solutions in commercial mobile platforms still rely on simple heuristic algorithms whose simplicity is hard to beat. For example, interactive and ondemand governors in mobile systems increase (decrease) the operating frequency by one level when the utilization exceeds (goes below) a static threshold. They are very aggressive in driving the frequency levels up since one of the main differentiating factors is still the performance perceived by the user, such as touch responsiveness or the maximum throughput while running key performance indicators (KPIs). Hence, this choice leads to running the applications at the highest frequencies during most of their active time, leaving little room for improving the performance. In contrast, the powersave governor uses the lowest frequency levels to minimize power consumption. However, this choice does not necessarily maximize energy efficiency [3]. Therefore, new solutions must not only maximize the energy efficiency but also need to have negligible runtime overhead to be practical.

In this paper, we present a novel and practical dynamic resource management technique for heterogeneous mobile platforms using the imitation learning (IL) framework [4]–[7]. The proposed technique controls the type (Big/Little), number, and the frequencies of active cores. IL is a promising approach

Manuscript received February 21, 2019; revised May 21, 2019; accepted June 14, 2019. Date of publication July 23, 2019; date of current version November 22, 2019. This work was supported in part by the National Science Foundation under Grant CNS-1526562, Grant OAC-1910213, and Grant IIS-1845922, in part by the Semiconductor Research Corporation (SRC) under Grant 2721.001, and in part by the U.S. Army Research Office under Grant W911NF-17-1-0485 and Grant W911NF-19-1-0162. *(Corresponding author: Sumit K. Mandal.)*

S. K. Mandal, G. Bhat, and U. Y. Ogras are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: skmandal@asu.edu; gmbhat@asu.edu; umit@asu.edu).

C. A. Patil is with the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: chetanpatil@asu.edu).

J. R. Doppa and P. P. Pande are with the School of Electrical Engineering and Computer Science (EECS), Washington State University, Pullman, WA 99164 USA (e-mail: jana.doppa@wsu.edu; pande@wsu.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2926106

since it enables us to automatically transform an optimal offline solution into an efficient online policy. More precisely, we first construct a near-optimal Oracle policy *offline* by exploiting characterization data for target applications available at design time. Then, we employ advanced IL algorithms to design low-overhead control policies that imitate the Oracle policy and generalize beyond the applications used for training, as summarized below.

A. Data Collection and Oracle Design

We instrument a set of benchmark applications, known to the designers, such that we can collect performance counters for a block of macroblocks [8], referred to as *snippets*. Then, we run the instrumented applications on a big.LITTLE platform [9] to measure the power consumption and collect performance counters, such as the number of instructions and active cycles, for each snippet. Finally, we construct the Oracle policy using the data collected at different core configurations. For a given snippet, the Oracle policy specifies the optimum configuration (i.e., the control actions for all control knobs) that optimizes a specified objective. In this paper, we present methodologies to generate three specific Oracle policies with the following objectives: 1) minimum energy consumption; 2) minimum energy consumption with a constraint on the execution time; and 3) maximum performance per Watt (PPW).

B. Online Policy Using IL

Given the Oracle policy, we can leverage off-the-shelf classification and regression algorithms to design low-cost online policies that can be stored in the form of parametric or nonparametric functions. In this paper, we employ linear regression (LR) and regression tree (RT) algorithms to generate policies with minimal storage and runtime decision-making overheads. As we discuss in Section II, the proposed IL framework has significantly lower storage, training time, and data requirements when compared to techniques based on reinforcement learning (RL) [10].

C. Contributions

The main contributions of this paper include.

1) Construction of an Oracle policy using benchmark applications that aid the design of the control policy with the desired behavior, such as minimizing the energy consumption under execution time constraints and maximizing the PPW.
2) An IL methodology based on the Oracle policy to automatically generate practical power management governors that generalize to unseen applications.
3) Empirical performance and implementation overhead evaluations of the proposed IL-based policies and comparison to existing governors on a big.LITTLE mobile platform [9].

II. Related Work

Advances in heterogeneous processors have enabled widespread adoption of mobile platforms. These platforms integrate multiple types of cores (Big/Little), graphics processing units, and other accelerators to support a wide variety of applications [11], [12]. The heterogeneity in mobile platforms requires new techniques for their resource management at runtime [13]–[18]. Default governors on mobile platforms, such as interactive and ondemand governors [19], and algorithms in [16] and [17] take management decisions based on the system utilization. However, decisions made with utilization alone may not be optimal in terms of PPW or energy consumption. Therefore, recent research has focused on new algorithms for runtime management of mobile platforms [3], [20]–[23]. However, [20]–[23] do not use phase-level instrumentation, which provides fine-grained information to take optimal power management decisions. The technique in [21] only chooses the type of core (Big/Little) on which the task has to be executed. In contrast, in our proposed power management technique, we control four configuration knobs, namely, the number of active big cores, number of active little cores and their corresponding frequencies. Controlling more knobs allows us to explore more number of decisions. Furthermore, the work presented in [20] uses application-specific policies, which are not scalable for unseen applications. A recent study uses a gradient search approach to decide the operating configuration of the platform such that temperature violations are minimized [23]. Specifically, it reduces the frequencies iteratively until the temperature constraint is met. Therefore, it takes a few iterations of decision making for the approach to achieve its objective. In contrast, the proposed approach takes the decision at each snippet using the learned machine learning models. The technique proposed in [3] considers phase-level instrumentation. It first characterizes the platform with a fixed set of configurations for different applications. These configurations are then pruned using clustering. Therefore, the runtime control algorithm chooses among a smaller set of configurations of control knobs. In contrast, we consider phase-level decisions for applications in our IL-based approach and create a generic policy that performs well on a wide range of applications. Also, the proposed approach considers a much larger set of configurations. Specifically, at runtime, the policy is not limited to the configurations it can choose to operate.

In recent work, machine learning techniques are employed for dynamic resource management in mobile platforms [24], [25]. For example, the work presented in [24] builds offline decision trees for choosing the frequency of CPU and GPU. These models are then used at runtime to assign CPU and GPU frequencies for gaming applications. However, they do not consider the problem of choosing the number of active CPU cores. More recently, RL algorithms based on Q-table approach have been proposed for dynamic voltage and frequency selection [26], [27]. However, prior RL-based solutions have several drawbacks. First, they are not practical as the size of the Q-table grows exponentially with the state and action space, as we demonstrate in our experimental results section. Thus, they require huge storage, which is limited to mobile platforms. Second, the performance of RL methods depends critically on the design of a good reward function to drive the learning process, which is a nontrivial task [28], [29]. Finally, the overall learning process to create near-optimal policies is very slow due to the need to try different actions

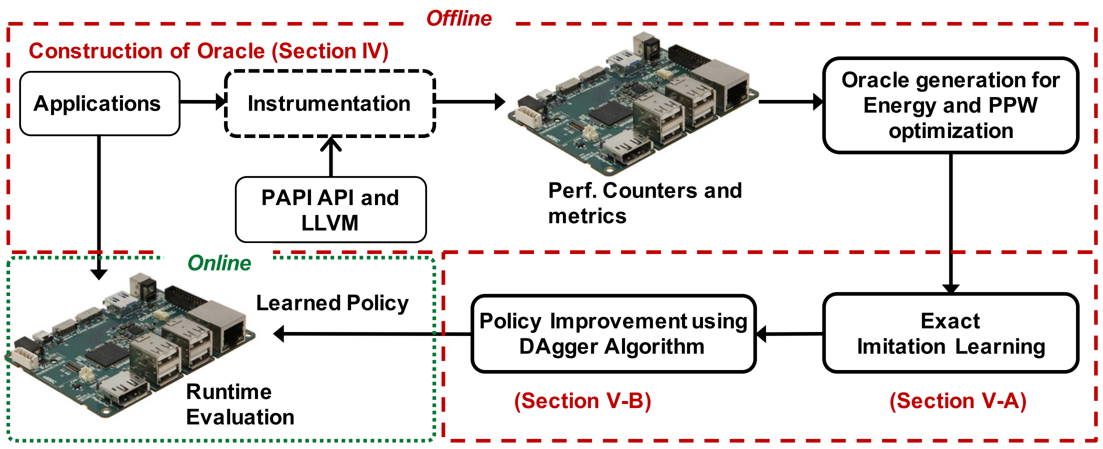
Offline
Construction of Oracle (Section IV)
Applications
Instrumentation
PAPI API and LLVM
Perf. Counters and metrics
Oracle generation for Energy and PPW optimization
Online
Learned Policy
Runtime Evaluation
Policy Improvement using DAgger Algorithm
Exact Imitation Learning
(Section V-B)
(Section V-A)

Fig. 1. Overview of the proposed dynamic resource management methodology.

at each state (i.e., exploration) to discover the best decision. An IL-based power management technique has been proposed for homogeneous manycore chips [7]. However, the proposed method is *not* applicable to heterogeneous mobile processors with different types of cores. Moreover, it is evaluated in a simulation setup for a specific suite of applications. In strong contrast to prior studies, we design *practical and global* policies that generalize to a large class of applications. We also present an extensive experimental evaluation on a heterogeneous mobile hardware platform.

III. Problem Setup

The state-of-the-art mobile platforms control the number of active cores and their frequencies at runtime periodically with 50–100 ms intervals as the workloads change dynamically [19]. Let us consider a heterogeneous mobile platform with n cores of k types. Without loss of generality, let us assume that we can control the platform at runtime by specifying the number of active cores for different types $(n_1, n_2, \ldots, n_k)$ and their corresponding frequencies $(f_1, f_2, \ldots, f_k)$. The set of supported configurations $\mathcal{C} = \{C_1, C_2, \ldots, C_M\}$ specify the number of active cores and their frequencies. For example, the hardware platform used in our experiments has four Big (B) and four Little (L) cores. Thus, each configuration C_i for $1 \le i \le M$ in this platform is a four-tuple $(n_{iB}, n_{iL}, f_{iB}, f_{iL})$ denoting the number of active Big cores, number of active Little cores, frequency of Big cores, and frequency of Little cores, respectively.

A given policy π maps the current system state to a candidate control configuration $C_i \in \mathcal{C}$ at each control epoch. We are provided with a set of target applications $\mathcal{T}$. Our goal is to create the best policy that maximizes the specified objective (e.g., PPW) on the given target applications and that also generalizes to new unseen applications. We develop an IL-based methodology to solve this problem, as outlined in Fig. 1. We first present an efficient mechanism to construct an Oracle policy π^* that drives our overall learning process presented in Section IV. Then, Section V describes the details of our IL methodology to create a policy that is very close to π^* in terms of its control behavior.

IV. Construction of Oracle Policy

The proposed mechanism for constructing the Oracle policy π^* consists of three steps. We first instrument the target applications $\mathcal{T}$ to enable workload conserving data collection. Then, we collect extensive power consumption and performance data on the target hardware platform. Finally, we construct the Oracle policy by taking the performance and power consumption overhead of hardware context switching into account. As this overhead is measured experimentally, other kinds of overheads such as thread migration, memory contention, etc., have been taken into account.

A. Instrumentation

We divide each target application from $\mathcal{T}$ into snippets, which can be considered as microbenchmarks. Each snippet is a collection of 10–100 millions of instructions that constitute a coherent subset of blocks [8] of the source code (e.g., a for loop). Consequently, a given target application can be represented as a sequence of snippets $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$, where N is the total number of snippets for that application, as summarized in Table II. We perform this process by automatically inserting performance application programming interface (PAPI) API [30] calls between snippets using low level virtual machine (LLVM) and Clang compiler infrastructure [3]. This instrumentation in user space allows us to collect the power consumption and performance counters shown in Table I for each snippet.

The features in Table I consist of six hardware performance counters and platform-level features such as utilization and power consumption. The maximum number of hardware counters that can be monitored is limited by the target platform to six counters. Adding more counters to the profiling leads to a multiplexing of the available performance monitoring unit (PMU) resources. For example, let us assume a sampling period of 50 ms for the performance counters. If we profile only six counters, then they will be measured for the entire 50-ms duration. However, if we choose to profile 12 counters, each counter will be measured for only 25 ms in total. Then, the counter values need to be extrapolated to obtain the corresponding counter values for the 50-ms duration. This extrapolation leads to inaccuracies in the counters. Therefore,

TABLE I
FEATURES EMPLOYED FOR LEARNING THE CONTROL POLICY

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	

TABLE II
LIST OF MAJOR PARAMETERS

Symbol	Description
N	Number of snippets in an application
M	Number of supported configurations on the platform
E_k	Total energy consumed till k^{th} snippet
$P_k(C_i)$	Power consumption of snippet S_k on configuration C_i
$t_k(C_i)$	Execution time of snippet S_k on configuration C_i
$\pi(S_k)$	Policy for choosing a configuration to run S_k, $1 \le k \le N$
$\pi^*(S_k)$	Optimal policy for snippet S_k, $1 \le k \le N$

we choose to profile six counters in the proposed work. Given the limited number of counters we can profile at runtime, we chose to enable counters that give the most representative information about a workload. The second set of features consists of the total little cluster utilization, per core big cluster utilization, and the total power consumption. We choose these system-level features since they are commonly used in default governors and other approaches in the literature [24], [31].

B. Data Collection

Our workload conserving PAPI instrumentation described earlier ensures that we can compare the performance achieved by each candidate configuration under an identical workload. To collect the data needed to construct the Oracle, we run each snippet of the target application in each supported configuration.

After collecting the characterization data, we compute a range of metrics, such as PPW and energy, that can be used as the cost function for optimization. For example, $J_{i,k}$ ($1 \le i \le M$ and $1 \le k \le N$) gives the value of the cost function when the snippet S_k is executed with configuration C_i. Therefore, we can select the optimum configuration for running this snippet with respect to an arbitrary cost function as described subsequently.

C. Oracle Policy for Minimizing the Energy Consumption

In this section, we present a methodology to generate the Oracle whose objective is to minimize energy consumption while executing the entire application. Since the energy consumption across snippet is additive, we can apply dynamic programming (DP) to generate the optimal policy that minimizes the total energy consumption. For a policy π that gives the sequence of configurations $\pi(S_1), \pi(S_2), \ldots, \pi(S_N)$ in snippets $S_1, S_2, \ldots, S_N$, total energy consumption E_k till kth snippet is given as

$$\begin{aligned} E_k = \sum_{j=1}^{k-1} [& P_j(\pi(S_j)) t_j(\pi(S_j)) \\ & + P_{\text{sw}}(\pi(S_j), \pi(S_{j+1})) t_{\text{sw}}(\pi(S_j), \pi(S_{j+1})) \\ & + P_k(\pi(S_k)) t_k(\pi(S_k))] \end{aligned} \tag{1}$$

where $P_k(\pi(S_k))$ is the power consumption and $t_k(\pi(S_k))$ is the execution time of the snippet S_k following the policy π. $\pi(S_{k-1})$ is the configuration of the previous snippet, $\pi(S_k)$ is the configuration of the current snippet, P_{sw} is the switching overhead function for power, and t_{sw} is the switching overhead function for execution time. Note that the overhead is zero if $\pi(S_{k-1}) = \pi(S_k)$. Furthermore, it depends only on the previous and current configurations, i.e., only one-step memory is needed. Using this notation, the minimum energy consumed by the application across all possible combination of configurations is denoted by E^*. We use DP to compute the minimum energy policy for an application by computing the optimal energy consumption of the partial application when starting at each snippet and configuration. This optimal energy consumption of starting at any snippet S_k with configuration c_i can be expressed as

$$E_N^*(C_i) = P_N(C_i) t_N(C_i), \quad C_i \in \mathcal{C} \tag{2}$$

and

$$\begin{aligned} E_k^*(C_i) = \min_{C_{\text{next}} \in \mathcal{C}} \{ & P_k(C_i) t_k(C_i) \\ & + P_{\text{sw}}(C_i, C_{\text{next}}) t_{\text{sw}}(C_i, C_{\text{next}}) \\ & + E_{k+1}^*(C_{\text{next}}) \}, \\ & C_i \in \mathcal{C}, k = 1, 2, \ldots, N-1 \end{aligned} \tag{3}$$

where C_{next} is the configuration in the next snippet. Equation (2) defines the optimal energy consumed by Nth snippet at different configurations C_i. To obtain the optimal energy consumed for a snippet other than N with a particular configuration, we have to take the minimum of energy consumed given the current snippet and the current configuration with each configuration in the next snippet. Specifically, there are three expressions which are being added inside minimization in (3). The first expression denotes the energy consumption of the current snippet. The second expression signifies the energy overhead to migrate from the current configuration to each configuration in the next snippet. The third expression evaluates the optimal energy of the next snippet with each configuration. Once the functions $E_k^*(c_i)$ are obtained, we can construct the optimal sequence of configurations for a given starting configuration by recursively applying the DP equation as

$$\begin{aligned} \pi^*(S_{k+1}) = \operatorname*{argmin}_{C_{\text{next}} \in \mathcal{C}} [& P_k(\pi^*(S_k)) t_k(\pi^*(S_k)) \\ & + P_{\text{sw}}(\pi^*(S_k), C_{\text{next}}) t_{\text{sw}}(\pi^*(S_{k+1}), C_{\text{next}}) \\ & + E_{k+1}^*(C_{\text{next}}), \quad k = 2, 3, \ldots, N. \end{aligned} \tag{4}$$

We use (4) to obtain the optimal policy that minimizes the energy consumption until the next snippet by considering the power management overheads. Then, we use the Oracle constructed using this methodology to train the IL policy for minimizing energy consumption.

D. Oracle Policy for Minimizing the Energy Consumption Under Execution Time Constraint

Energy minimization alone can hurt the performance unless there is a constraint on the execution time. Therefore, the second Oracle policy chooses the optimal configuration sequence that minimizes the total energy consumption under an execution time constraint. This problem can be formulated as a shortest path problem with travel time constraints where the energy consumption of snippets is equivalent to stage distances and the travel time is equivalent to the execution time of a snippet. This problem has been shown to be NP-Complete [32]. An optimal solution to the problem can be obtained by performing an exhaustive search over all snippets and configurations, but this approach is not tractable. Therefore, we use our energy-optimal DP solution to obtain a suboptimal Oracle. More specifically, we first run the whole application with all available configurations and record total execution time for all configurations. Subsequently, we discard the configurations that do not meet the given execution time constraint. The execution time constraint needs to be relaxed if none of the configurations satisfy it. After that, we apply DP in this reduced configuration space to find the configuration of each snippet which minimizes the total energy consumption by the application.

E. Oracle Policy for Maximizing the PPW

Our third policy chooses the configurations that maximize the PPW. Let I_k be the number of instructions in snippet S_k. The PPW achieved at runtime also depends on whether the hardware configuration changes before executing S_k. For example, if the current snippet runs only on the Little cores (C_i) and the next configuration (C_{next}) that involves one or more Big cores, then there is a context switching overhead. Hence, we can express the instructions per second IPS_{ik} as

$$IPS_{ik} = \frac{I_k}{t_k(C_i) + t_{\text{sw}}(C_i, C_{\text{next}})} \tag{5}$$

where $t_{\text{sw}}(C_i, C_{\text{next}})$ denotes the performance overhead of switching the hardware configuration. Similarly, we can express our objective function J_{ik}, i.e., the PPW for running the snippet S_k with configuration C_i as the ratio of instructions per second and the power consumption

$$J_{ik} = \frac{I_k}{P_k(C_i)t_k(C_i) + P_{\text{sw}}(C_i, C_{\text{next}})t_{\text{sw}}(C_i, C_{\text{next}})} \tag{6}$$

where $P_{\text{sw}}(C_i, C_{\text{next}})$ represents the power consumption overhead of switching the hardware configuration.

In this particular type of Oracle construction, our goal is to compute the policy that maximizes PPW defined in (6) for the entire application. Since all the quantities in (6) are obtained during the data collection phase, we construct Table III iteratively by starting with S_1 and an initial configuration. In this table, each sequence of snippet-configuration pairs, such as the one illustrated by the dashed line, corresponds to a candidate policy. The inherent structure in this problem enables a greedy algorithm for constructing Oracle policy. As PPW is not an additive function of snippets and configurations, DP is not an option to obtain the Oracle which maximizes PPW. Therefore, we sequentially select the best configuration at each snippet that maximizes the objective function for the snippets seen so far (values in the corresponding column). As the denominator in (6) is the energy consumption in that particular snippet, the overall objective function at any snippet $k > 1$ for a given trajectory of configurations up to snippet k can be written as

TABLE III
ORACLE CONSTRUCTION USING THE COLLECTED DATA

	Application Snippets					
Config.	S_1	S_2	S_3	S_4		S_N
C_1	J_{11}	J_{12}	J_{13}	$\boldsymbol{J_{14}^*}$	...	J_{1N}
C_2	J_{21}	$\boldsymbol{J_{22}^*}$	$\boldsymbol{J_{23}^*}$	J_{24}	...	J_{2N}
C_3	$\boldsymbol{J_{31}^*}$	J_{32}	J_{33}	J_{34}	...	$\boldsymbol{J_{3N}^*}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
C_M	J_{M1}	J_{M2}	J_{M3}	J_{M4}		J_{MN}

$$J_k = \frac{\sum_{j=1}^{k} I_j}{E_k}. \tag{7}$$

Equation (7) takes the total number of instructions until the current snippet as the numerator and the total energy consumed until the current snippet as the denominator. Thus, we obtain the overall PPW until the current snippet for different configurations. We use the overall PPW as the objective function and choose the Oracle configuration for the $(k+1)$th snippet as

$$C_{k+1}^* = \underset{C_{k+1} \in \mathcal{C}}{\operatorname{argmax}}\, J_{k+1}, \quad k = 1, 2, \ldots, N-1 \tag{8}$$

where J_{k+1} for each configuration in $\mathcal{C}$ is calculated using (7). In Table III, the selected sequence of configurations for each snippet is marked with an asterisk and bold typeface (e.g., $\boldsymbol{J_{31}^*}$ for snippet S_1). This sequence corresponds to the Oracle policy π^* that is used to design the desired control policy using IL algorithms.

V. IMITATION LEARNING METHODOLOGY

This section describes our IL methodology to design a low-overhead policy using the Oracle π^* described in Section IV.

Learning Problem: We are provided with an Oracle policy π^* that provides the supervision for best configuration C^* at each snippet in the training set S_k in the form of a lookup table. Our goal is to learn a compact representation of a policy π (e.g., linear or nonlinear function in terms of features for a given snippet) that closely imitates the behavior of the Oracle policy π^* to maximize the PPW.

Key Challenges: The main challenges include the following.

1) We have a sequential decision-making problem, where configuration selection at each snippet is not independent. In short, the learning problem is harder than traditional classification/regression tasks as the input examples are not drawn from an independent and identically distributed (IID) distribution.

2) The learned policy should have very little storage and computational overhead for decision-making due to the resource-constrained nature of mobile platforms. Our IL methodology addresses all the above challenges.

A. Exact Imitation Approach

The most basic approach for IL is the exact imitation. In exact imitation, we execute the Oracle policy π^* at every snippet for each target application and collect supervised learning examples (classification or regression) using the configurations from Oracle. The aggregate set of training examples are given to an IID classification/regression learning algorithm to induce the policy π.

In our target platform, each configuration selected from Oracle policy π^* is a four-tuple $(a_1^*, a_2^*, a_3^*, a_4^*)$ corresponding to the number of active Big cores, number of active Little cores, frequency of Big cores, and frequency of Little cores. We represent the state of each snippet S_k using a set of informative features $\Phi(S_k)$, as shown in Table I, that will be useful for making proper control decisions. We decompose the policy π into four components $(\pi_1, \pi_2, \pi_3, \pi_4)$, where each control decision a_i is made by policy π_i using features $\Phi(S_k)$ from a given snippet S_k. For each target application in $\mathcal{T}$ and for each snippet S_k, we collect one supervised learning example for each policy π_i, where the input example corresponds to features $\Phi(S_k)$ and output label corresponds to the control decision a_i^* from the Oracle policy. We give all the aggregate training examples for each policy π_i to a regression learning algorithm (e.g., LR or RT) to create the policy π.

Unfortunately, the policies learned using exact imitation approach suffer from error propagation. That is, errors made in configuration selection at one snippet will contribute to more errors at the subsequent snippets. The main reason for this behavior is as follows: the learned policy is trained on supervised examples from snippets where all the previous control decisions were made by Oracle policy π^*, which is not true in practice. To overcome the challenges of error propagation to learn a robust control policy, we consider an advanced IL algorithm called DAgger [5].

B. Data Aggregation (DAgger) Algorithm

The key idea behind the DAgger algorithm is to collect additional training data from Oracle policy π^* to be able to learn to recover from mistakes. Fig. 2 shows an overview of the iterative DAgger algorithm. In each iteration, we execute the current control policy π_{current} at each snippet S_k for every target application. Subsequently, we collect one supervised training example for each mistake at snippet S_k using the Oracle policy π^*. The aggregate sets of training examples over all the iterations (includes the first iteration that corresponds to exact imitation) for each component of the policy is given to a regression learning algorithm to create the new policy π_{new}. DAgger comes with strong theoretical guarantees for learning a policy that is very close to Oracle policy π^* in a small number of iterations [5]. We select the best control policy across all the iterations using cross-validation experiments. In summary, we are learning the optimal control actions as a function of the system state captured by the universal features given in Table I. Furthermore, the DAgger iterations allow the policy to see diverse types of system states, which are representative of a broader set of potentially unseen applications. Indeed, experimental evaluations using leave-one-out cross validation show that our global policy generalizes extremely well to unseen applications.

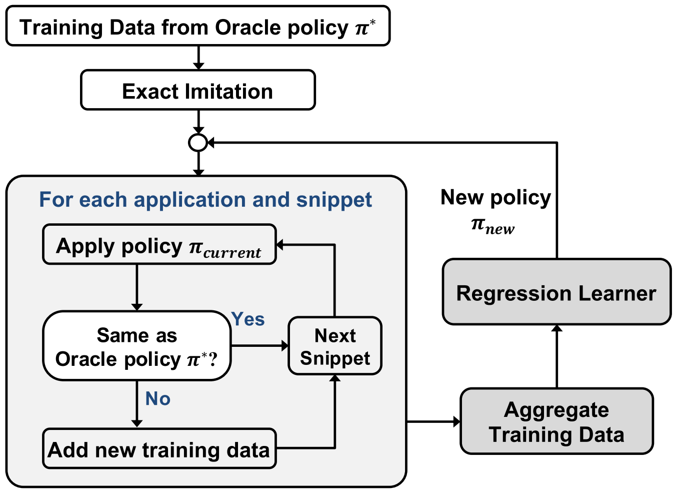
Training Data from Oracle policy π^*
Exact Imitation
For each application and snippet
Apply policy $\pi_{current}$
Same as Oracle policy π^*?
Yes
Next Snippet
No
Add new training data
New policy π_{new}
Regression Learner
Aggregate Training Data

Fig. 2. Overview of the DAgger algorithm.

C. Storage and Decision-Making Overhead

We employ two simple and effective regression learning algorithms, namely, LR and RTs, to create our policies in the form of linear and nonlinear functions as needed. Recall that our policies are trained offline and are implemented as user space governors on Odroid-XU3 board [9] to evaluate the performance of the learned controllers against the default governors, as detailed in Section VI-E.

1) Storage Overhead: Linear regressor learns a weighted linear function of the features $\Phi(S_k)$: one weight for each feature. Since we have less than ten features and learn four policies, the storage requirements are only 40 weights. For RTs, we consider short depth values (less than ten). Trees can be interpreted as a set of if-then rules, where we need to store the comparison operators for each if condition, which has also very low storage requirements.

2) Decision-Making Overhead: The proposed policies, which are implemented as user space governors, are invoked periodically. The complexity of making control decisions using linear functions and RTs is around 40 multiplications and up to ten comparisons, respectively, which is insignificant.

We note that the proposed power management technique can be easily extended to the case where each individual core has its own voltage/frequency island. In this case, the frequency of each core will be determined using an RT. Therefore, the number of RTs required is linear in the number of cores if each core has its voltage/frequency island. Since the number of types of cores is small in mobile platforms (<10) and the overhead for each RT is small, our overall approach is highly scalable.

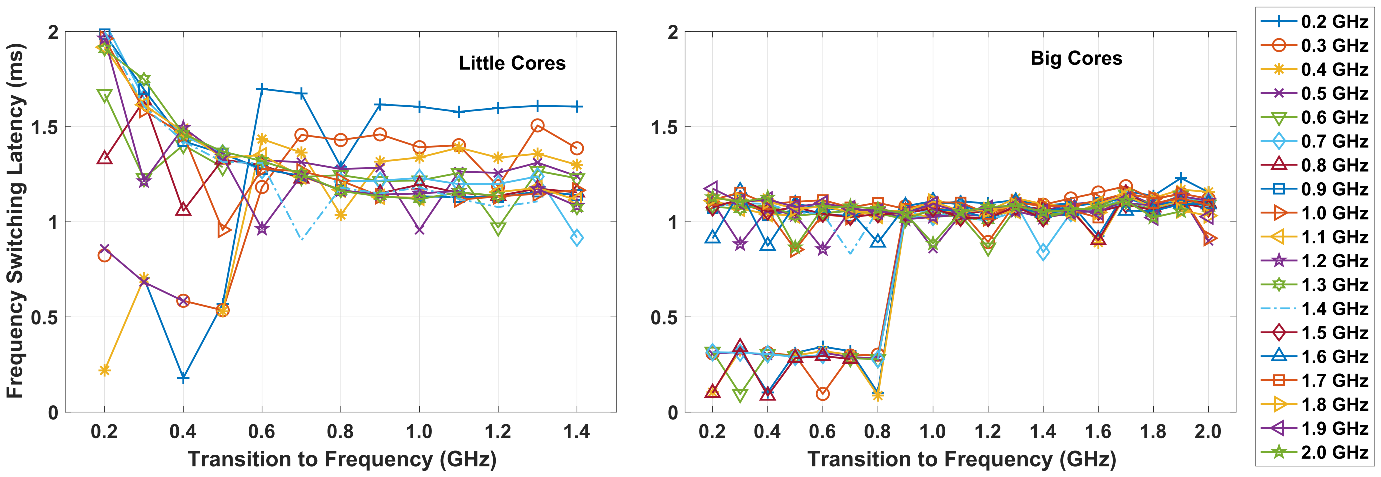

Fig. 3. Illustration of frequency transition overhead for little cores and big cores, respectively. Lines in each subplot correspond to the initial frequency in the transition while the x-axis corresponds to the target frequency in the transition.

VI. Experimental Evaluation

A. Experimental Setup

1) Platform: We perform our data characterization and experimental evaluations on an Odroid-XU3 board [9] running Ubuntu OS. The board is equipped with a Samsung Exynos 5422 SoC that integrates four A15 (Big) and four A7 (Little) cores. It supports heterogeneous multiprocessing, which allows tasks to be scheduled to any of the eight cores. We use dynamic hot plugging feature for the CPU cores to apply the decisions of our IL policy at runtime. We also measure the power consumption of the A7 cluster, A15 cluster, the main memory, and the GPU using the INA231 current sensors on the board. In our experiments, we set the power sensors to sample the power every 2.2 ms and average over 16 samples to mitigate the noise in the measurements. The averaged power consumption values are then recorded by the processor. Using this setup, we evaluate the proposed IL approach for 19 application workloads from PARSEC with "simsmall" inputs [33], MiBench [34], CortexSuite [35] benchmark suites, and Tetris, which is a gaming application.

2) Data Collection: The frequency ranges of Big and Little cores are 200 MHz–2.0 GHz and 200 MHz–1.4 GHz, respectively. Both can be changed with a step size of 100 MHz. Assuming one Little core is always active, we can run each application on $4 \times 5 \times 13 \times 19$ (4940) different configurations. Both our measurements and other studies [20] show that energy efficiency degrades when the frequency is below 600 MHz. Thus, we sweep the frequencies starting with 600 MHz with a lock-step size of 100 MHz to make the experiments practical. Finally, the collected data are used to construct the Oracle as illustrated in Table III.

3) Switching Overhead Characterization: Accounting for the switching overhead between configurations is an important step in the construction of the Oracle. Therefore, we characterize the overhead of turning cores on/off and changing the frequency of cores. To find the overhead of turning cores on/off, we add timing calls in the kernel before and after the function used to turn a core on or off. Then, we repeatedly turn

TABLE IV
Latency of Turning On/Off Cores

	Little (ms)	Big (ms)
Turn On	1.37	1.13
Turn Off	5.18	8.04

the cores on and off to record the latency. Finally, the average latency obtained from 1000 experiments is reported. Table IV summarizes the results for the latency of turning cores on/off.

We observe that turning little cores on incurs an average overhead of 1.37 ms, while turning off little cores incurs a higher overhead of 5.18 ms. The overhead to turn off a core is higher since all the processes running on the core have to be migrated before it can be safely turned off. We observe a similar trend for the big cores.

Similarly, we characterize the latency of changing the frequency of the little and big cores by instrumenting the platform-specific frequency governor. Since the little and big clusters use separate voltage regulators, we instrument their latencies separately. After instrumenting the code, we sweep the frequencies of the little and big cores to record the latency of changing the frequency. The average latency for switching between each pair of frequencies is then stored in a lookup table, as illustrated in Fig. 3. This lookup table is used by the Oracle generation algorithm to account for the overhead of frequency changes for the little and big clusters, respectively. We note that only the offline Oracle generation stage uses the lookup table. At runtime, the overhead is automatically reflected in the execution time and energy consumption of the applications. Therefore, the online IL policy does not need to store the lookup table.

The time it takes to change the configuration is in the order of milliseconds, while the sampling period of the power measurement sensors is in the order of tens of milliseconds. Hence, there is no observable change in power consumption while switching to a new configuration. Therefore, the power consumption during switching is assumed to be equal to the

TABLE V
ACCURACY COMPARISON OF LR AND RT-BASED IL POLICIES USING PPW ORACLE

	Prediction Accuracy (%)							
	n_B		n_L		f_B		f_L	
Application	LR	**RT**	LR	**RT**	LR	**RT**	LR	**RT**
BasicMath-Large	99	**99**	99	**99**	89	**90**	80	**81**
Qsort	99	**98**	93	**99**	78	**99**	77	**99**
BlackScholes-4T	98	**98**	71	**65**	81	**82**	71	**72**
FluidAnimate-4T	95	**94**	68	**74**	51	**85**	53	**78**
Tetris	96	**96**	93	**93**	81	**88**	71	**81**

TABLE VI
ACCURACY COMPARISON OF LR AND RT-BASED IL POLICIES USING ENERGY ORACLE

	Prediction Accuracy (%)							
	n_B		n_L		f_B		f_L	
Application	LR	**RT**	LR	**RT**	LR	**RT**	LR	**RT**
BasicMath-Large	99	**99**	99	**99**	99	**99**	99	**99**
Qsort	99	**99**	99	**99**	99	**99**	99	**99**
BlackScholes-4T	90	**84**	66	**64**	88	**86**	79	**77**
FluidAnimate-4T	93	**91**	77	**77**	90	**90**	86	**86**
Tetris	98	**98**	76	**86**	95	**98**	91	**97**

power consumption of the current configuration. We also emphasize that the proposed Oracle and IL policy design methodologies are general. Since they do not depend on any specific overhead values, these methodologies are broadly applicable to other platforms.

B. Linear Regression Versus Regression Tree

We can employ different supervised learning algorithms within our IL methodology to approximate the Oracle policy, including LR and RTs, as described in Section V. The primary considerations are the accuracy with respect to the unseen applications/snippets and efficiency of implementation. We implement our exact imitation-based policy first using LR due to its simplicity, which is important for the practicality of our solution. Then, we compare it to an RT-based policy which allows us to use nonlinear functions by varying the depth of trees.

Our goal is to learn a policy that can accurately predict the configurations of the Oracle at each snippet. In order to compare different policies, we use the configuration accuracy metric. The configuration accuracy measures the distance between the actions chosen by the proposed policy and those chosen by Oracle policy. Accuracy of 100% implies the actions are identical. To express this distance, we define the total number of levels for a control action as C, the level chosen by the oracle as L^*, and the level chosen by our policy as L_{policy}. Accuracy for each snippet is expressed as

$$\text{Accuracy}(\%) = 100 \times \left(1 - \frac{|L_{\text{policy}} - L^*|}{C-1}\right). \qquad (9)$$

The denominator in (9) is $(C-1)$ since this is the maximum possible difference in the number of levels. After finding the accuracy of each snippet, we take the average across all snippets to compute the overall accuracy for each control action. For example, consider an application that consists of two snippets and the big core frequency has eight levels ($C = 8$). If $L_{\text{policy}} = 1$ and $L^* = 2$ for the first snippet, first snippet accuracy $= 100 \times (1-|1-2|/7) = 86\%$. Similarly, assume $L_{\text{policy}} = 2$ and $L_{\text{oracle}} = 2$ for the second snippet. In this case, the achieved accuracy is 100%. Therefore, the overall accuracy is (86+100)/2 = 93% for the big core frequency.

Using this configuration accuracy metric, we observe that accuracy with LR is considerably lower than an RT with depth 12. Tables V and VI compare the accuracy between LR and RT for some representative applications. For example, RT achieves 99% accuracy in predicting the frequency of little cores for Qsort application when the objective of the Oracle is to maximize PPW. In contrast, the corresponding accuracy of LR is 77%, which is not acceptable. Similarly, RT achieves 85% accuracy in predicting the frequency of big cores for four threaded FluidAnimate applications with the same Oracle. In contrast, LR achieves 51% accuracy which is significantly lower than RT. RT performs better than LR also for Tetris, which is a gaming application. Similarly, RT has an advantage over LR for a few other applications, such as those listed in Tables V and VI. In a few cases, RT achieves less accuracy than LR. For example, RT achieves 84% and LR achieves 90% accuracy in predicting the number of big cores for four threaded BlackScholes application when the objective of the Oracle is to minimize energy. However, with the DAgger algorithm, the accuracy when using RT increases for different applications as discussed in Section VI-D. RT provides more freedom due to its parameterized depth, which is useful for incorporating new applications. Since the implementation overhead of RT is small (in the order of 13–200 μs), as described in Section VI-G, the RT-based policy is adopted in the rest of the paper.

C. Application-Specific Versus Global Policy

Existing IL approaches on homogeneous manycore systems [7] employ an application-based optimized policy. However, application-specific optimization is not practical since many applications are unknown, or may not even exist, at design time. Since controllers optimized for a specific application can be considered as a best-case solution, we start with comparing our global IL-based policy against application-specific policies.

We first train our IL-based policy for each application separately to obtain 19 app-specific policies. Then, we use leave-one-out cross validation and Oracle policies for all applications at once to generate a single global policy. That is, the test application is not included in the training of the global policy. Fig. 4 compares our global policy against the app-specific policies in terms of their total energy consumption. In this case, both global and application-specific policies are trained with Oracle minimizing energy. The global policy performs very similarly (in terms of energy consumed) to app-specific policies which are not practical. The largest difference in

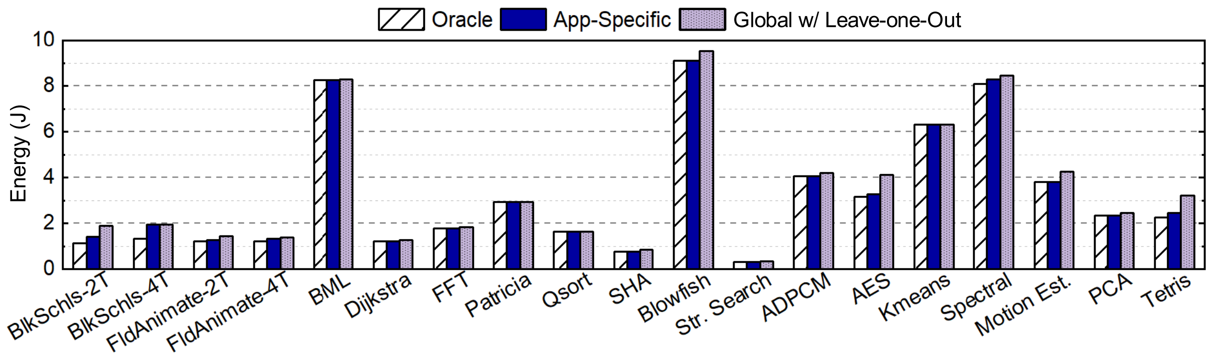

Fig. 4. Comparison of energy between the Oracle, App-Specific, and global policies using Oracle minimizing energy without timing constraints.

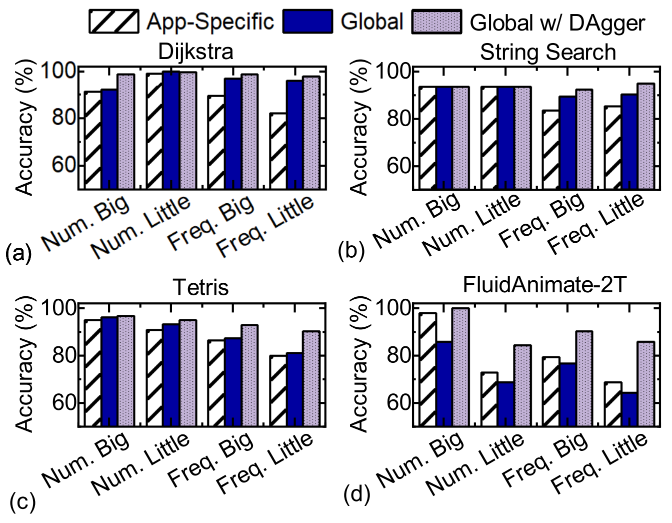

Fig. 5. Comparison of the application-specific and global policies that maximize the PPW while running the (a) Dijkstra, (b) String Search, (c) Tetris, and (d) FluidAnimate-2T applications.

energy between App-Specific and Global policy is observed for the Tetris application which is 30%. On average, the energy consumption of the applications obtained by our global policy is within 6% of the app-specific policies and within 9% of the Oracle policies.

We observe a similar trend in the prediction accuracies. As a representative example, Fig. 5(a) compares the accuracies for making the decisions while running the Dijkstra application. We observe that the prediction accuracies for all decisions are above 90%. Furthermore, the accuracy of the global policy is within 3% of the app-specific policy. The biggest difference between the app-specific and global policy occurs for the two-threaded FluidAnimate application, as shown in Fig. 5(d). For this application, the global policy underestimates the number of active little and big cores. The global policy still achieves a competitive PPW compared to the app-specific policy since it successfully utilizes three–four cores similar to the Oracle, with high accuracy.

In summary, the proposed IL methodology is able to produce a practical global policy which has comparable performance to app-specific policies while generalizing to unseen applications.

D. Global Policy With DAgger

The exact IL-based policy is trained, assuming that the controller always makes the best decision at each snippet. However, at run-time, the controller may make mistakes in choosing the decisions. Therefore, we employ the DAgger algorithm that adds additional training data whenever the current policy makes an incorrect decision. After that, we retrain the policy with the aggregate training data to improve the policy for each of the four control decisions.

Our evaluations with the same applications show that the policy based on exact imitation provides a good starting point for most applications. At the same time, additional DAgger iterations provide significant improvement for Dijkstra and two-threaded FluidAnimate applications. For example, the classification accuracy for the number of Big cores increases from 92% to 99% for Dijkstra and from 86% to 100% for two-threaded FluidAnimate application, as shown in Fig. 5(a) and (d), respectively. On an average, these additional DAgger training iterations enable us to achieve 99.8% accuracy for the number of Big cores, 99.7% accuracy for the number of Little cores, 97.5% accuracy for the Big core frequency, and 95.1% accuracy for the Little core frequency when the Oracle was obtained following the greedy policy maximizing PPW. Because of the high accuracy of control decisions from Dagger-based policy, in this case, we obtain a PPW that is within 2% of the Oracle policy. We also measure the performance of DAgger while the policy is learned from the Oracle that minimizes energy consumption. Fig. 6 shows that DAgger iterations provide significant improvement for String Search and two-threaded FluidAnimate applications. The classification accuracy for the number of Little cores increases from 61% to 84% for String Search, as shown in Fig. 6(d). On average, in this case, we achieve 99.7% accuracy for both the number of Big and Little cores and 99.9% accuracy for both Big and Little core frequencies. As a result of the high accuracy of control decisions from DAgger-based policy, we obtain energy that is within 2% of the Oracle policy.

Moreover, we performed leave-one-out cross validation with the policy learned from DAgger iterations. In this case, the test application is not present while training the policy. Fig. 7 shows the comparison between application-specific policy and global policy obtained from DAgger iterations in terms of their

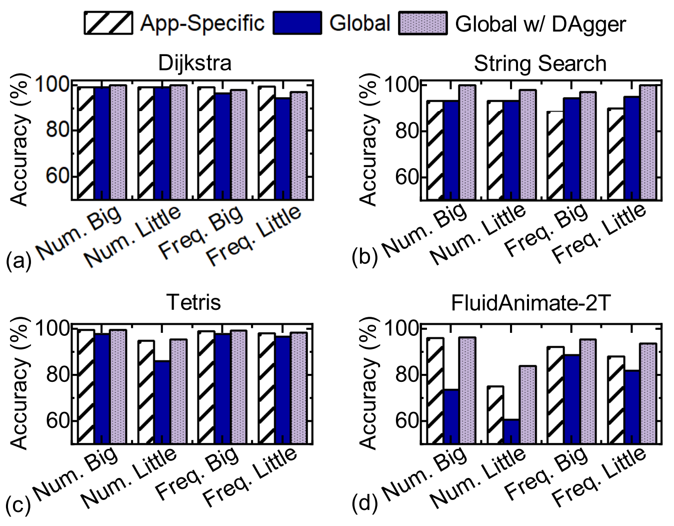

Fig. 6. Comparison of the application-specific and global policies that minimize the energy while running the (a) Dijkstra, (b) String Search, (c) Tetris, and (d) FluidAnimate-2T applications.

PPW measured as giga-instructions per Joule. Here both the policies are trained with the Oracle which maximizes PPW following a greedy approach. The global policy after DAgger iteration performs very similar to application-specific policies in terms of PPW. The largest difference between these two policies is seen for the Qsort application as 6%. We note that for applications like Dijkstra, two-threaded and four-threaded BlackScholes, and Tetris applications, the policy after DAgger iterations is better than the application-specific policy as DAgger iterations have aggregated useful training samples to produce a more generalized policy. Also, on an average, PPW of the global policy is within 2% of the Oracle policy.

We also evaluate the proposed IL policy where we train the policy with the Oracle which minimizes energy with a constraint on the total execution time of the application. In our experiments, we specified execution time constraints as a multiple of the execution time when the application is executed with the performance governor. Specifically, we chose time constraints as 1.2× and 1.5× of the execution time of the performance governor ($t_{\text{performance}}$). With the time constraints, we construct Oracle for each application following the methodology described in Section IV. Then the global IL policy with DAgger iterations is trained with the Oracle. Fig. 8(a) and (b) shows total execution time and energy consumption of an application when the execution time constraint is 1.2× performance governor. In Fig. 8(a), we observe that 16 out of 19 applications satisfy the time constraints when executed with the constructed IL policy. Execution time of Tetris, which is a gaming application, is 3.69 s, where the execution time constraint is 3.85 s. Therefore, Tetris meets the constraint when executed with the global policy with DAgger. The total execution time of fast Fourier transform (FFT), Patricia, and MotionEstimation applications is within 10% of the specified timing constraint. At the same time, with IL policy, 17 out of 19 applications consume energy no higher than 7% of the Oracle energy as shown in Fig. 8(b). Gaming application Tetris consumes 2.43 J of energy which is 6% higher than the Oracle energy. Similarly, we present experimental results when the execution time constraint is 1.5× performance governor. In this case, execution time with learned policy for all applications satisfies the constraint as shown in Fig. 8(c). Except for two-threaded and four-threaded BlackScholes applications, the energy consumption is no higher than 6% of the Oracle energy.

In summary, the proposed IL methodology with DAgger improves the exact imitation policy. Furthermore, the constructed policy through DAgger iterations is suitable to approximate different Oracle policies with different objective functions for different kind of applications.

E. Comparison With State-of-the-Art Governors

Commercial devices are shipped with power and performance management governors like `performance`, `ondemand`, `interactive`, and `powersave`. Therefore, it is critical to compare the proposed IL approach with these default governors. To enable this evaluation, we run each application using all the governors. During this evaluation, we characterize performance, power consumption, and PPW of all the applications similar to our earlier experiments. To provide a holistic view, we compare these governors in terms of both performance and PPW.

The performance governor operates close to the highest operating states whenever the SoC is actively running the applications. Therefore, we normalize the execution time and PPW obtained for each application to the corresponding values given by the performance governor. Hence, the reference in red square (□) marker in Fig. 9 corresponds to the performance governor. Ideally, the results should be in the ideal region (marked in a dashed circle), where execution time is minimized, and performance is maximized. The results of the interactive and ondemand governors are shown with ⋆ and △ markers, respectively. Each marker shows the normalized execution time (x-axis) and the percentage improvement in PPW (y-axis) with respect to the performance governor. Both interactive and ondemand governors have low execution time but poor PPW since they tend to ramp up the frequency whenever the cores are highly utilized. In contrast, the powersave governor, shown in blue circle markers (○), achieves on an average 92% higher PPW than the performance governor. However, this comes at the cost of more than 2.5× higher execution time.

The proposed global IL policy with PPW Oracle (♢) dominates the powersave governor for all data points both in terms of PPW and execution time. We achieve on an average 15% boost in PPW combined with a 25% reduction in execution time. For Tetris application, the IL policy achieves 10% improvement in PPW with a 31% reduction in execution time with respect to powersave governor. While running Kmeans and BML applications concurrently, the IL policy achieves 150% improvement in PPW with respect to the performance governor. At the same time, it reduces execution time, by 27% and consumes 5% less energy with respect to powersave governor. The PPW improvement of our IL policy (with PPW Oracle) with respect to the performance governor ranges

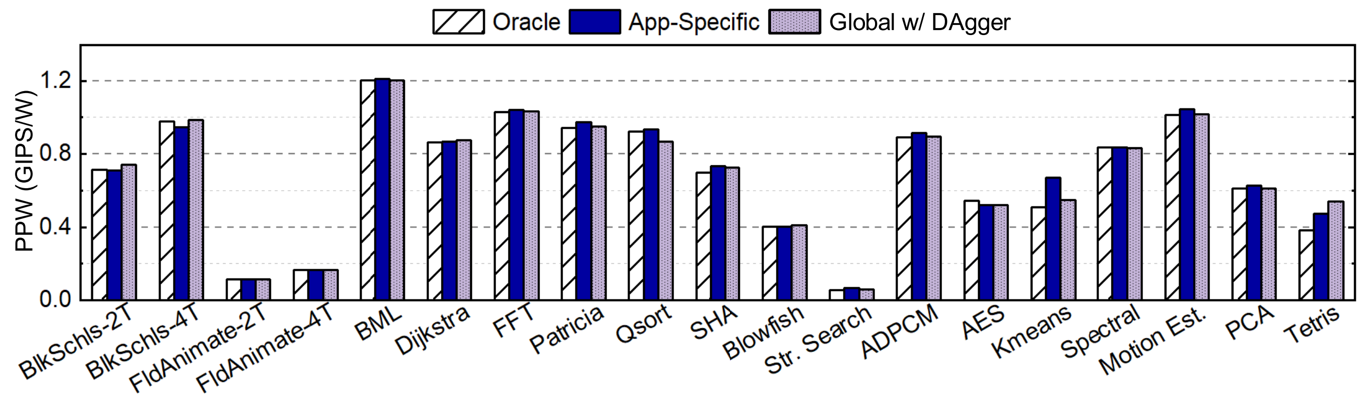

Fig. 7. Comparison of PPW between the Oracle, App-Specific, and global policies using Oracle maximizing PPW.

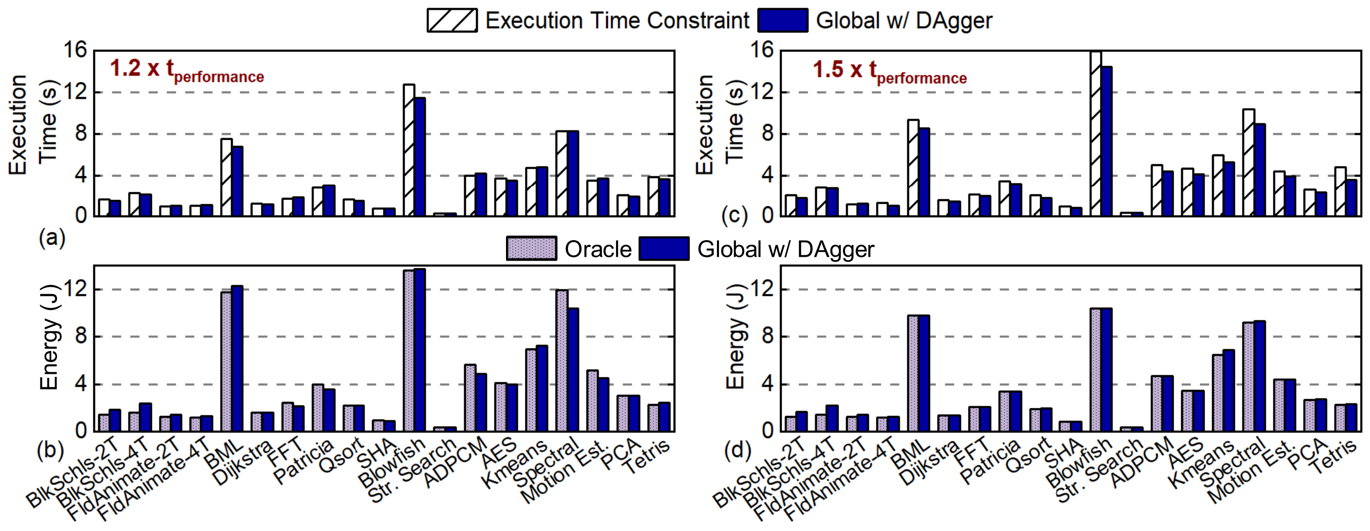

Fig. 8. Comparison of (a) energy and (b) execution time with Oracle when execution time constraint is 1.2× of $t_{\text{performance}}$, and (c) energy and (d) execution time with Oracle when execution time constraint is 1.5× of $t_{\text{performance}}$.

from 75% to 150%. Similarly, IL policy achieves significant PPW improvements when compared to the interactive and ondemand governors. The corresponding penalty in execution time is much smaller than that of the powersave governor.

We also compare the existing governors in the platform with the IL policy that minimizes the energy consumption using red (◊) in Fig. 9. We see improvements in both PPW and execution time when compared to the powersave governor, which aims to minimize power consumption. The improvements for Basicmath and Blowfish applications are highlighted using red dashed arrows in Fig. 9. The PPW improvement of the IL policy range from 101% to 159% with respect to the performance governor. For the gaming application Tetris, this IL policy consumes 39% less energy than powersave governor while having a 9% improvement in execution time. At the same time, Tetris shows 152% improvement in PPW in comparison to the performance governor. Furthermore, while running Kmeans and BML applications concurrently, the policy achieves 5% more PPW than powersave governors with 36% improvement in execution time. At the same time, it achieves 117% increase in PPW than performance governor while running Kmeans and BML applications concurrently. The proposed IL policy for energy also achieves significant improvements when compared to the interactive and ondemand governors. We also observe that the IL policy for energy minimization has a higher PPW than the IL policy for PPW maximization, but at a higher execution time penalty. The IL policy for PPW, on the other hand, achieves a lower penalty for execution time while sacrificing the improvement in PPW.

In summary, our IL policies provide a significant step toward the ideal corner. They can replace the powersave governor and can be used in conjunction with the performance-oriented governors whenever the optimization goal is to maximize the PPW or minimize energy consumption.

F. Comparison With DyPO [3]

DyPO is a recent approach proposed in [3] for Pareto-optimal configuration selection in heterogeneous processors. The DyPO algorithm first identifies the Pareto-optimal configurations for a given performance metric and designs a classifier to choose the optimal configurations at runtime. We compare our IL policies for PPW and energy with the DyPO-Energy algorithm in [3]. In Fig. 10, we compare energy

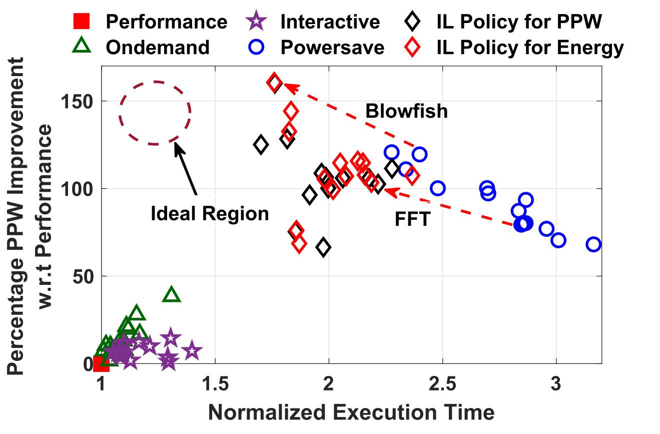

Fig. 9. Comparison of the IL policy with default governors on Odroid-XU3. Markers represent different applications.

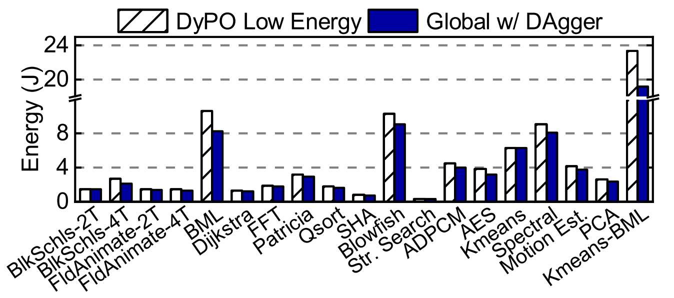

Fig. 10. Comparison of energy between DyPO and global IL policy for different applications.

consumption for different applications with DyPO and the proposed global policy constructed through IL and DAgger iterations. For all applications, the proposed IL policy has less energy consumption than DyPO. For BML, the energy consumption with the IL policy is 22% less than DyPO. For multithreaded applications also, the IL policy shows improvement over DyPO. For example, four-threaded Blackscholes application consumes 20% less energy when it is executed with the IL policy than DyPO. Moreover, proposed IL policy consumes 18% less energy than DyPO when Kmeans and BML applications run concurrently. On average, the proposed IL policy consumes 10% less energy than DyPO for the applications shown in the figure. Table VII summarizes the average improvement in PPW, energy, and execution time obtained by our respective IL policies. We see that the proposed IL policy has 6.3%, 4.7%, and 9.2% improvement in PPW, energy, and execution time, respectively, with respect to DyPO when the objective of the Oracle is to maximize PPW. On the other hand, the IL policy has 10.7%, 9.3%, and 5.1% improvement in PPW, energy, and execution time, respectively, with respect to DyPO when the objective of the Oracle is to minimize the energy consumed. The improvement over DyPO can be attributed to two primary reasons. First, DyPO limits the number of configurations available at runtime by pruning the configurations using the k-means clustering algorithms. As a result, it is unable to use the full breadth of configurations available in the platform. In contrast, the proposed approach

TABLE VII
IMPROVEMENT WITH RESPECT TO DYPO [3]

	Percentage Improvement		
Oracle type	**PPW (GIPS/W)**	**Energy (J)**	**Exe. time (s)**
Maximizing PPW	6.3%	4.7%	9.2%
Minimizing Energy	10.7%	9.3%	5.1%

TABLE VIII
Q-TABLE SIZE FOR DIFFERENT NUMBER OF BINS

Bins	Features	State-Action Pairs	Accuracy (%)
4	10	$10^6 \times 4940$	92.3
6	10	$6 \times 10^7 \times 4940$	95.1

has a much larger range of configurations to choose from at runtime. As a result of this, it is able to choose configurations that have better energy or PPW, depending on the optimization objective of the policy. Second, DyPO uses a much simpler logistic regression classifier while the proposed approach uses RTs. RTs are able to approximate the nonlinear behavior of the policy better in comparison to logistic regression. In summary, the proposed IL-based policy is able to improve energy, PPW, and execution time as well when compared to the DyPO approach.

G. Implementation Overhead

In this section, we evaluate the implementation overhead of the proposed approach on the Odroid-XU3 platform [9]. To this end, the proposed IL policies are implemented as user space governors. We measure the run-time overhead of our policy for each snippet in the application set. The execution time of our policy ranges from 13 to 200 μs. This amounts to an overhead of 0.07%–1% for an average snippet of length 20 ms. The proposed approach also has a negligible storage overhead as we only have to store the comparison operators of the if-then rules corresponding to RT-based policy. Specifically, the RTs implemented on the board have a maximum of 512 nonleaf nodes. This leads to a storage overhead of 2 kB for each RT. In contrast, RL approaches require the storage of a Q-table with one entry for each state-action pair defined by the features in Table I and available control actions (4940). For example, with four bins and ten features, we achieve an average prediction accuracy of 92.3%, as shown in Table VIII. Increasing the number of bins to six improves the accuracy to 95.1%. However, with six bins, the Q-table size grows to $6 \times 10^7 \times 4940$ entries, which is infeasible on most mobile platforms including ours.

VII. CONCLUSION

Managing mobile platforms at runtime is challenging due to increasing heterogeneity, the number of processors, the large space of control actions, and exponential growth in applications. Existing governors on commercial devices employ simple heuristics based on system utilization, which leads to suboptimal performance. This paper presented a practical approach for dynamic management of mobile processors using the framework of IL. We start with the construction of an Oracle policy to maximize PPW for a set of applications. Using this Oracle, we construct runtime policies that are applicable to a broad range of application workloads. Experiments on a commercial mobile platform show 101% improvement in PPW on an average while running 19 commonly employed benchmarks. As a next step, our future work will extend the proposed IL technique to perform online improvements in the policy. This will enable the policy to adapt itself to a wide range of previously unseen workloads.

References

[1] Statista. *Mobile App Usage—Statistics & Facts*. Accessed: Nov. 24, 2018. [Online]. Available: https://www.statista.com/topics/1002/mobile-app-usage/

[2] D. Brodowski. *CPUFreq Governor*. Accessed: Apr. 29, 2014. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[3] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, "DyPO: Dynamic Pareto-optimal configuration selection for heterogeneous MpSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, p. 123, 2017.

[4] S. Schaal, "Is imitation learning the route to humanoid robots?" *Trends Cognit. Sci.*, vol. 3, no. 6, pp. 233–242, 1999.

[5] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 627–635.

[6] W. Sun, A. Venkatraman, G. J. Gordon, B. Boots, and J. A. Bagnell, "Deeply AggreVaTeD: Differentiable imitation learning for sequential prediction," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, pp. 3309–3318, Aug. 2017.

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

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim., Feedback-Directed Runtime Optim.*, 2004, p. 75.

[9] Hardkernel. (2014) *ODROID-XU3*. Accessed: Nov. 24, 2018. [Online]. Available: https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[11] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[12] U. Gupta *et al.*, "Dynamic power budgeting for mobile systems running graphics workloads," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 1, pp. 30–40, Jan./Mar. 2018.

[13] N. Vallina-Rodriguez and J. Crowcroft, "Energy management techniques in modern mobile handsets," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 1, pp. 179–198, 4th Quart., 2012.

[14] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 2007, pp. 38–43.

[15] X. Chen *et al.*, "Dynamic voltage and frequency scaling for shared resources in multicore processor designs," in *Proc. Design Autom. Conf.*, 2013, p. 114.

[16] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated CPU-GPU power management for 3D mobile games," in *Proc. 51st ACM/EDAC/IEEE Design Automat. Conf.*, May 2014, pp. 1–6.

[17] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. 50th Annu. Design Automat. Conf.*, 2013, p. 174.

[18] D. Kadjo, U. Ogras, R. Ayoub, M. Kishinevsky, and P. Gratz, "Towards platform level power management in mobile systems," in *Proc. IEEE 27th Int. Syst.-Chip Conf. (SOCC)*, Sep. 2014, pp. 146–151.

[19] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proc. Linux Symp.*, vol. 2. 2006, pp. 215–230.

[20] A. Aalsaud *et al.*, "Power–aware performance adaptation of concurrent applications in heterogeneous many-core systems," in *Proc. Int. Symp. Low Power Electron. Design*, 2016, pp. 368–373.

[21] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "SPARTA: Runtime task allocation for energy efficient heterogeneous manycores," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2016, pp. 1–10.

[22] K. R. Basireddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster thread-to-core mapping and DVFS on heterogeneous multi-cores," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 3, pp. 369–382, Jul. 2018.

[23] G. Bhat, G. Singla, A. K. Unver, and U. Y. Ogras, "Algorithmic optimization of thermal and power management for heterogeneous mobile platforms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 544–557, Mar. 2018.

[24] J.-G. Park, N. Dutt, and S.-S. Lim, "ML-Gov: A machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming," in *Proc. Symp. Embedded Syst. Real-Time Multimedia*, 2017, pp. 12–21.

[25] J. F. Martinez and E. Ipek, "Dynamic multicore resource management: A machine learning approach," *IEEE Micro*, vol. 29, no. 5, pp. 8–17, Sep./Oct. 2009.

[26] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, 2015, pp. 1521–1526.

[27] F. M. M. U. 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.

[28] Q. Fettes, M. Clark, R. Bunescu, A. Karanth, and A. Louri, "Dynamic voltage and frequency scaling in NoCs with supervised and reinforcement learning techniques," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 375–389, Mar. 2018.

[29] U. Gupta, S. K. Mandal, M. Mao, C. Chakrabarti, and U. Y. Ogras, "A deep Q-learning approach for dynamic management of heterogeneous processors," *IEEE Comput. Archit. Lett.*, vol. 18, no. 1, pp. 14–17, Jun./Jul. 2019.

[30] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Dept. Defense HPCMP Users Group Conf.*, vol. 710, 1999.

[31] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative CPU-GPU thermal management," in *Proc. Design Autom. Conf.*, 2016, p. 47.

[32] X. Cai, T. Kloks, and C. K. Wong, "Time-varying shortest path problems with constraints," *Netw., Int. J.*, vol. 29, no. 3, pp. 141–150, 1997.

[33] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. Int. Workshop Workload Characterization*, 2001, pp. 3–14.

[35] S. Thomas *et al.*, "CortexSuite: A synthetic brain benchmark suite," in *Proc. IISWC*, Oct. 2014, pp. 76–79.