Runtime Task Scheduling Using Imitation Learning for Heterogeneous Many-Core Systems

Anish Krishnakumar, *Graduate Student Member, IEEE*, Samet E. Arda, *Member, IEEE*, A. Alper Goksoy, Sumit K. Mandal, *Graduate Student Member, IEEE*, Umit Y. Ogras, *Senior Member, IEEE*, Anderson L. Sartor, *Member, IEEE*, and Radu Marculescu, *Fellow, IEEE*

***Abstract*—Domain-specific systems-on-chip, a class of heterogeneous many-core systems, is recognized as a key approach to narrow down the performance and energy-efficiency gap between custom hardware accelerators and programmable processors. Reaching the full potential of these architectures depends critically on optimally scheduling the applications to available resources at runtime. Existing optimization-based techniques cannot achieve this objective at runtime due to the combinatorial nature of the task scheduling problem. As the main theoretical contribution, this article poses scheduling as a classification problem and proposes a hierarchical imitation learning (IL)-based scheduler that learns from an Oracle to maximize the performance of multiple domain-specific applications. Extensive evaluations with six streaming applications from wireless communications and radar domains show that the proposed IL-based scheduler approximates an offline Oracle policy with more than 99% accuracy for performance- and energy-based optimization objectives. Furthermore, it achieves almost identical performance to the Oracle with a low runtime overhead and successfully adapts to new applications, many-core system configurations, and runtime variations in application characteristics.**

***Index Terms*—Domain-specific SoC (DSSoC), heterogeneous computing, imitation learning (IL), many-core architectures, scheduling.**

I. Introduction

HOMOGENEOUS multicore architectures have successfully exploited thread- and data-level parallelism to achieve performance and energy efficiency beyond the limits of single-core processors. While general-purpose computing achieves programming flexibility, it suffers from significant performance and energy efficiency gap when compared to special-purpose solutions. Domain-specific architectures, such as graphics processing units (GPUs) and neural network (NN) processors, are recognized as some of the most promising solutions to reduce this gap [13]. Domain-specific systems-on-chips (DSSoCs), a concrete instance of this new architecture, judiciously combine general-purpose cores, special-purpose processors, and hardware accelerators. DSSoCs approach the efficacy of fixed-function solutions for a specific domain while maintaining programming flexibility for other domains [11].

The success of DSSoCs depends critically on satisfying two intertwined requirements. First, the available processing elements (PEs) must be utilized optimally, at runtime, to execute the incoming tasks. For instance, scheduling all tasks to general-purpose cores may work, but diminishes the benefits of the special-purpose PEs. Likewise, a static task-to-PE mapping could unnecessarily stall the parallel instances of the same task. Second, the acceleration of domain-specific applications needs to be oblivious to the application developers to make DSSoCs practical. This article addresses these two requirements simultaneously.

The task scheduling problem involves assigning tasks to PEs and ordering their execution to achieve the optimization goals, e.g., minimizing execution time, power dissipation, or energy consumption. To this end, applications are abstracted using mathematical models, such as directed acyclic graph (DAG) and synchronous data graphs (SDGs) that capture both the attributes of individual tasks (e.g., expected execution time) and the dependencies among the tasks [6], [28], [33]. Scheduling these tasks to the available PEs is a well-known NP-complete problem [9], [34]. An optimal *static schedule* can be found for small problem sizes using optimization techniques, such as mixed-integer programming (MIP) [10] and constraint programming (CP) [27]. These approaches are not applicable to runtime scheduling for two fundamental reasons. First, statically computed schedules lose relevance in a dynamic environment where tasks from multiple applications stream in parallel, and PE utilizations change dynamically. Second, the execution time of these algorithms, hence their overhead, can be prohibitive even for small problem sizes with few tens of tasks. Therefore, a variety of heuristic schedulers, such as the shortest job first (SJF) [35] and complete fair

Manuscript received April 17, 2020; revised June 12, 2020; accepted July 6, 2020. This work was supported by the Air Force Research Laboratory and Defense Advanced Research Projects Agency under Grant FA8650-18-2-7860. This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Anish Krishnakumar.)*

Anish Krishnakumar, Samet E. Arda, A. Alper Goksoy, Sumit K. Mandal, and Umit Y. Ogras are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: anishnk@asu.edu; sarda1@asu.edu; aagoksoy@asu.edu; skmandal@asu.edu; umit@asu.edu).

Anderson L. Sartor is with the Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: asartor@cmu.edu).

Radu Marculescu is with the Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX 78712 USA (e-mail: radum@utexas.edu).

Digital Object Identifier 10.1109/TCAD.2020.3012861

schedulers (CFSs) [24], are used in practice for homogeneous systems. These algorithms tradeoff the quality of scheduling decisions and computational overhead.

To improve this state of affairs, this article addresses the following challenging proposition: can we achieve a scheduler performance close to that of optimal MIP and CP schedulers while using minimal runtime overhead compared to commonly used heuristics? Furthermore, we investigate this problem in the context of heterogeneous PEs. We note that much of the scheduling in heterogeneous many-core systems is tuned manually, even to date [3]. For example, OpenCL, a widely used programming model for heterogeneous cores, leaves the scheduling problem to the programmers. Experts manually optimize the task to resource mapping based on their knowledge of application(s), characteristics of the heterogeneous clusters, data transfer costs, and platform architecture. However, manual optimization suffers from scalability for two reasons. First, optimizations do not scale well for all applications. Second, extensive engineering efforts are required to adapt the solutions to different platform architectures and varying levels of concurrency in applications. Hence, there is a critical need for a methodology to provide optimized scheduling solutions applicable to a variety of applications at runtime in heterogeneous many-core systems.

Scheduling has traditionally been considered as an optimization problem [10]. We change this perspective by formulating runtime scheduling for heterogeneous many-core platforms as a classification problem. This perspective and the following *key insights* enable us to employ machine learning (ML) techniques to solve this problem.

Key Insight 1: One can use an optimal (or near-optimal) scheduling algorithm offline without being limited by computational time and other runtime overheads. Then, the inputs to this scheduler and its decisions can be recorded along with relevant features to construct an Oracle.

Key Insight 2: One can design a policy that approximates the Oracle with minimum overhead and use this policy at runtime.

Key Insight 3: One can exploit the effectiveness of ML to learn from Oracle with different objectives, which includes minimizing execution time, energy consumption, etc.

Realizing this vision requires addressing several challenges. First, we need to construct an Oracle in a dynamic environment where tasks from multiple applications can overlap arbitrarily, and each incoming application instance observes a different system state. Finding optimal schedules is challenging even offline since the underlying problem is NP-complete. We address this challenge by constructing Oracles using both CP and a computationally expensive heuristic, called the earliest task first (ETF) [14]. ML uses informative properties of the system (*features*) to predict the category in a classification problem. The second challenge is identifying the minimal set of relevant features that can lead to high accuracy with minimal overhead. We store a small set of 45 relevant features for a many-core platform with 16 PEs along with the Oracle to minimize the runtime overhead. This enables us to represent a complex scheduling decision as a set of features and then predict the best PE for task execution. The final challenge is approximating the Oracle accurately with a minimum implementation overhead. Since runtime task scheduling is a sequential decision-making problem, supervised learning methodologies, such as linear regression and regression tree (RT), may not generalize for unseen states at runtime. Reinforcement learning (RL) and imitation learning (IL) are more effective for sequential decision-making problems [19], [29], [31]. Indeed, RL has shown promise when applied to the scheduling problem [20], [21], [37], but it suffers from slow convergence and sensitivity of the reward function [15], [18]. In contrast, IL takes advantage of the expert's inherent knowledge and produces policies that imitate expert decisions [30]. Hence, we propose an IL-based framework that schedules incoming applications to heterogeneous multicore systems.

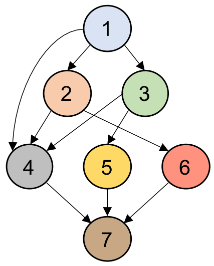
The proposed IL framework is formulated to facilitate generalization, i.e., it can be adapted to learn from any Oracle that optimizes a specific objective, such as performance and energy efficiency, of an arbitrary DSSoC. We evaluate the proposed framework with six domain-specific applications from wireless communications and radar systems. The proposed IL policies successfully approximate the Oracle with more than 99% accuracy, achieving fast convergence and generalizing to unseen applications. In addition, the scheduling decisions are made within 1.1 μs (on an Arm A53 core), which is better than CFS performance (1.2 μs). To the best of our knowledge, this is the first IL-based scheduling framework for heterogeneous many-core systems capable of handling multiple applications exhibiting streaming behavior. The main contributions of this article are as follows.

1) An IL framework to construct policies for task scheduling in heterogeneous many-core platforms.
2) Oracle design using both optimal and heuristic schedulers for performance- and energy-based optimization objectives.
3) Extensive experimental evaluation of the proposed IL policies along with latency and storage overhead analysis.
4) Performance comparison of IL policies against RL and optimal schedules obtained by CP.

The remainder of this article is organized as follows. We review the related work in Section II. Section III provides background information on DAG scheduling and IL. In Section IV, we discuss the proposed methodology, followed by relevant experimental results in Section V. Section VI presents the conclusions and possible future research for this work.

II. Related Work and Novel Contributions

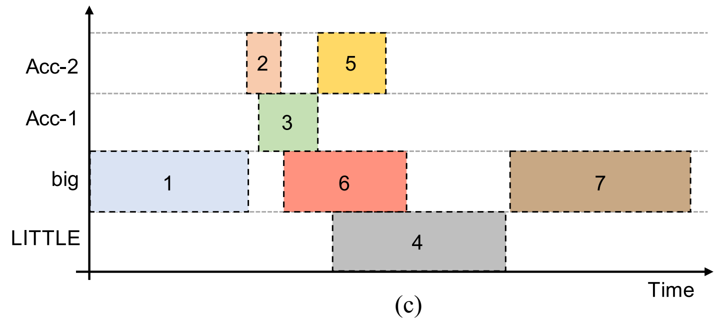
Current many-core systems use runtime heuristics to enable scheduling with low overheads. For example, the completely fair scheduler (CFS) [24], widely used in Linux systems, aims to provide fairness for all processes in the system. CFS maintains two queues (active and expired) to manage task scheduling. In addition, CFS gives a fixed-time quantum for each process. Tasks are swapped between active and expired queues based on activation and expiration of the time quantum. However, complex heuristics are required to manage such queues. CFS also does not generalize to optimization objectives apart from performance and fairness. More importantly,



(a)

Clusters and PEs	Supported Tasks
High-performance (big) general-purpose	1, 2, 3, 4, 5, 6, 7
Low-power (LITTLE) general-purpose	1, 2, 3, 4, 5, 6, 7
Hardware accelerator-1 (Acc-1)	3, 5
Hardware accelerator-2 (Acc-2)	2, 5, 6

(b)



(c)

Fig. 1. (a) Example DAG consisting of seven tasks. (b) Heterogeneous computing platform with four PEs and list of tasks in DAG supported by each PE. (c) Sample schedule of the DAG on the heterogeneous many-core system.

CFS scheduling is limited to general-purpose cores and lacks support for specialized cores and hardware accelerators [7]. With the same limitations, SJF [35] scheduler estimates the task's CPU processing time and assigns the first available resource to the task with the shortest execution time.

List scheduling techniques [16], [28] for DAGs [4], [8], [33] prioritize various objectives, such as energy [6], [32], fairness [41], and security [40]. In general, this technique places the nodes (tasks) of a DAG in a list and provides a PE assignment and order at design time. Heterogeneous earliest finish time (HEFT) [33] is one example, in which an upward rank is computed to perform the scheduling decisions. Bittencourt *et al.* [8] used a lookahead algorithm as an enhancement to the HEFT scheduler to improve the execution time, but suffers from fourth-order complexity $O(n^4)$ on the number of tasks (n). Another recent technique shows improvement in performance with quadratic complexity [4]. However, these algorithms suffer from the time complexity problem and are tailored to particular objectives and fail to generalize to a combination of objectives and choice of applications.

ML-based schedulers show promise in eliminating the drawbacks of list scheduling and runtime heuristic techniques. ML-based schedulers possess the capabilities to be further tuned at runtime [20]. A recent support vector machine (SVM)-based scheduler for OpenCL kernels assigns kernels (tasks) between CPUs and GPUs [38]. In contrast to schedulers that use supervised learning, Mirhoseini *et al.* [22] used RL to schedule Tensorflow device placement but lacks the ability of scheduling streaming jobs. DeepRM [20] uses deep NNs with RL for scheduling at an application granularity as opposed to using the notion of DAGs. On the other hand, Decima [21] uses a combination of graph NNs and RL to perform coarse-grained processing-cluster level scheduling for streaming DAGs.

RL-based scheduling techniques have two major drawbacks. *First,* they require a significant number of episodes to converge. For example, the technique proposed in [21] takes 50k episodes, with 1.5 s each, to converge to a solution that is equivalent to 21 h of simulation in Nvidia Tesla P100 GPU. *Second*, the efficiency of an RL-based technique predominantly depends on the choice of the reward function. Usually, the reward function is hand tuned, depending on the problem under consideration.

To overcome these difficulties, we propose an IL-based scheduling methodology. Since IL uses an Oracle to construct a policy, it does not suffer from slow convergence, as seen in RL. IL-based policies were initially used in robotics to show their fast convergence property [30]. Recently, the use of IL to intelligently manage power and energy consumption in SoCs has been demonstrated [15], [18]. To the best of our knowledge, this is the first approach that applies IL for multiapplication streaming task scheduling in heterogeneous many-core platforms.

III. Background and Overview

The runtime scheduling problem addressed in this article is illustrated in Fig. 1. We consider streaming applications that can be modeled using DAGs, such as the one shown in Fig. 1(a). These applications process data frames that arrive at a varying rate over time. For example, a WiFi-transmitter, one of our domain applications, receives and encodes raw data frames before they are transmitted over the air. Data frames from a single application or multiple simultaneous applications can overlap in time as they go through the tasks that compose the application. For instance, Task-1 in Fig. 1(a) can start processing a new frame, while other tasks continue processing earlier frames. Processing of a frame is said to be completed after the terminal task without any successor [Task-7 in Fig. 1(a)] is executed. We define the application formally to facilitate a description of the schedulers.

Definition 1: An *application graph* $G_{\text{App}}(\mathcal{T}, \mathcal{E})$ is a DAG, where each node $T_i \in \mathcal{T}$ represents the tasks that compose the application. Directed edge $e_{ij} \in \mathcal{E}$ from task T_i to T_j shows that T_j cannot start processing a new frame before the output of T_i reaches T_j for all $T_i, T_j \in \mathcal{T}$. v_{ij} for each edge $e_{ij} \in \mathcal{E}$ denotes the communication volume over this edge. It is used to account for communication latency.

Each task in a given application graph G_{App} can execute on different PEs in the target DSSoC. We formally define the target DSSoC as follows.

Definition 2: An *architecture graph* $G_{\text{Arch}}(\mathcal{P}, \mathcal{L})$ is a directed graph, where each node $P_i \in \mathcal{P}$ represents PEs, and $L_{ij} \in \mathcal{L}$ represents the communication links between P_i and P_j in the target SoC. The nodes and links have the following quantities associated with them.

1) $t_{\text{exe}}(P_i, T_j)$ is the execution time of task T_j on PE $P_i \in \mathcal{P}$, if P_i can execute (i.e., it supports) T_j.
2) $t_{\text{comm}}(L_{ij})$ is the communication latency from P_i to P_j for all $P_i, P_j \in \mathcal{P}$.

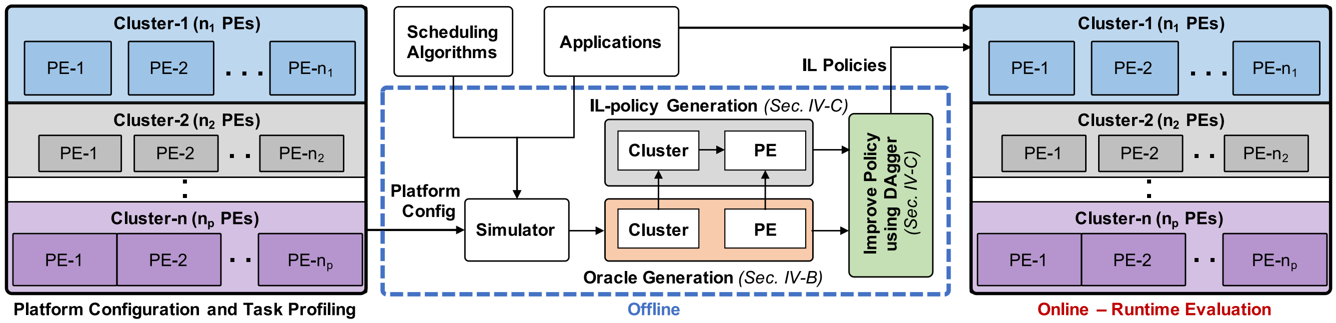


Fig. 2. Overview of the proposed IL framework for task scheduling in heterogeneous many-core systems. The framework integrates the system configuration, profiling information, scheduling algorithms, and applications to construct Oracle, and trains IL policies for task scheduling. The IL policies that are improved using DAgger are then evaluated on the heterogeneous many-core system at runtime.

3) $C(P_i) \in \mathcal{C}$ is the PE cluster $P_i \in \mathcal{P}$ belongs to.

The DSSoC example in Fig. 1(b) assumes one big core cluster, one LITTLE core cluster, and two hardware accelerators each with a single PE in them for simplicity. The low-power (LITTLE) and high-performance (big) general-purpose clusters can support the execution of all tasks, as shown in the *supported tasks* column in Fig. 1(b). In contrast, hardware accelerators (Acc-1 and Acc-2) support only a subset of tasks.

A particular instance of the scheduling problem is illustrated in Fig. 1(c). Task-6 is scheduled to big core (*although it executes faster on Acc-2*) since Acc-2 is not available at the time of decision making. Similarly, Task-4 is scheduled to the LITTLE core (*even if it executes faster on big*) because the big core is utilized when Task-4 is ready to execute. In general, scheduling complex DAGs in heterogeneous many-core platforms present a multitude of choices making the runtime scheduling problem highly complex. The complexity increases further with: 1) overlapping DAGs at runtime; 2) executing multiple applications simultaneously; and 3) optimizing for objectives, such as performance, energy, etc.

The proposed solution leverages IL, and is outlined in Fig. 2. It is also referred to as learning by demonstration and is an adaption of supervised learning for sequential decision-making problems. The decision-making space is segmented into distinct decision epochs, called *states* ($\mathcal{S}$). There exists a finite set of actions $\mathcal{A}$ for every state $s \in \mathcal{S}$. IL uses policies that map each state (s) to corresponding action.

Definition 3 [Oracle Policy (expert)]: $\pi^*(s) : \mathcal{S} \rightarrow \mathcal{A}$ maps a given system state to the optimal action. In our runtime scheduling problem, the state includes the set of ready tasks and actions that correspond to the assignment of tasks $\mathcal{T}$ to PEs $\mathcal{P}$. Given the Oracle π^*, the goal with IL is to learn a runtime policy that can approximate it. We construct an Oracle offline and approximate it using a hierarchical policy with two levels. Consider a generic heterogeneous many-core platform with a set of clusters $\mathcal{C}$, as illustrated in Fig. 2. At the first level, an IL policy chooses one cluster (among n clusters) for a task to be executed in.

The first-level policy assigns the ready tasks to one of the clusters in $\mathcal{C}$, since each PE within the same cluster has the same static parameters. Then, a cluster-level policy assigns the tasks to a specific PE within that cluster. The details of state representation, Oracle generation, and hierarchical policy design are presented in the next section.

TABLE I
SUMMARY OF THE NOTATIONS USED IN THIS ARTICLE

T_j	Task-j	$\mathcal{T}$	Set of Tasks
P_i	PE-i	$\mathcal{P}$	Set of PEs
c	Cluster-c	$\mathcal{C}$	Set of clusters
L_{ij}	Communication links between P_i to P_j	$\mathcal{L}$	Set of communication links
$t_{exe}(P_i, T_j)$	Execution time of task T_j on PE P_i	$t_{comm}(L_{ij})$	Communication latency from P_i to P_j
s	State-s	$\mathcal{S}$	Set of states
v_{jk}	Communication volume from task T_j to T_k	$\mathcal{A}$	Set of actions
$\mathcal{F}_S$	Static features	$\mathcal{F}_D$	Dynamic features
$\pi_C(s)$	Apply cluster policy for state s	$\pi_{P,c}(s)$	Apply PE policy in cluster-c for state s
π	Policy	π^*	Oracle policy
π^G	Policy for many-core platform configuration G	π^{*G}	Oracle for many-core platform configuration G

IV. Proposed Methodology and Approach

This section first introduces the system state representation, including the features used by the IL policies. Then, it presents the Oracle generation process and the design of the hierarchical IL policies. Table I details the notations that will be used hereafter.

A. System State Representation

Offline scheduling algorithms are NP-complete even though they rely on static features, such as average execution times. The complexity of runtime decisions is further exacerbated as the system schedules multiple applications that exhibit streaming behavior. In the streaming scenario, incoming frames do not observe an empty system with idle processors. In strong contrast, PEs have different utilization, and there may be an arbitrary number of partially processed frames in the wait queues of the PEs. Since our goal is to learn a set of policies that generalize to all applications and all streaming intensities, the ability to learn the scheduling decisions critically depends on the effectiveness of state representation. The system state should encompass both static and dynamic aspects of the set of tasks, applications, and the target platform. Naive representations of DAGs include adjacency matrix and adjacency list. However, these representations suffer from drawbacks such as large storage requirements, highly sparse matrices which complicates the training of supervised learning techniques, and scalability for multiple streaming applications. In contrast, we carefully study the factors that influence task scheduling in a streaming scenario and construct features that accurately represent the system state. We broadly categorize the features that make up the state as follows.

1) *Task Features:* This set includes the attributes of individual tasks. They can be both static, such as average execution time of a task on a given PE ($t_{\text{exe}}(P_i, T_j)$), and dynamic, such as the relative order of a task in the queue.
2) *Application Features:* This set describes the characteristics of the entire application. They are static features, such as the number of tasks in the application and the precedence constraints between them.
3) *PE Features:* This set describes the dynamic state of the PEs. Examples include the earliest available times (readiness) of the PEs to execute tasks.

The static features are determined at the design time, whereas the dynamic features can only be computed at runtime. The static features aid in exploiting design-time behavior. For example, $t_{\text{exe}}(P_i, T_j)$ helps the scheduler compare the expected performance of different PEs. Dynamic features, on the other hand, present the runtime dependencies between tasks and jobs and also the busy states of the PEs. For example, the expected time when cluster c becomes available for processing adds invaluable information, which is only available at runtime.

In summary, the features of a task comprehensively represent the task itself and the state of the PEs in the system to effectively learn the decisions from the Oracle policy. The specific types of features used in this work to represent the state and their categories are listed in Table II. The static and dynamic features are denoted as $\mathcal{F}_S$ and $\mathcal{F}_D$, respectively. Then, we define the systems state at a given time instant k using the features in Table II as

$$s_k = \mathcal{F}_{S,k} \cup \mathcal{F}_{D,k} \tag{1}$$

where $\mathcal{F}_{S,k}$ and $\mathcal{F}_{D,k}$ denote the static and dynamic features, respectively, at a given time instant k. For an SoC with 16 PEs grouped as five clusters, we obtain a set of 45 features for the proposed IL technique.

B. Oracle Generation

The goal of this work is to develop generalized scheduling models for streaming applications of multiple types to be executed on heterogeneous many-core systems. The generality of the IL-based scheduling framework enables using IL with any Oracle. The Oracle can be any scheduling algorithm that optimizes an arbitrary metric, such as execution time, power consumption, and total SoC energy.

TABLE II
TYPES OF FEATURES EMPLOYED FOR STATE REPRESENTATION FROM POINT OF VIEW OF TASK T_j

Feature Type	Feature Description	Feature Categories
Static ($\mathcal{F}_S$)	ID of task-j in the DAG	Task
	Execution time of a task T_j in PE P_i ($t_{exe}(P_i, T_j)$)	Task PE
	Downward depth of task T_j in the DAG	Task Application
	IDs of predecessor tasks of task T_j	Task Application
	Application ID	Application
	Power consumption of task T_j in PE P_i	Task PE
Dynamic ($\mathcal{F}_D$)	Relative order of task T_j in the ready queue	Task
	Earliest time when PEs in a cluster-c are ready for task execution	PE
	Clusters in which predecessor tasks of task T_j executed	Task
	Communication volume from task T_j to task $T_k(v_{jk})$	Task

To generate the training dataset, we implemented both optimal schedulers using CP and heuristics. These schedulers are integrated into an SoC simulation framework, as explained under experimental results. Suppose a new task T_j becomes ready at time k. The Oracle is called to schedule the task to a PE. The Oracle policy for this action task with system state s_k can be expressed as

$$\pi^*(s_k) = P_i \tag{2}$$

where $P_i \in \mathcal{P}$ is the PE T_j scheduled to and s_k is the system state defined in (1). After each scheduling action, the particular task that is scheduled (T_j), the system state $s_k \in \mathcal{S}$, and the scheduling decision are added to the training data. To enable the Oracle policies to generalize for different workload conditions, we constructed workload mixes using the target applications at different data rates, as detailed in Section V-A.

C. IL-Based Scheduling Framework

This section presents the hierarchical IL-based scheduler for runtime task scheduling in heterogeneous many-core platforms. A hierarchical structure is more scalable since it breaks a complex scheduling problem down into simpler problems. Furthermore, it achieves a significantly higher classification accuracy compared to a flat classifier (>93% versus 55%), as detailed in Section V-D.

Our hierarchical IL-based scheduler policies approximate the Oracle with two levels, as outlined in Algorithm 1. The first-level policy $\pi_C(s) : \mathcal{S} \to \mathcal{C}$ is a coarse-grained scheduler that assigns tasks into clusters. This is a natural choice since individual PEs within a cluster have identical static parameters, i.e., they differ only in terms of their dynamic states. The second level (i.e., fine-grained scheduling) consists of *one dedicated policy* $\pi_{P,c}(s) : \mathcal{S} \to \mathcal{P}$ for each cluster $c \in \mathcal{C}$. These

Algorithm 1: Hierarchical IL Framework

for *task* $T \in \mathcal{T}$ **do**
    s = Get current state for task T
    `/* Level-1 IL policy to assign cluster */`
    $c = \pi_C(s)$
    `/* Level-2 IL policy to assign PE */`
    $p = \pi_{P,c}(s)$
    `/* Assign T to the predicted PE */`
end

Algorithm 2: Methodology to Aggregate Data in a Hierarchical IL Framework

for *task* $T \in \mathcal{T}$ **do**
    s = Get current state for task T
    if $\pi_C(s) == \pi_C^*(s)$ **then**
        if $\pi_{P,c}(s)$!= $\pi_{P,c}^*(s)$ **then**
            Aggregate state s and label $\pi_{P,c}^*(s)$ to the dataset
        end
    end
    else
        Aggregate state s and label $\pi_C^*(s)$ to the dataset
        $c^* = \pi_C^*(s)$
        if $\pi_{P,c^*}(s)$!= $\pi_{P,c^*}^*(s)$ **then**
            Aggregate state s and label $\pi_{P,c}^*(s)$ to the dataset
        end
    end
    `/* Assign T to the predicted PE */`
end

policies assign the input task to a PE within its own cluster, i.e., $\pi_{P,c}(s) \in \mathcal{P}^c \; \forall c \in \mathcal{C}$. We leverage off-the-shelf ML techniques, such as RTs and NNs, to construct the IL policies. The application of these policies approximates the corresponding Oracle policies constructed offline.

IL policies suffer from error propagation as the state–action pairs in the Oracle are not necessarily i.i.d. (independent and identically distributed). Specifically, if the decision taken by the IL policies at a particular decision epoch is different from the Oracle, then the resultant state for the next epoch is also different with respect to the Oracle. Therefore, the error further accumulates at each decision epoch. This can occur during runtime task scheduling when the policies are applied to applications that the policies did not train with. This problem is addressed by the data aggregation algorithm (DAgger), proposed to improve IL policies [26]. DAgger adds the system state and the Oracle decision to the training data whenever the IL policy makes a wrong decision. Then, the policies are retrained after the execution of the workload.

DAgger is not readily applicable to the runtime scheduling problem since the number of states is unbounded as a scheduling decision at time t for state s (s_t) can result in any possible resultant state, s_{t+1}. In other words, the feature space is continuous, and hence, it is infeasible to generate an exhaustive Oracle offline. We overcome this challenge by generating an Oracle on-the-fly. More specifically, we incorporate the proposed framework into a simulator. The offline scheduler used as the Oracle is called *dynamically* for each new task. Then, we augment the training data with all the features, Oracle actions, as well as the results of the IL policy under

TABLE III
CHARACTERISTICS OF APPLICATIONS USED IN THIS STUDY AND THE NUMBER OF FRAMES OF EACH APPLICATION IN THE WORKLOAD

App	# of Tasks	Execution Time (μs)	Supported Clusters	Representation in workload	
				#frames	#tasks
WiFi-TX	27	301	big, LITTLE, FFT	69	1863
WiFi-RX	34	71	big, LITTLE, FFT, Viterbi	111	3774
RangeDet	7	177	big, LITTLE, FFT	64	448
SC-TX	8	56	big, LITTLE	64	512
SC-RX	8	154	big, LITTLE, Viterbi	91	728
TempMit	10	81	big, LITTLE, Matrix mult.	101	1010
TOTAL				**500**	**8335**

construction. Hence, the data aggregation process is performed as part of the dynamic simulation.

The hierarchical nature of the proposed IL framework introduces one more complexity to data aggregation. The cluster policy's output may be correct while the PE cluster reaches a wrong decision (or *vice versa*). If the cluster prediction is correct, we use this prediction to select the PE policy of that cluster, as outlined in Algorithm 2. Then, if the PE prediction is also correct, the execution continues; otherwise, the PE data are aggregated in the dataset. However, if the cluster prediction does not align with the Oracle, in addition to aggregating the cluster data, an on-the-fly Oracle is invoked to select the PE policy, then the PE prediction is compared to the Oracle, and the PE data are aggregated in case of a wrong prediction.

V. EXPERIMENTAL RESULTS

Section V-A presents the experimental methodology and setup. Section V-B explores different ML classifiers for IL. The significance of the proposed features is studied using an RT classifier in Section V-C. Section V-D presents the evaluation of the proposed IL-scheduler. Section V-E analyzes the generalization capabilities of IL-scheduler. The performance analysis with multiple workloads is presented in Section V-F. We demonstrate the evaluation of the proposed IL technique to energy-based optimization objectives in Section V-G. Section V-H presents comparisons with RL-based scheduler and Section V-I analyzes the complexity of the proposed approach.

A. Experimental Methodology and Setup

Domain Applications: The proposed IL scheduling methodology is evaluated using applications from wireless communication and radar processing domains. We employ WiFi-transmitter (WiFi-TX), WiFi-receiver (WiFi-RX), range detection (RangeDet), single-carrier transmitter (SC-TX), single-carrier receiver (SC-RX), and temporal mitigation (TempMit) applications, as summarized in Table III. We construct workload mixes using these applications and run them in parallel.

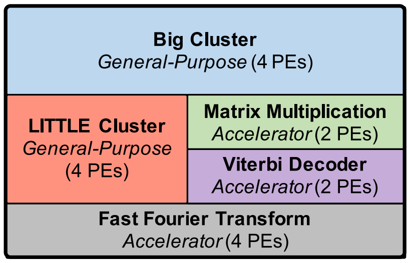


Fig. 3. Configuration of the heterogeneous many-core platform comprising 16 PEs, used for scheduler evaluations.

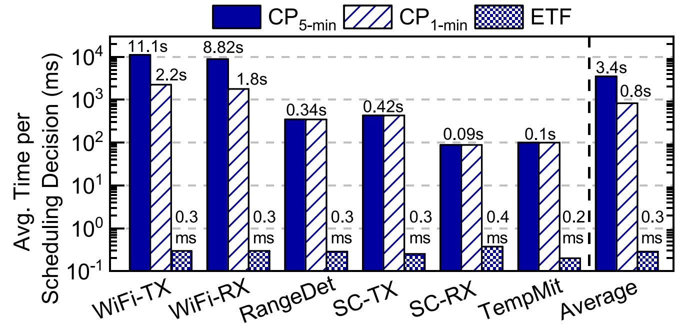


Fig. 4. Comparison of average runtime per scheduling decision for each application with $CP_{5\text{-min}}$, $CP_{1\text{-min}}$, and ETF schedulers.

Heterogeneous DSSoC Configuration: Considering the nature of applications, we employ a DSSoC with 16 PEs, including accelerators for the most computationally intensive tasks; they are divided into five clusters with multiple homogeneous PEs, as illustrated in Fig. 3. To enable power-performance tradeoff while using general-purpose cores, we include a big cluster with four Arm A57 cores and a LITTLE cluster with four Arm A53 cores. In addition, the DSSoC integrates accelerator clusters for matrix multiplication, FFT, and Viterbi decoder to address the computing requirements of the target-domain applications summarized in Table III. The accelerator interfaces are adapted from [17]. The number of accelerator instances in each cluster is selected based on how much the target applications use them. For example, three out of the six reference applications involve FFT while range detection application alone has three FFT operations. Therefore, we employ four instances of FFT hardware accelerators and two instances of Viterbi and matrix multiplication accelerators, as shown in Fig. 3.

Simulation Framework: We evaluate the proposed IL scheduler using the discrete event-based simulation framework [5], *which is validated against two commercial SoCs*: 1) Odroid-XU3 [12] and 2) Zynq Ultrascale+ ZCU102 [1]. This framework enables simulations of the target applications modeled as DAGs under different scheduling algorithms. More specifically, a new instance of a DAG arrives following a specified interarrival time rate and distribution, such as an exponential distribution. After the arrival of each DAG instance, called a frame, the simulator calls the scheduler under study. Then, the scheduler uses the information in the DAG and the current system state to assign the ready tasks to the waiting queues of the PEs. The simulator facilitates storing this information and the scheduling decision to construct the Oracle, as described in Section IV-B.

The execution times and power consumption for the tasks in our domain applications are profiled on Odroid-XU3 and Zynq ZCU102 SoCs. The simulator uses these profiling results to determine the execution time and power consumption of each task. After all the tasks that belong to the same frame are executed, the processing of the corresponding frame completes. The simulator keeps track of the execution time and energy consumed for each frame. These end-to-end values are within 3%, on average, of the measurements on Odroid-XU3 and Zynq ZCU102 SoCs.

Scheduling Algorithms Used for Oracle and Comparisons: We developed a CP formulation using IBM ILOG CPLEX Optimization Studio [2] to obtain the optimal schedules whenever the problem size allows. After the arrival of each frame, the simulator calls the CP solver to find the schedule dynamically as a function of the current system state. Since the CP solver takes hours for large inputs (~100 tasks), we implemented two versions with 1 min ($CP_{1\text{-min}}$) and 5 min ($CP_{5\text{-min}}$) time out per scheduling decision. When the model fails to find an optimal schedule, we use the best solution found within the time limit. Fig. 4 shows that the average time of the CP solver per scheduling decision for the benchmark applications is about 0.8 and 3.5 s, respectively, based on the time limit. Consequently, one entire simulation can take up to two days, even with a time out.

We also implemented the ETF heuristic scheduler, which goes over all tasks and possible assignments to find the earliest finish time considering communication overheads. Its average execution time is close to 0.3 ms, which is still prohibitive for a runtime scheduler, as shown in Fig. 4. However, we observed that it performs better than $CP_{1\text{-min}}$ and marginally worse than $CP_{5\text{-min}}$, as we detail in Section V-D.

Oracle generation with the CP formulation is not practical for two reasons. First, it is possible that for small input sizes (e.g., less than ten tasks), there might be multiple (incumbent) optimal solutions, and CP would choose one of them randomly. The other reason is that for large input sizes, CP terminates at the time limit providing the best solution found so far, which is suboptimal. The suboptimal solutions produced by CP vary based on the problem size and the limit. In contrast, ETF is easier to imitate at runtime and its results are within 8.2% of $CP_{5\text{-min}}$ results. Therefore, we use ETF as the Oracle policy in our experiments and use the results of CP schedulers as reference points. We train IL policies for this Oracle in Section V-B and evaluate their performance in Section V-D.

B. Exploring Different Machine Learning Classifiers for IL

We explore various ML classifiers within the IL methodology to approximate the Oracle policy. One of the key metrics that drive the choice of ML techniques is the classification accuracy of the IL policies. At the same time, the policy should also have a low storage and execution time overheads. We evaluate the following algorithms for classification accuracy and implementation efficiency: RT, support vector classifier (SVC),

TABLE IV
CLASSIFICATION ACCURACIES OF TRAINED IL POLICIES WITH DIFFERENT ML CLASSIFIERS

Classifier	Cluster Policy	LITTLE Policy	big Policy	MatMult Policy	FFT Policy	Viterbi Policy
RT	99.6	**93.8**	**95.1**	99.9	99.5	100
SVC	95.0	**85.4**	**89.9**	97.8	97.5	98.0
LR	89.9	**79.1**	**72.0**	98.7	98.2	98.0
NN	97.7	**93.3**	**93.6**	99.3	98.9	98.1

TABLE V
EXECUTION TIME AND STORAGE OVERHEADS PER IL POLICY FOR RT AND NN CLASSIFIERS

Classifier	Latency (μs)		Storage (KB)
	Odroid-XU3 (Arm A15)	Zynq Ultrascale+ ZCU102 (Arm A53)	
RT	1.1	1.1	19.3
NN	14.4	37	16.9

logistic regression (LR), and a multilayer perceptron NN with four hidden layers and 32 neurons in each hidden layer.

The classification accuracy of ML algorithms under study is listed in Table IV. In general, all classifiers achieve a high accuracy to choose the cluster (the first column). At the second level, they choose the correct PE with high accuracy (> 97%) within the hardware accelerator clusters. However, they have lower accuracy and larger variation for the LITTLE and big clusters (highlighted columns). This is intuitive as the LITTLE and big clusters can execute all types of tasks in the applications, whereas accelerators execute fewer tasks. In strong contrast, a flat policy, which directly predicts the PE, results in training accuracy with 55% at best. Therefore, we focus on the proposed hierarchical IL methodology.

RTs trained with a maximum depth of 12 produce the best accuracy for the cluster and PE policies, with more than 99.5% accuracy for the cluster and hardware acceleration policies. RT also produces an accuracy of 93.8% and 95.1% to predict PEs within the LITTLE and big clusters, respectively, which is the highest among all the evaluated classifiers. The classification accuracy of NN policies is comparable to RT, with a slightly lower cluster prediction accuracy of 97.7%. In contrast, SVCs and LR are not preferred due to the lower accuracy of less than 90% and 80%, respectively, to predict PEs within LITTLE and big clusters.

We choose RTs and NNs to analyze the latency and storage overheads (due to their superior performance). The latency of RT is 1.1 μs on Arm Cortex-A15 in Odroid-XU3 and on Arm Cortex-A53 in Zynq ZCU102, as shown Table V. In comparison, the scheduling overhead of CFS, the default Linux scheduler, on Zynq ZCU102 running Linux Kernel 4.9 is 1.2 μs, which is slightly larger than our solution. The storage overhead of an RT policy is 19.33 KB. The NN policies incur an overhead of 14.4 μs on the Arm Cortex-A15 cluster in Odroid-XU3 and 37 μs on Arm Cortex-A53 in Zynq, with a storage overhead of 16.89 KB. NNs are preferable for use in an online environment as their weights can be incrementally updated using the backpropagation algorithm. However, due to competitive classification accuracy and lower latency overheads of RTs over NNs, we choose RT for the rest of the experiments.

TABLE VI
TRAINING ACCURACY OF IL POLICIES WITH SUBSETS OF THE PROPOSED FEATURE SET

Features Excluded from Training	Cluster Policy	LITTLE Policy	big Policy	MatMul Policy	FFT Policy	Viterbi Policy
None	**99.6**	**93.8**	**95.1**	**99.9**	**99.5**	**100**
Static features	87.3	93.8	92.7	99.9	99.5	100
Dynamic features	88.7	52.1	57.6	94.2	70.5	98
PE availability times	92.2	51.1	61.5	94.1	66.7	98.1
Task ID, depth, app. ID	90.9	93.6	95.3	99.9	99.5	100

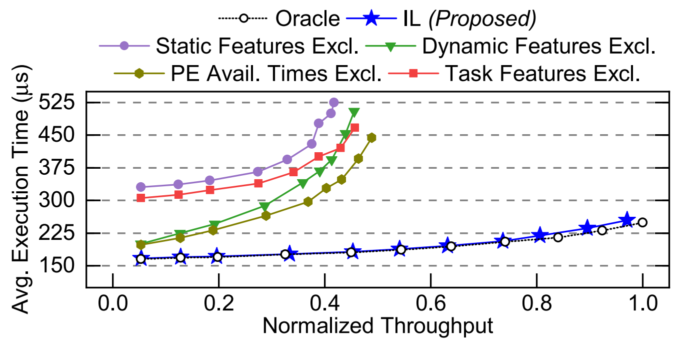


Fig. 5. Average execution time comparison of the applications with Oracle, IL (*Proposed*), and IL policies with subsets of features. As shown, the average execution time with IL closely follows the Oracle.

C. *Feature Space Exploration With Regression Tree Classifier*

This section explores the significance of the features chosen to represent the state. For this analysis, we assess the impact of the input features on the training accuracy with RT classifier and average execution time following a systematic approach.

The training accuracy with subsets of features and the corresponding scheduler performance is shown in Table VI and Fig. 5, respectively. *First*, we exclude all static features from the training dataset. The training accuracy for the prediction of the cluster significantly drops by 10%. Since we use hierarchical IL policies, an incorrect first-level decision results in a significant penalty for the decisions at the next level. *Second*, we exclude all dynamic features from training. This results in a similar impact for the cluster policy (10%) but significantly affects the policies constructed for the LITTLE, big, and FFT. *Next*, a similar trend is observed when PE availability times are excluded from the feature set. The accuracy is marginally higher since the other dynamic features contribute to learning the scheduling decisions. *Finally*, we remove a few task-related features, such as the downward depth, task, and application identifier. In this case, the impact is to the cluster policy accuracy since these features describe the node in the DAG and influence the cluster mapping.

As observed in Fig. 5, the average execution time of the workload significantly degrades when all features are not included. Hence, the chosen features help to construct effective IL policies, approximating the Oracle with over 99% accuracy in execution time.

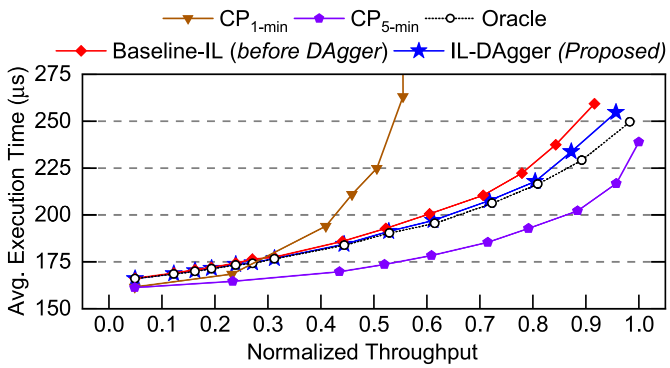


Fig. 6. Comparison of average job execution time between Oracle, CP solutions, and IL policies to schedule a workload comprising a mix of six streaming applications. IL scheduler policies with baseline-IL (*before DAgger*) and with IL-DAgger (*Proposed*) are shown in the comparison.

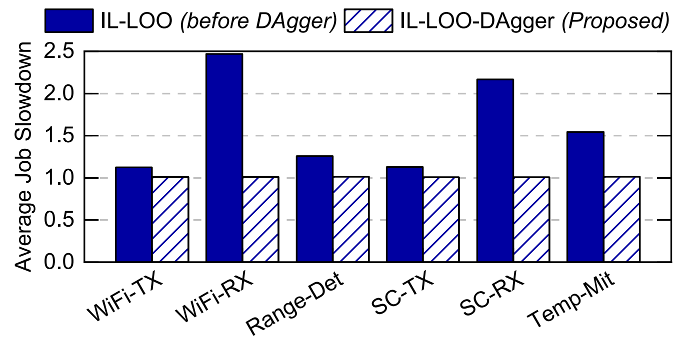


Fig. 7. Average slowdown of IL policies in comparison with Oracle for leave-one-out (LOO) experiments before and after DAgger (*Proposed*).

D. IL-Scheduler Performance Evaluation

This section compares the performance of the proposed policy to the ETF Oracle, $CP_{1\text{-}min}$, and $CP_{5\text{-}min}$. Since heterogeneous many-core systems are capable of running multiple applications simultaneously, we stream the frames in our application mix (see Table III) with increasing injection rates. For example, a normalized throughput of 1.0 in Fig. 6 corresponds to 19.78 frames/ms. Since the frames are injected faster than they can be processed, there are many overlapping frames at any given time.

First, we train the IL policies with all six reference applications and refer to this as the baseline-IL scheduler. IL policies suffer from error propagation due to the non i.i.d. nature of training data. To overcome this limitation, we use a data aggregation technique adapted for a hierarchical IL framework (IL-DAgger), as discussed in Section IV-C. A DAgger iteration involves executing the entire workload. We execute ten DAgger iterations and choose the best iteration with performance within 2% of the Oracle. If we fail to achieve the target, we continue to perform more iterations.

Fig. 6 shows that the proposed IL-DAgger scheduler performs almost identical to the Oracle; the mean average percentage difference between them is 1%. More notably, the gap between the proposed IL-DAgger policy and the optimal $CP_{5\text{-}min}$ solution is only 9.22%. We emphasize that $CP_{5\text{-}min}$ is included only as a reference point, but it has six orders of magnitude larger execution time overhead and cannot be used at runtime. Furthermore, the proposed approach performs better than $CP_{1\text{-}min}$, which is not able to find a good schedule within the 1-min time limit per decision. Finally, we note that the baseline IL can approach the performance of the proposed policy. This is intuitive since both policies are tested on known applications in this experiment. This is in contrast to the leave one out experiments presented in Section V-E.

Pulse Doppler Application Case Study: We demonstrate the applicability of the proposed IL-scheduling technique in complex scenarios using a pulse Doppler application. It is a real-world radar application, which computes the velocity of a moving target object. This application is significantly more complex, with $13-64\times$ more tasks than the other applications. Specifically, it consists of 449 tasks comprising 192 FFT tasks, 128 inverse-FFT tasks, and 129 other computations. The FFT and inverse-FFT operations can execute on the general-purpose cores and hardware accelerators. In contrast, the other tasks can execute only on the general-purpose cores.

The proposed IL policies achieve an average execution time within 2% of the Oracle. The 2% error is acceptable, considering that the application saturates the computing platform quickly due to its high complexity. Moreover, the CP-based approach does not produce a viable solution either with 1- or 5-min time limits due to the large problem size. For this reason, this application is not included in our workload mixes and the rest of the comparisons.

E. Illustration of Generalization With IL for Unseen Applications, Runtime Variations, and Platforms

This section analyzes the generalization of the proposed IL-based scheduling approach to unseen applications, runtime variations, and many-core platform configurations.

IL-Scheduler Generalization to Unseen Applications Using Leave-One-Out Experiments: IL, being an adaptation of supervised learning for sequential decision making, suffers from the lack of generalization to unseen applications. To analyze the effects of unseen applications, we train IL policies, excluding applications one each at a time from the training dataset [36].

To compare the performances of two schedulers S_1 and S_2, we use the job slowdown metric $\text{slowdown}_{S_1,S_2} = T_{S_1}/T_{S_2}$. $\text{Slowdown}_{S_1,S_2} > 1$ when $T_{S_1} > T_{S_2}$ [20]. The average slowdown of scheduler S_1 with respect to scheduler S_2 is computed as the average slowdown for all jobs at all injection rates. The results present an interesting and intuitive explanation of the average job slowdown in execution times for each of the LOO experiments.

Fig. 7 shows the average slowdown of the baseline IL (IL-LOO) and proposed policy with DAgger iterations (IL-LOO-DAgger) with respect to the Oracle. We observe that the proposed policy outperforms the baseline IL for all applications, with the most significant gains obtained for WiFi-RX and SC-RX applications. These two applications consist of a Viterbi decoder operation, which is very expensive to compute on general-purpose cores and highly efficient to compute on hardware accelerators. When these applications are excluded, the IL policies are not exposed to the corresponding states in

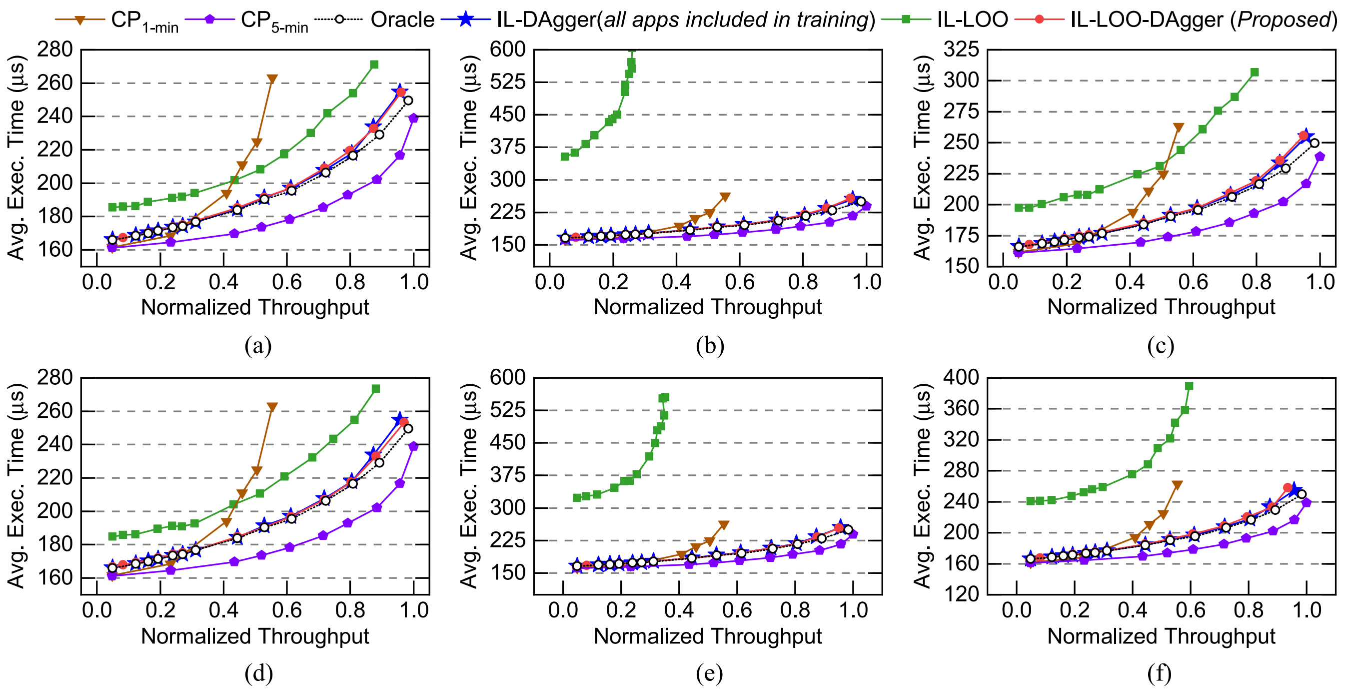


Fig. 8. Average execution time with Oracle, IL-DAgger (*all applications are included for training*), IL with one application excluded from training (IL-LOO), and finally, the LOO policy improved with DAgger (*Proposed* IL-LOO-DAgger) technique. The *excluded* applications are: (a) WiFi-TX; (b) WiFi-RX; (c) range detection; (d) single-carrier TX; (e) single-carrier RX; and (f) temporal mitigation.

the training dataset and make incorrect decisions. The erroneous PE assignments lead to an average slowdown of more than 2× for the receiver applications. The slowdown when the transmitter applications (WiFi-TX and SC-TX) are excluded from training is approximately 1.13×. Range detection and temporal mitigation applications experience a slowdown of 1.25× and 1.54×, respectively, for LOO experiments. The extent of the slowdown in each scenario depends on the application excluded from training and its execution time profile in different processing clusters. In summary, the average slowdown of all LOO IL policies after DAgger (IL-LOO-DAgger) improves to ~1.01× in comparison with the Oracle, as shown in Fig. 7.

Fig. 8(a)–(f) shows the average job execution times for the Oracle (ETF), baseline-IL, IL with LOO, and DAgger for IL with LOO policies for each of the applications. The highest number of DAgger iterations needed was 8 for SC-RX application, and the lowest was 2 for range detection application. If the DAgger criterion is relaxed to achieving a slowdown of 1.02×, all applications achieve the same in less than five iterations. A drastic improvement in the accuracy of the IL policies with few iterations shows that the policies generalize quickly and well to unseen applications, thus making them suitable for applicability at runtime.

IL-Scheduler Generalization With Runtime Variations: Tasks experience runtime variations due to variations in system workload, memory, and congestion. Hence, it is crucial to analyze the performance of the proposed approach when tasks experience such variations rather than observing only their static profiles. Our simulator accounts for variations by using a Gaussian distribution to generate variations in execution time [39]. To allow evaluation in a realistic scenario, all

TABLE VII
STANDARD DEVIATION (IN PERCENTAGE OF EXECUTION TIME) PROFILING OF APPLICATIONS IN ODROID-XU3 AND ZYNQ ZCU-102

Application	WiFi-TX	WiFi-RX	RangeDet	SC-TX	SC-RX	TempMit
Zynq ZCU-102	0.34	0.56	0.66	1.15	1.80	0.63
Odroid-XU3	6.43	5.04	5.43	6.76	7.14	3.14

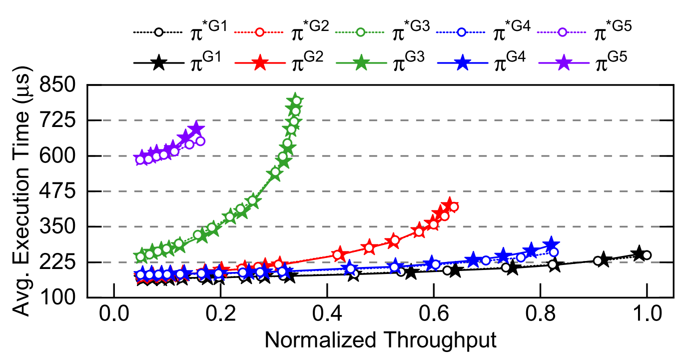


Fig. 9. IL policy evaluation with multiple many-core platform configurations. IL policies are trained with only configuration *G*1.

tasks in every application are profiled on big and LITTLE cores of Odroid-XU3, and, on Cortex-A53 cores and hardware accelerators on Zynq for variations in execution time.

We present the average standard deviation as a ratio of execution time for the tasks in Table VII. The maximum standard deviation is less than 2% of the execution time for the Zynq platform and less than 8% on the Odroid-XU3. To account for variations in runtime, we add a noise of 1%, 5%, 10%, and 15% in task execution time during the simulation. The IL policies achieve average slowdowns of less than 1.01× in all

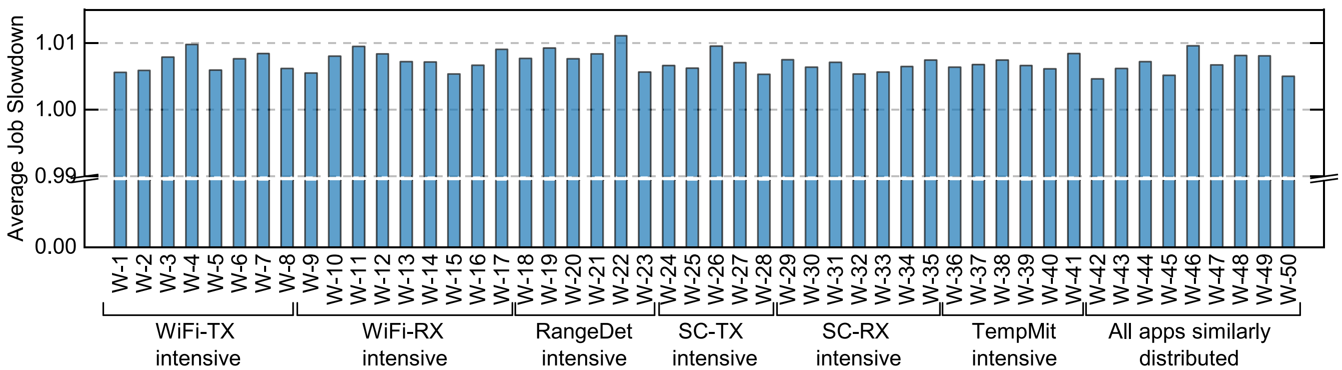


Fig. 10. Comparison of average job slowdown normalized with IL-DAgger (*Proposed*) policies against the Oracle for *50* different workloads. The slowdown of IL-DAgger policies are shown for workloads with different *intensities* of each application in the benchmark suite.

TABLE VIII
CONFIGURATION OF MANY-CORE PLATFORMS

Platform Config.	LITTLE PEs	big PEs	MatMul Acc. PEs	FFT Acc. PEs	Decoder Acc. PEs
G1 (Baseline)	4	4	2	4	2
G2	2	2	2	2	2
G3	1	1	1	1	1
G4	4	4	1	1	1
G5	4	4	0	0	0

cases of runtime variations. Although IL policies are trained with static execution time profiles, the aforementioned results demonstrate that the IL policies adapt well to execution time variations at runtime. Similarly, the policies also generalize to variations in communication time and power consumption.

IL-Scheduler Generalization With Platform Configuration: This section presents a detailed analysis of the IL policies by varying the configuration, i.e., the number of clusters, general-purpose cores, and hardware accelerators. To this end, we choose five different SoC configurations presented in Table VIII. The Oracle policy for a configuration $G1$ is denoted by π^{*G1}. An IL policy evaluated on configuration $G1$ is denoted as π^{G1}. $G1$ is the baseline configuration that is used for extensive evaluation. Between configurations $G1$–$G4$, we vary the number of PEs within each cluster. We also consider a degenerate case that comprises only LITTLE and big clusters (configuration $G5$). We train IL policies with only configuration $G1$. The average execution times of π^{G1}, π^{G2}, and π^{G3} are within 1%, π^{G4} performs within 2%, and π^{G5} performs within 3%, of their respective Oracles. The accuracy of π^{G5} with respect to the corresponding Oracle (π^{*G5}) is slightly lower (97%) as the platform saturates the computing resources very quickly, as shown in Fig. 9.

Based on these experiments, we observe that the IL policies generalize well for different many-core platform configurations. The change in system configuration is accurately captured in the features (in execution times, PE availability times, etc.), which enables us to generalize well to new platform configurations. When the cluster configuration in the many-core platform changes, the IL policies generalize well (within 3%) but can also be improved by using DAgger to obtain improved performance (within 1% of the Oracle).

F. Performance Analysis With Multiple Workloads

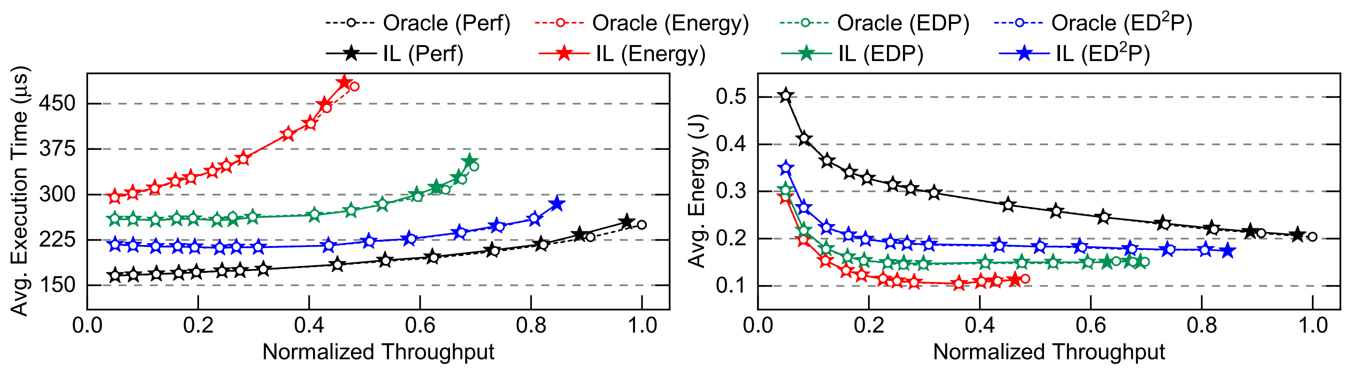
To demonstrate the generalization capability of the IL policies trained and aggregated on one workload (IL-DAgger), we evaluate the performance of the same policies on 50 different workloads consisting of different combinations of application mixes at varying injection rates, and each of these workloads contains 500 frames. For this extensive evaluation, we consider workloads each of which are intensive on one of WiFi-TX, WiFi-RX, range detection, SC-TX, SC-RX, and temporal mitigation. Finally, we also consider workloads in which all applications are distributed similarly.

Fig. 10 presents the average slowdown for each of the 50 different workloads (represented as W-1, W-2, and so on). While W-22 observes a slowdown of 1.01× against the Oracle, all other workloads experience an average slowdown of less than 1.01× (within 1% of Oracle). Independent of the distribution of the applications in the workloads, the IL policies approximate the Oracle well. On average, the slowdown is less than 1.01×, demonstrating the IL policies generalize to different workloads and streaming intensities.

G. Evaluation With Energy and Energy-Delay Objectives

Average execution time is crucial in configuring computing systems for meeting application latency requirements and user experience. Another critical metric in modern computing systems, especially battery-powered platforms, is energy consumption [23], [25]. Hence, this section presents the proposed IL-based approach with the following objectives: performance, energy, energy-delay product (EDP), and energy-delay^2 product (ED^{2}P). We adapt ETF to generate Oracles for each objective. Then, the different Oracles are used to train IL policies for the corresponding objectives. The scheduling decisions are significantly more complex for these Oracles. Hence, we use an RT of depth 16 (execution time uses RT of depth 12) to learn the decisions accurately. The average latency per scheduling decision remains similar for RT of depth 16 (~1.1 μs) on Cortex-A53.

Fig. 11(a) and (b) presents the average execution time and average energy consumption, respectively, for IL policies with different objectives. The lowest energy is achieved by the energy Oracle, while it increases as more emphasis is added to performance (EDP → ED^{2}P → performance), as



(a) (b)

Fig. 11. (a) Average execution time and (b) average energy consumption of the workload with Oracles and IL policies for performance, energy, energy-delay product (EDP), and energy-delay^2 product (ED^{2}P) objectives.

expected. The average execution time and energy consumption in all cases are within 1% of the corresponding Oracles. This demonstrates the proposed IL scheduling approach is powerful as it learns from Oracles that optimize for any objective.

H. Comparison With Reinforcement Learning

Since the state-of-the-art ML techniques [20], [21] do not target streaming DAG scheduling in heterogeneous many-core platforms, we implemented a policy-gradient-based RL technique using a deep NN (multilayer perceptron with four hidden layers with 32 neurons in each hidden layer) to compare with the proposed IL-based task scheduling technique. For the RL implementation, we vary the exploration rate between 0.01 and 0.99 and the learning rate from 0.001 to 0.01. The reward function is adapted from [21]. RL starts with random weights and then updates them based on the extent of exploration, exploitation, learning rate, and reward function. These factors affect the convergence and quality of the learned RL models.

Fewer than 20% of the experiments with RL converge to a stable policy and less than 10% of them provide competitive performance compared to the proposed IL-scheduler. We choose the RL solution that performs best to compare with the IL-scheduler. The Oracle generation and training parts of the proposed technique take 5.6 and 4.5 min, respectively, when running on an Intel Xeon E5-2680 processor at 2.40 GHz. In contrast, an RL-based scheduling policy that uses the policy gradient method converges in 300 min on the same machine. Hence, the proposed technique is 30× faster than RL. As shown in Fig. 12, the RL scheduler performs within 11% of the Oracle, whereas the IL scheduler presents average execution time that is within 1% of the Oracle.

In general, RL-based schedulers suffer from the following drawbacks: 1) need for excessive fine tuning of the parameters (learning rate, exploration rate, and NN structure); 2) reward function design; and 3) slow convergence for complex problems. In strong contrast, IL policies are guided by strong supervision eliminating the slow convergence problem and the need for a reward function.

I. Complexity Analysis of the Proposed Approach

In this section, we compare the complexity of our proposed IL-based task scheduling approach with ETF, which is used to construct the Oracle policies. The complexity of ETF is $O(n^2m)$ [14], where n is the number of tasks and m is the number of PEs in the system. While ETF is suitable for use in Oracle generation (offline), it is not efficient for online use due to the quadratic complexity on the number of tasks. However, the proposed IL-policy which uses RT has the complexity of $O(n)$. Since the complexity of the proposed IL-based policies is linear, it is practical to implement in heterogeneous many-core systems.

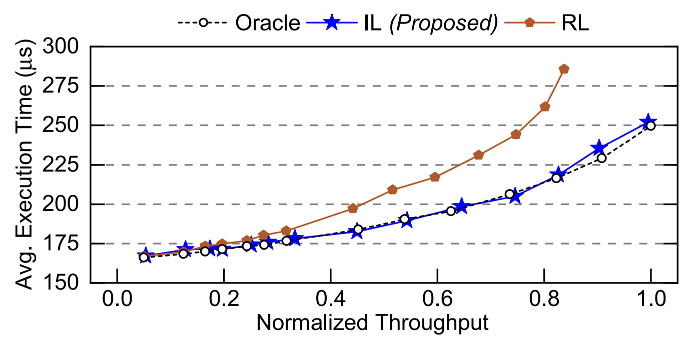


Fig. 12. Comparison of average execution time between Oracle, IL, and RL policies to schedule a workload comprising a mix of six streaming real-world applications.

VI. Conclusion

Efficient task scheduling in heterogeneous many-core platforms is crucial to improve the system performance but is very challenging due to its NP-hardness. In this work, we have presented an IL-based approach for task scheduling in many-core platforms executing streaming applications from wireless communications and radar systems. We have presented a hierarchical IL framework that learns from an Oracle to develop task scheduling policies to minimize the execution time of applications. The framework has been evaluated comprehensively with six domain-specific applications and analyzed the storage and latency overheads of the IL policies. We have shown that the IL policies approximate the Oracle better than 99%. The overhead of the policies is significantly low at 1.1 μs latency per scheduling decision and lower than the CFS (1.2 μs). Our IL policies achieve application execution times within 9.3% of optimal schedules obtained offline using CP.

Preliminary experiments in which we have used IL to bootstrap RL for task scheduling in heterogeneous many-core platforms show much faster convergence to optimal policies. We envision this work to initiate a new direction in scheduling studies with optimal Oracle generation and evaluation with applications from various domains.

Acknowledgment

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] *ZYNQ ZCU102 Evaluation Kit*. Accessed: 10 Apr. 2020. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

[2] "V12.8: User's manual for CPLEX," *Int. Bus. Mach. Corporat.*, vol. 46, no. 53, p. 157, 2009.

[3] A. M. Aji, A. J. Peña, P. Balaji, and W.-C. Feng, "MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL," *Parallel Comput.*, vol. 58, pp. 37–55, Oct. 2016.

[4] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.

[5] S. E. Arda *et al.*, "DS3: A system-level domain-specific system-on-chip simulation framework," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1248–1262, Aug. 2020.

[6] S. Baskiyar and R. Abdel-Kader, "Energy aware DAG scheduling on heterogeneous systems," *Clust. Comput.*, vol. 13, no. 4, pp. 373–383, 2010.

[7] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann, "Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler," in *Proc. IEEE Int. Conf. Appl. Specific Syst. Archit. Process.*, 2011, pp. 223–226.

[8] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira, "DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *Proc. IEEE Euromicro Conf. Parallel Distrib. Netw. Based Process.*, 2010, pp. 27–34.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[10] V. Goel, M. Slusky, W.-J. van Hoeve, K. C. Furman, and Y. Shao, "Constraint programming for LNG ship scheduling and inventory management," *Eur. J. Oper. Res.*, vol. 241, no. 3, pp. 662–673, 2015.

[11] D. Green *et al.*, "Heterogeneous integration at DARPA: Pathfinding and progress in assembly approaches," in *Proc. ECTC*, May 2018, pp. 1–40.

[12] Hardkernel. (2014). *ODROID-XU3*. Accessed: 20 Mar. 2020. [Online]. Available: https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3

[13] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[14] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM J. Comput.*, vol. 18, no. 2, pp. 244–257, 1989.

[15] 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.

[16] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.

[17] J. Mack, N. Kumbhare, U. Y. Ogras, and A. Akoglu. (2020). *User-Space Emulation Framework for Domain-Specific SoC Design*. [Online]. Available: https://arxiv.org/abs/2004.01636

[18] S. K. Mandal, G. Bhat, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "An energy-aware online learning framework for resource management in heterogeneous platforms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 25, no. 3, pp. 1–26, 2020.

[19] S. K. Mandal, G. Bhat, C. A. Patil, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "Dynamic resource management of heterogeneous mobile platforms via imitation learning," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 12, pp. 2842–2854, Dec. 2019.

[20] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.

[21] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 270–288.

[22] A. Mirhoseini *et al.*, "Device placement optimization with reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, vol. 70, 2017, pp. 2430–2439.

[23] K. Moazzemi, B. Maity, S. Yi, A. M. Rahmani, and N. Dutt, "HESSLE-FREE: Heterogeneous systems leveraging fuzzy control for runtime resource management," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5S, pp. 1–19, 2019.

[24] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 184, p. 4, Aug. 2009.

[25] B. K. Reddy, 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.–Sep. 2018.

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

[27] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of Constraint Programming*. Amsterdam, The Netherlands: Elsevier, 2006.

[28] R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2004, p. 111.

[29] A. L. Sartor, A. Krishnakumar, S. E. Arda, U. Y. Ogras, and R. Marculescu, "HiLITE: Hierarchical and lightweight imitation learning for power management of embedded SoCs," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 63–67, Jan.–Jun. 2020.

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

[31] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 1057–1063.

[32] V. Swaminathan and K. Chakrabarty, "Real-time task scheduling for energy-aware embedded systems," *J. Franklin Inst.*, vol. 338, no. 6, pp. 729–750, 2001.

[33] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[34] J. Ullman, "NP-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, 1975.

[35] M.-A. Vasile, F. Pop, R.-I. Tutueanu, V. Cristea, and J. Kołodziej, "Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing," *Future Gener. Comput. Syst.*, vol. 51, pp. 61–71, Oct. 2015.

[36] A. Vehtari, A. Gelman, and J. Gabry, "Practical bayesian model evaluation using leave-one-out cross-validation and WAIC," *Stat. Comput.*, vol. 27, no. 5, pp. 1413–1432, 2017.

[37] Y. Wang *et al.*, "Multi-objective workflow scheduling with deep-Q-network-based multi-agent reinforcement learning," *IEEE Access*, vol. 7, pp. 39974–39982, 2019.

[38] Y. Wen, Z. Wang, and M. F. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *Proc. Int. Conf. High Perform. Comput.*, 2014, pp. 1–10.

[39] C. Xian, Y.-H. Lu, and Z. Li, "Dynamic voltage scaling for multitasking real-time systems with uncertain execution time," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 8, pp. 1467–1478, Aug. 2008.

[40] T. Xiaoyong, K. Li, Z. Zeng, and B. Veeravalli, "A novel security-driven scheduling algorithm for precedence-constrained tasks in heterogeneous distributed systems," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 1017–1029, Jul. 2011.

[41] G. Xie, G. Zeng, L. Liu, R. Li, and K. Li, "Mixed real-time scheduling of multiple DAGs-based applications on heterogeneous multi-core processors," *Microprocess. Microsyst.*, vol. 47, pp. 93–103, Nov. 2016.

Anish Krishnakumar (Graduate Student Member, IEEE) received the B.Tech. degree in electrical and electronics engineering from the National Institute of Technology, Tiruchirappalli, India, in 2013, and the master's degree from the Birla Institute of Technology, Pilani, India, in 2016. He is currently pursuing the Ph.D degree in electrical engineering, Arizona State University, Tempe, AZ, USA.

He worked as a Physical Design Engineer with Qualcomm, San Diego, CA, USA, and as a Research Scientist with Intel Labs, Bengaluru, India.

Samet E. Arda (Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from Arizona State University (ASU), Tempe, AZ, USA, in 2013 and 2016, respectively.

He is currently an Assistant Research Scientist with ASU. His Ph.D. thesis focused on development of dynamic models for small modular reactors. His current research interests include system-level design and optimization techniques for scheduling in heterogeneous SoCs.

A. Alper Goksoy received the B.S. degree in electrical and electronics engineering from Bogazici University, Istanbul, Turkey, in 2018. He is currently pursuing the Ph.D. degree in electrical engineering with Arizona State University, Tempe, AZ, USA.

Sumit K. Mandal (Graduate Student Member, IEEE) received the dual (B.S.+M.S.) degrees from the Indian Institute of Technology, Kharagpur, Kharagpur, India, in 2015. He is currently pursuing the Ph.D. degree with Arizona State University, Tempe, AZ, USA.

His research interest includes analysis and design of NoC architecture, power management of multicore processors, and AI hardware.

Umit Y. Ogras (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2007.

From 2008 to 2013, he worked as a Research Scientist with the Strategic CAD Laboratories, Intel Corporation, Santa Clara, CA, USA. He is an Associate Professor with the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ, USA, where he is an Associate Director of WISCA Center.

Anderson L. Sartor (Member, IEEE) received the B.Sc. and Ph.D. degrees in computer engineering from the Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, in 2013 and 2018, respectively.

He is currently a Postdoctoral Researcher with Carnegie Mellon University, Pittsburgh, PA, USA. His primary research interests include embedded systems design, machine learning for energy optimization, SoC modeling, and adaptive processors.

Radu Marculescu (Fellow, IEEE) received the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 1998.

He is the Laura Jennings Turner Chair of engineering and a Professor with the Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX, USA. His current research focuses on developing ML/AI methods and tools for modeling and optimization of embedded systems, cyber–physical systems, and social networks.