Runtime Monitoring of ML-Based Scheduling Algorithms Toward Robust Domain-Specific SoCs

A. Alper Goksoy, *Member, IEEE*, Alish Kanani, Satrajit Chatterjee, and Umit Ogras, *Senior Member, IEEE*

***Abstract*—Machine learning (ML) algorithms are being rapidly adopted to perform dynamic resource management tasks in heterogeneous system on chips. For example, ML-based task schedulers can make quick, high-quality decisions at runtime. Like any ML model, these offline-trained policies depend critically on the representative power of the training data. Hence, their performance may diminish or even catastrophically fail under unknown workloads, especially new applications. This article proposes a novel framework to continuously monitor the system to detect unforeseen scenarios using a gradient-based generalization metric called coherence. The proposed framework accurately determines whether the current policy generalizes to new inputs. If not, it incrementally trains the ML scheduler to ensure the robustness of the task-scheduling decisions. The proposed framework is evaluated thoroughly with a domain-specific SoC and six real-world applications. It can detect whether the trained scheduler generalizes to the current workload with 88.75%–98.39% accuracy. Furthermore, it enables 1.1×–14× faster execution time when the scheduler is incrementally trained. Finally, overhead analysis performed on an Nvidia Jetson Xavier NX board shows that the proposed framework can run as a real-time background task.**

***Index Terms*—Domain-specific system on chip (SoC), imitation learning (IL), reinforcement learning (RL), resource management, robustness, runtime monitoring, task scheduling.**

I. Introduction

HETEROGENEOUS architectures integrate diverse computing elements, each tailored to optimize specific objectives, resulting in enhanced performance across various optimization fronts. Among these architectures, domain-specific systems on chip (SoCs) are meticulously designed to excel in particular domains, such as augmented/virtual reality, autonomous driving, and telecommunication [1], [2]. They maximize energy efficiency by integrating domain-specific hardware accelerators while supporting general-purpose computing by including general-purpose cores [3], [4], effectively blending adaptability and efficiency. In the context of scheduling, the NP-complete nature of the task scheduling problem poses significant challenges to traditional algorithms as the number of processing elements (PEs) and tasks increase due to the concurrent execution of multiple applications [5], [6]. This challenge has recently led researchers to develop machine learning (ML)-based task scheduling and other dynamic resource management (DRM) techniques [7], [8], [9], [10], [11], [12].

Manuscript received 12 August 2024; accepted 13 August 2024. Date of current version 6 November 2024. This work was supported by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under Agreement FA8650-18-2-7860. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: A. Alper Goksoy.)*

A. Alper Goksoy, Alish Kanani, and Umit Ogras are with the Department of Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, WI 53706 USA (e-mail: agoksoy@wisc.edu).

Satrajit Chatterjee is with the Research Division, Kepler AI, Palo Alto, CA 94303 USA.

Digital Object Identifier 10.1109/TCAD.2024.3445815

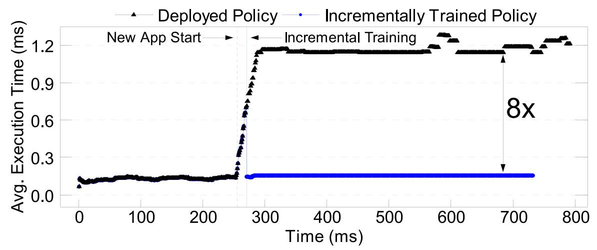

Fig. 1. Illustration of incremental training. The gray dotted line represents the arrival of the unknown application, whereas the green line represents when the policy is updated with the incrementally trained version. The deployed and incrementally trained policies are IL-based scheduling algorithms. The execution time is 8× lower after incremental training.

ML-based policies can deliver fast and high-quality decisions tailored to a particular domain by leveraging system, application, and task information as features. They are trained using diverse workloads representing a target domain to achieve this objective. Like any ML model, ML-based schedulers operate reliably within the confines of the datasets and applications used during training. Consequently, they may fail, or their performance may deteriorate when faced with new workload scenarios, especially those involving new applications [13], [14], [15]. Therefore, there is a strong need to monitor the scheduling decisions to detect nonrobust decisions.

Fig. 1 illustrates the variation in the execution time as an ML policy schedules streaming tasks to the PEs in an SoC. Initially, the SoC runs a mixture of applications that were used while training the ML scheduler. An unknown application replaces the previous mix at the instance marked by the gray dotted line. The average execution time begins to increase substantially after the unknown application arrives and converges to the execution time of the new application. A close inspection of the decisions reveals that the scheduler makes incorrect decisions. As a concrete example, it fails to recognize that one of the tasks in the new application could utilize a hardware accelerator PE. Due to the incorrect decisions, the execution time is 8× longer than a scheduler trained with this new application could achieve. Indeed, if one could detect the arrival of a new application class and incrementally train the scheduler, it could achieve significantly higher performance, as depicted by the green vertical line in Fig. 1. This example shows two crucial needs when an ML-based resource manager, such as a scheduler, is used in domain-specific SoCs. First, it must recognize the input changes (e.g., the arrival of a new application) to which the scheduler does not generalize. Second, an on-the-fly incremental training technique must adapt the scheduler to changes in data distribution over time while retaining knowledge from past data.

This article proposes a novel framework that achieves the following goals: 1) it monitors the actions of an ML-based scheduler; 2) it detects the input changes that deviate from the training data; and 3) it incrementally trains the ML policy to adapt to the new application. *To present a concrete implementation of the proposed framework*, we employ two runtime task schedulers, one trained using imitation learning (IL) and the other with reinforcement learning (RL). The proposed runtime monitoring is performed as a background task while an ML scheduler assigns incoming tasks to the PEs in the SoC. It first reads the features used by the ML scheduler, such as expected task execution times and PE states. Then, it computes the gradient of the trained ML policy and a coherence value using the gradient. When the gradient of the trained policy and information added by the new data samples are aligned, the coherence value is low, indicating that the current model generalizes well to the latest data samples. In contrast, when the latest data samples are not aligned with training, the coherence increases, indicating the need for retraining. When this happens, the proposed framework incrementally updates the ML policy, adapting it to new applications while retaining past information.

The efficacy of the proposed framework is assessed using six real-world communication and radio frequency (RF) applications running on a domain-specific SoC with sixteen PEs, including general-purpose big core clusters alongside fixed-point accelerators. Two instances of the proposed framework tailored to IL and RL schedulers monitor the SoC while a subset of the domain applications are launched. The proposed framework determines whether the IL scheduler generalizes to incoming data with over 98% accuracy. It misses only 0.59% of data points the scheduler fails to generalize. Furthermore, incrementally training the scheduler enables, on average, 4.21× faster execution time. The detection accuracy drops to 88.75% while monitoring an RL scheduler since RL policies rely on a reward function, a weaker feedback than the reference label available in IL. The proposed framework can still effectively flag when incremental training is needed and enable, on average, 1.32× faster execution. Finally, we implemented the proposed monitoring framework on the Nvidia Jetson Xavier NX board [16] to assess its runtime overhead. Since the proposed framework is not on the critical path, the execution time affects only 1) how fast poor scheduling decisions can be detected and 2) how frequently the monitoring process can be repeated. Even when all the steps of the proposed framework run sequentially, the worst-case execution times of the IL and RL instances are 83.74 and 117.53 ms, respectively. Hence, they can be effectively used as real-time background processes that run periodically, as detailed in Section V.

The main contributions of this article are as follows.

1) A framework that continuously monitors the system, identifying unforeseen tasks and incrementally training the model as needed.
2) Integration of a coherence-based detection mechanism within reinforcement and IL approaches.
3) Comprehensive experiments showcasing the effectiveness of the proposed framework in restoring performance.
4) Runtime overhead analysis with hardware measurements.

The remainder of this article is organized as follows. Sections II and III review the related work and present the background on the coherence metric and ML schedulers. Section IV describes the proposed framework and its application to IL/RL schedulers. Section V presents comprehensive experimental evaluations and hardware measurements. Finally, Section VI summarizes our conclusions.

II. Related Work

Domain-specific SoCs have gained traction in recent years following the demand for specialized processing and energy-efficient solutions. Recent work discussed these architectures and proposed accelerations frameworks across different application domains [17], [18], [19], [20]. Modern computing systems, including domain-specific SoCs, often rely on runtime heuristics for task scheduling [21], [22], [23]. Alongside heuristic schedulers, list-based schedulers [24], [25], [26], [27], [28], [29] have been proposed for task scheduling, aiming to optimize performance metrics at design time. However, one limitation of list-based schedulers is their inability to account for scenarios involving multiple streaming applications with varying initialization times. Furthermore, optimization-based schedulers, such as those utilizing integer linear or constraint programming techniques [30], [31], [32], aim for optimal decision making but suffer from infeasible runtime overheads due to computational complexity.

ML-based task schedulers have recently emerged as alternatives to conventional algorithms and heuristics [7], [8], [9], [10], [11], [12], [33], [34], [35], [36], offering reduced overhead while achieving near-optimal outcomes. They leverage various features, including performance counters, task, and application-related data, to make informed decisions. These features encompass a broad spectrum of metrics, ranging from task execution durations on diverse resources to communication latencies and resource availability, chosen strategically to optimize decision making. At the same time, a deep neural network or decision tree policy enables predictable execution time and runtime overhead optimization. These schedulers leverage various ML methods, such as support vector machine (SVM) [36], IL [10], [11], and RL [7], [8], [9], [33], [34], [35]. IL models, for instance, emulate the behaviors of complex schedulers impractical for runtime usage, showcasing efficiency by eliminating the need for exhaustive search or optimization algorithms. However, they are prone to sensitivity toward their training datasets and inherent biases from expert behaviors, rendering them vulnerable to unseen changes and generalization issues. In contrast, RL-based schedulers learn a policy that optimizes a performance metric [8], [9] or multiple metrics [34] by exploration. For instance, Decima [7] specializes in cluster-level scheduling for streaming applications using graph and deep neural networks. These schedulers also allow runtime adaptability, iteratively refining their weights to accommodate changes in response to evolving workload dynamics. They provide a significant advantage over static approaches, such as heuristics, particularly in swiftly changing environments inherent to domain-specific SoCs. Nonetheless, all these methods necessitate a monitoring framework to 1) confirm the generalization of the ML policy to new data encountered at runtime and 2) adapt the policies if needed.

Monitoring frameworks for ML models focus on detecting data and concept drifts. Data drift detection methods [37], [38] typically employ statistical models to assess whether the observed data deviate significantly from a reference distribution. In contrast, concept drift detection methods [39], [40], [41] focus on detecting shifts in the relationship between input and output using statistical and ML-based classifiers. However, these methods often have high computational complexity and execution times in the order of seconds, making them impractical for runtime applications, especially in scenarios with short task durations typical of domain-specific SoCs. Additionally, their ability to adapt to evolving data distributions may be limited due to inherent assumptions. In contrast, our proposed framework takes a different approach by leveraging changes in gradients to quantify generalization to new data without making assumptions about the input data. Moreover, recent work has explored the robustness of ML models using mixed-integer linear programming (ILP), resulting in runtime requirements ranging from seconds to minutes [42]. On the task scheduling problem, researchers discuss the robustness of task scheduling methods [43], using metrics, such as expected execution time and missed deadlines, often overlooking considerations related to generalizing to new applications. To the best of our knowledge, our proposed framework is the first runtime monitoring framework tailored for ML-based task scheduling on domain-specific SoCs.

III. Background on Coherence and ML Schedulers

This section first introduces coherence and its implications for the generalization of ML policies. Then, it overviews the use of ML schedulers in the domain-specific SoC context and describes the schedulers used in this work.

A. Background on the Coherence

Deep learning models trained with gradient descent have shown promising results in various fields, often demonstrating impressive generalization capabilities on unseen data. However, recent work notes that these networks theoretically have the capacity to memorize the training data. So, they could fail on any new input data [44], [45], [46]. Indeed, studies have shown that training with even entirely random data can lead to good training accuracy. Still, the models fail to generalize and exhibit poor accuracy on new data, indicating memorization. Hence, it is crucial to understand how gradient descent and the training process find solutions that generalize well among all possible solutions that fit the training data [45], [46].

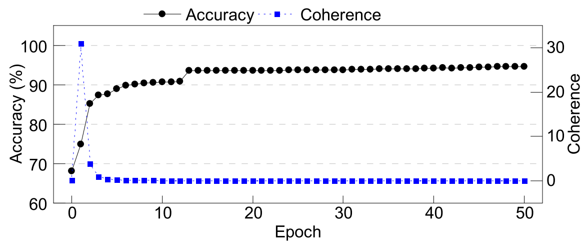

Fig. 2. Evolution of the coherence and accuracy throughout training. The examples exhibit stronger mutual support in the early epochs, resulting in higher coherence (the right y-axis). As training progresses, the expected gradient of samples approaches zero, indicating that the samples no longer provide significant assistance to one another. Consequently, coherence tends to diminish toward zero by the end of the training period.

One of the recent ongoing attempts to explain generalization in deep learning is “Coherent Gradients” [44], [47]. The core idea is that gradients calculated from similar training samples should be coherent, meaning they point in similar directions, allowing generalization (rather than memorization) to occur. In other words, the theory suggests that the interaction and reinforcement between gradients from different training examples lead the model to learn features that generalize well to unseen data.

Suppose z is a sample from a batch ($\mathcal{M}$) with $M = |\mathcal{M}|$ data samples. Further, let $l_z(w)$ denote the loss function for this sample, where w represents the trainable parameters of this model. One can compute the gradient for this sample as $g_z = [\nabla l_z](w)$. Chatterjee and Zielinski [44] quantified the coherence over these M samples using per-sample gradients. Specifically, they refer to the similarity between per-sample gradients as *coherence* and define it as

$$\alpha_M = M \cdot \frac{\underset{z\sim\mathcal{M}}{\mathbb{E}}\left[g_z\right] \cdot \underset{z\sim\mathcal{M}}{\mathbb{E}}\left[g_z\right]}{\underset{z\sim\mathcal{M}}{\mathbb{E}}\left[g_z \cdot g_z\right]}. \tag{1}$$

When the gradients (g_z) are perfectly aligned, the numerator and denominator will be equal, leading to maximum coherence (M). When all samples are fit, coherence will be zero, meaning the individual gradients will become zero.

During the initial training epochs, training data often shares many common features. This results in aligned gradients and, consequently, a higher coherence value. As training progresses and trainable parameters converge, new features become more specific, and the model tries to learn them individually. Consequently, the coherence value tends to decrease, as illustrated in Fig. 2.

Using Coherence for Runtime Monitoring: Gradients reinforce each other when learning takes place during the early training phases, leading to high coherence, as shown in the first few epochs of Fig. 2. After the model has learned what is common to all the samples and the samples have been fit (in a well-generalizing manner), the coherence drops and stabilizes to a low value. When the workload falls within the generalized set at runtime (not necessarily identical to the training data), its behavior resembles the end of the training phase illustrated in Fig. 2. Consequently, it is characterized by a low coherence value (like the latest data sample during training. However, if the new data samples deviate from the training data, their gradients would align, leading to a rise in the coherence value. Therefore, increasing coherence indicates that the model processes features from an application that it has not generalized yet. The coherence will remain high unless the ML policy, e.g., the scheduling algorithm, is incrementally trained. We leverage this observation in the proposed runtime monitoring framework. A low coherence for generalized workload indicates good performance, while a sustained high coherence suggests encountering new data that requires retraining. Using a smaller sample size of M would result in a smaller overall coherence range [zero to M in (1)], making the framework more susceptible to random noise during inference and negatively impacting accuracy. In contrast, a larger M would result in increased overhead, as discussed in Section V-D (see Tables II and III).

While Chatterjee and Zielinski [44] and Chatterjee [47] focused only on neural network models, we observe that the generalization theory using coherence can be used with any ML policy trained with gradient descent. Hence, we applied this framework to two scheduling algorithms: IL and RL. The IL model is trained with neural networks, while the RL model is trained with differential decision trees (DDTs), as elaborated in the following sections.

B. ML-Based Scheduling for Domain-Specific SoCs

Domain-specific SoCs are designed to deliver high performance when running applications from a target domain. A defining characteristic of these applications is processing streaming inputs for prolonged periods. For example, consider a domain-specific SoC designed for telecommunication. When the user starts the WiFi application, it processes received frames or transmits new ones for minutes, if not hours. Throughout this duration, the SoC continuously schedules the tasks comprising the WiFi transmitter and receiver chains. We envision that the proposed framework can run when a new application launches or periodically. The monitoring can repeat in the order of seconds or slower since there is no need to check the scheduler operations faster than that due to application lifetimes. Notable approaches of ML schedulers (IL- and RL-based training) are discussed next.

IL Schedulers: IL is an ML method where an agent learns a policy (π) that mimics the behavior of an expert (π^*) using the expert's actions. IL aims to minimize the error between the actions taken by the agent (a_t) and the expert (a_t^*). The expert actions (a_t^*) are collected offline and paired with corresponding states (s_t, a_t^*) for the agent to learn a policy (π_θ) [48]. However, this approach has limitations, as the behavior of the expert confines the agent's policy. To address this issue, the data aggregation (DAgger) algorithm [49] enhances the performance of IL by iteratively reinforcing incorrect decision–state pairs into the training set, thereby correcting deviations and improving overall performance.

In the context of task scheduling, IL-based models leverage offline training capabilities. For example, the training data is collected through executing various workloads under different system states to cover low to high congestion. During this process, an expert scheduler makes decisions for these workloads, with the data representing the system state (s_t) collected alongside the expert's policy decisions ($\pi^*(s_t)$). These system states and their corresponding action pairs are then utilized as features and target labels for supervised learning methods within the IL model. Subsequently, the learned IL policy (π_θ) is deployed for runtime decision making, replacing the expert policy (π^*). The expert may be a sophisticated heuristic or a constrained programming scheduler, which can make high-quality decisions but with a significant overhead.

RL Schedulers: Unlike IL schedulers, RL schedulers do not require an expert scheduler to guide the policy toward optimal behavior [8], [33], [34]. During training, the agent interacts with the environment by taking action (a_t) based on the current state (s_t), such as expected task execution and earliest PE availability times. For each action, the environment gives the agent a reward (r_t) that reflects how well the action aligns with the performance objectives, such as minimizing the execution time.

RL training algorithms commonly use actor–critic architectures, where the actor selects the actions (a_t), and the critic evaluates their expected outcomes. Both the actor and critic are continuously updated based on the feedback from the environment in terms of reward, allowing the agent to refine its policy (π_θ) over time [50]. The agent aims to optimize policy (π_θ) that takes actions to maximize the total reward over time. The state value function can be used to find expected rewards starting from an initial state following the policy. This value function ($V_\phi(s_t)$) can be approximated with a critic network with parameter (ϕ) that returns an expected value according to the state of the environment.

IV. Robust Monitoring of ML-Based Scheduling Algorithms

This section describes the proposed robust monitoring framework overviewed in Fig. 3. Section IV-A introduces the runtime monitoring component of the framework for detecting workload changes. Then, Sections IV-B and IV-C present the application of the proposed framework to IL and RL schedulers, respectively. Finally, Section IV-D discusses its applicability to other ML-based dynamic runtime management frameworks, and Section IV-E presents the incremental training approach used in our robust monitoring framework.

A. Robust Detection of Workload Changes

The first step of the proposed robust monitoring framework is continuous monitoring to detect the variations in the workload that can lead to incorrect decisions, as shown in Fig. 3. It is implemented as a background process to avoid any performance impact. Suppose the scheduler takes an action at time T_0. The system runs as usual by committing this action without interrupting the operation. At the same time, the background monitoring process is invoked to evaluate the quality of this action after the task is completed, as illustrated in Fig. 4. This evaluation is performed by calling a reference

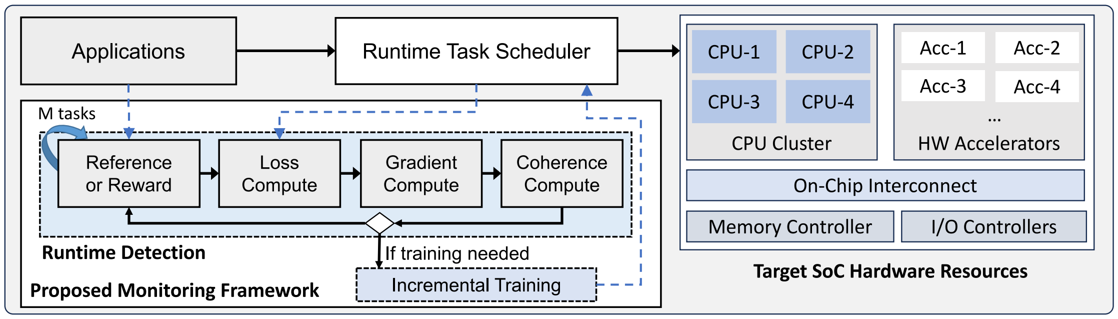

Fig. 3. Overview of the proposed framework that monitors the scheduler decisions and application features used for decision making. It is activated to compute the coherence of a batch with M samples. The primary steps are: 1) generating the reference scheduler action for IL or reward calculation for RL; 2) loss, gradient, and coherence calculations; and 3) an optional incremental training step triggered by the coherence value.

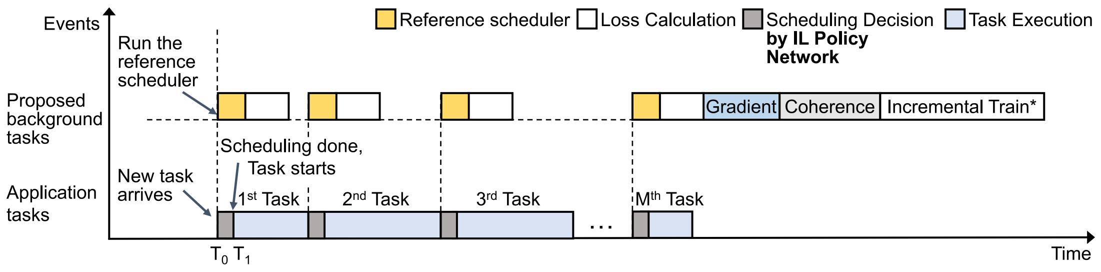

Fig. 4. Event diagram illustrating the proposed monitoring framework *for IL schedulers*. This figure shows the tasks in series for clarity, but multiple parallel tasks can be scheduled and monitored concurrently. While monitoring IL schedulers, a trustworthy (but slower) scheduler runs in the background to determine the correct action (a_t^*). This reference and actual policy actions (a_t) for a batch with M tasks are used for the loss, gradient, and coherence calculations (detailed in Algorithm 1). The incremental training step is executed if the framework decides the IL model policy (π_θ) should be updated.

scheduler with identical inputs and finding the reference action when monitoring an IL-based scheduler (detailed in Section IV-B). In the case of an RL-based scheduler, the reward received for this action is used to assess its quality (detailed in Section IV-C). Then, the outcome of this assessment is used to compute the gradient of the ML policy. Finally, the gradient is used to compute the coherence, as described in Section III-A. An insignificant change in the coherence value shows that the current ML policy handles the monitored application well. That is, the policy generalizes well to the monitored application. In contrast, a rise in coherence indicates new directions in the gradient, signifying the need to adopt the policy to address the changes in the workload. The specific details of the coherence calculation for IL- and RL-based schedulers are described in the following sections.

Background Process Overhead: The proposed monitoring and detection framework is implemented as a background process, as mentioned above and illustrated in Fig. 4. The system moves on with the current scheduling decision to avoid interruption since an incorrect decision only leads to transient performance degradation but not catastrophic failure. Hence, the proposed framework is not on the critical path. However, its overhead is still crucial since it determines how frequently the proposed framework can be called and the detection speed. Our hardware measurements indicate that the proposed monitoring and detection can be performed in the order of milliseconds, allowing frequent checks for the robustness of the ML policies. Given the types and composition of applications running on SoCs do not change in the order of seconds, the proposed framework enables runtime monitoring with negligible overhead, as detailed in Section V-D.

B. Application to IL-Based Schedulers

This section outlines the runtime detection framework employed for IL-based task scheduling frameworks. Fig. 4 illustrates the calculation steps in runtime monitoring for IL-based schedulers, while Algorithm 1 provides a detailed breakdown of these steps in the runtime detection process.

Once activated, the proposed monitoring framework processes the actions (a_t) taken by the policy (π_θ) for a sample size of M (Algorithm 1, lines 4 and 5). IL schedulers operate as supervised learning models, wherein an agent learns a policy (π_θ) from an expert's decision-making patterns to guide runtime scheduling decisions by generating actions (a_t) (line 6). Therefore, the runtime detection framework requires the reference targets (a_t^*) obtained by invoking the expert scheduler and collecting necessary performance metrics in the background to avoid execution time overhead (line 7). In this work, we employ a resource-intensive heuristic, the earliest task first (ETF) scheduler that loops through all ready tasks and PEs to choose the task assignment that minimizes the expected execution time [23]. The authors of the IL scheduler select ETF as the reference scheduler because its overhead grows quadratically, ranging from 0.3 to 8 ms. In contrast, the IL scheduler overhead grows linearly and enables nanosecond-level decisions [10], [34]. The reference actions are used to compute the loss function, denoted as $\mathcal{L}_\theta$ (line 8), in conjunction with the IL policy actions. We utilize the cross-entropy loss for $\mathcal{L}_\theta$. Then, the loss function is used to calculate the gradients g_z. Subsequently, we calculate the expected value of the gradient vector, $\mathbb{E}_{z\sim\mathcal{M}}[g_z]$, by adding the gradient vectors and $\mathbb{E}_{z\sim\mathcal{M}}[g_z \cdot g_z]$ by adding the dot product of the

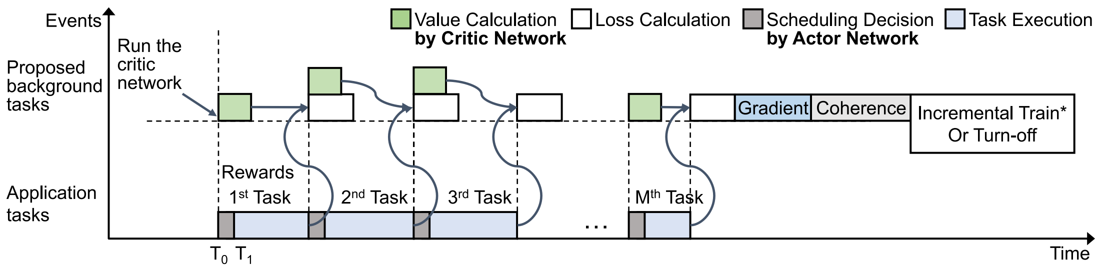

Fig. 5. Event diagram illustrating the proposed monitoring framework *for RL schedulers*. As in Fig. 4, the tasks are shown in series for clarity, but multiple parallel tasks can be scheduled and monitored concurrently. The proposed framework performs the loss calculation using the estimated value function ($V_\phi(s_t)$) from the critic network and rewards (r_t) from completed tasks. Then, the gradient is calculated for mini-batches (lines 13–17 in Algorithm 2), while coherence is calculated using batch coherence as given in line 19 in Algorithm 2. If the RL policy does not generalize to the current data points, it can be incrementally trained or turned off until the coherence reduces.

Algorithm 1 Detection Phase for the IL Scheduler

```
1: Input: Policy action set $\mathcal{M}$ with $M$ actions, Call period of the framework $P$
2: Output: Coherence value
3: Initialization: ML policy $\pi_\theta$ with parameters $\theta$, $\underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z] := 0$, $\underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z \cdot g_z] := 0$
4: Call robust monitoring framework every $P$ timeframe
5: while Total number of actions $< M$ do
6:     Get policy action $a_t$
7:     Get ground truth $a_t^*$ in the background using reference scheduler
8:     Calculate the loss function, $\mathcal{L}_\theta$ using $a_t^*$ and $a_t$ as CrossEntropyLoss($a_t^*, a_t$) [51]
9: end while
10: // gradients and coherence calculation
11: for sample from 1 to $M$ do
12:     Calculate gradients ($g_z$) using loss function $\mathcal{L}_\theta$ from current sample
13:     Update estimate $\underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z] := \underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z] + g_z$
14:     Update estimate $\underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z \cdot g_z] := \underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z \cdot g_z] + g_z \cdot g_z$
15: end for
16: Coherence $\alpha_M = M \cdot \frac{\underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z] \cdot \underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z]}{\underset{z\sim\mathcal{M}}{\mathbb{E}}[g_z \cdot g_z]}$
```

gradient vectors of weights, respectively. This process can be executed efficiently, with expected values computed using running sums without storing the gradients, either incrementally or collectively, at each monitoring session's conclusion. Finally, the coherence is computed using (1). It determines the coherence of gradients among all examples in the sample set $\mathcal{M}$, thereby detecting the unforeseen task scheduling scenarios that differ significantly from those encountered during training. We use the loss function for coherence instead of relying on accuracy because the accuracy metric misses differences when the policy generates the same actions with low confidence. Besides, accuracy remains relatively stable when a few new application instances are added to the workload mix. The loss value, in contrast, is sensitive to such variations.

C. Application to RL-Based Schedulers

This section presents the steps to apply our runtime monitoring framework to RL schedulers. During monitoring, the actor policy (π_θ) makes scheduling decisions (action a_t) for new tasks based on the SoC state (s_t). The selected PEs process the tasks as normal. As described in Section III-B, RL is an unsupervised learning method where both the actor and critic networks are trained during the offline training phase to maximize the reward defined as the negative execution time. Therefore, estimated state values ($V_\phi(s_t)$) from the trained critic network and rewards (r_t) expressed as the negative of the task execution times are used to calculate the loss function $\mathcal{L}_\theta$ required for the gradient calculation. As new tasks arrive, the trained critic network updates the state values in the background, as illustrated in Fig. 5. Upon task completion, rewards in terms of execution time are acquired from the PEs. These rewards and the state values are used to calculate the advantage function ($A(s_t, a_t)$) for the state–action pair, following the same equation as given in the PPO algorithm [50]:

$$A(s_t, a_t) = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{2}$$

where γ represents the discount factor and $V_\phi(s_{t+1})$ is the state value after completion of the task. The loss calculation during training also uses the ratio between the updated policy and the previous policy $\rho(\theta)$. Since the policy remains fixed during inference at runtime, the probability ratio $\rho(\theta)$ remains equal to one. Thus, policy loss $\mathcal{L}_\theta$ is given by the advantage function in (2) and used in Algorithm 2 (line 10)

$$\mathcal{L}_\theta = \rho(\theta) \cdot A(s_t, a_t); \;\; \rho(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} = 1. \tag{3}$$

Since this loss is not directly derived from the ground truth, the resulting gradient and coherence become noisy. To address this, we split the batch $\mathcal{M}$ (with $M = |\mathcal{M}|$ samples) into a set of $\mathcal{K}$ mini-batches (with $K = |\mathcal{K}|$, each of size M/K). Then, we use the average advantage within each mini-batch for gradient calculation. The coherence for each mini-batch and the overall batch coherence are calculated using [44, Th. 3] (lines 18 and 19 in Algorithm 2). This theorem ensures statistical equivalence of the per-sample coherence described in (1).

Algorithm 2 Detection Phase for the RL Scheduler

```
1: Input: Batch $\mathcal{M}$, mini-batch $\mathcal{K}$, Framework activation period $P$
2: Output: Coherence value
3: Initialization: Learned policy $\pi_\theta$ with parameters $\theta$, Trained critic $V_\phi$ with parameters $\phi$, $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] \coloneqq 0$, $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z] \coloneqq 0$
4: Call robust monitoring framework every $P$ seconds
5: // Loss calculation
6: while Total number of actions $< M$ do
7:     Get policy actions $a_t$
8:     Get value estimates $V_\phi(s_t)$ using trained critic $V_\phi$
9:     Get rewards ($r_t = -$task execution time) from the system for the completed tasks
10:    Calculate the advantage function as loss, $\mathcal{L}_\theta$ using $V_\phi(s_t)$ and $r_t$ for all the actions as given in the equation (2)
11: end while
12: // Mini-batch gradients and coherence calculation
13: for mini-batch from 1 to $K$ do
14:    Calculate gradients ($g_z$) using average $\mathcal{L}_\theta$ from current mini-batch
15:    Update estimate $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] \coloneqq \mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] + g_z$
16:    Update estimate $\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z] \coloneqq \mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z] + g_z \cdot g_z$
17: end for
18: Mini-batch Coherence $\alpha_K = \frac{\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z] \cdot \mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z]}{\mathop{\mathbb{E}}_{z\sim\mathcal{K}}[g_z \cdot g_z]}$
19: Coherence $\alpha_M = M \cdot \frac{\alpha_K}{K-(K-1)\cdot\alpha_K}$
```

D. Application to Other ML-Based DRM Algorithms

The proposed framework uses a loss function and gradients to compute the coherence for detecting workload changes. Hence, it can be applied to monitor the decisions of other ML-based schedulers and DRM algorithms that allow runtime gradient calculation. For example, dynamic thermal and power management techniques determine the optimal voltage–frequency pairs for computing cores to meet thermal constraints while preserving performance. These algorithms encompass a variety of approaches, including IL [12], [52], [53] and RL [54], [55] methods. Our framework can work with all these methods to prevent unexpected behavior due to a mismatch between training and runtime inputs. For example, Sartor et al. [12] employed a hierarchical IL framework featuring distinct policies for frequency, core selections, and execution time predictions. Our framework can effectively monitor these policies, utilizing the described policy and expert actions outlined in the study to compute loss and subsequent steps. It can ensure robust performance across various scenarios. In summary, our framework offers monitoring support for any runtime ML-based framework that utilizes gradient-based optimizations, ensuring robustness and reliability across various dynamic runtime management applications.

E. Response to Significant Workload Changes

The final stage of the proposed runtime monitoring framework is the response to significant changes in the workload. The objective of this stage is to detect the substantial changes in the workload to which the trained model does not generalize. We compare this detection's coherence (α_M) against a threshold (τ) learned during training. For this purpose, we employ a simple classifier, such as an SVM, to learn the threshold that maximizes the detection accuracy. Coherence values lower than the threshold ($\alpha_M < \tau$) indicate that scheduler decisions are trustworthy and no intervention is required. In contrast, larger coherence values ($\alpha_M > \tau$) require action since they indicate that the model is not generalizing well to coming samples.

There are two possible responses when a significant workload change deems the scheduler unreliable. The most straightforward remedy is to fall back to a traditional algorithm (e.g., the reference scheduler) for actions. The ML scheduler decisions can be monitored during this time until the coherence value moves below the threshold. In this way, the SoC will be protected from unreliable ML decisions. The second option is incrementally training the scheduler to adapt to workload changes, which will conserve the advantages of using ML schedulers. The rest of this section describes how IL and RL schedulers respond when a significant change is triggered.

IL Scheduler: The monitoring process for the IL scheduler involves a reference scheduler whose decisions are used to compute the loss function, as shown in Fig. 4. This implies that the reference actions (a_t^*) for the samples received during monitoring (s_t) are readily available, making incremental training a practical option. To this end, we utilize these state–action pairs (s_t, a_t^*) to *incrementally* train the IL policy. We also measure the overhead of this training process. It takes approximately 2 ms per epoch for incremental training of the IL scheduler on the Nvidia Jetson Xavier NX board [16], a timeframe negligible compared to the domain-specific application lifecycle. The execution of the tasks continues with the previous policy to ensure continuity during this process. Subsequently, the IL scheduler starts using the new policy ($\hat{\pi}$), leading to significant benefits detailed in Section V.

RL Scheduler: Unlike the IL scheduler case, RL training is unsupervised, learning from rewards (task execution time) provided by the environment (PEs in SoC) rather than a reference. Hence, the corresponding monitoring process does not involve a reference scheduler that gives correct actions. RL scheduler can be trained at runtime using the rewards received at the end of task executions. However, the RL scheduler can make poor decisions during this time, potentially impacting the runtime of tasks it executes. If this degradation in performance is acceptable, the policy can be incrementally updated during the operation. Otherwise, turning it off may be preferable while the coherence value is above the threshold. One can also train an RL policy offline incrementally and update the scheduler if the workload changes are permanent.

V. Experimental Evaluation

This section presents the experimental evaluations of our framework. We detail the experimental setup in Section V-A.

Section V-B discusses the results obtained for the IL scheduler, while Section V-C presents the findings for the RL scheduler. Lastly, Section V-D discusses the runtime overhead of our proposed framework for both schedulers.

A. Experimental Setup

Domain-Specific SoC Configuration: The selection of the SoC configuration is tailored to the requirements of domain-specific applications. Our simulation configuration consists of sixteen PEs, comprising eight general-purpose cores utilizing the Arm big.LITTLE architecture. These cores include four Arm A57 performance and four Arm A53 low-power cores. Additionally, the SoC incorporates eight fixed-function accelerators designed for handling intensive tasks: four accelerators dedicated to Fast Fourier Transform, two for Viterbi decoding, and two for matrix multiplication. This configuration is designed based on the specific demands of the target domain applications and the computational intensities of the tasks in these applications.

Domain Applications: The evaluation of the runtime monitoring framework encompasses six real-world applications spanning the telecommunication and RF domains. These applications include WiFi transmitter, WiFi receiver, temporal mitigation, lag detection, single-carrier transmitter, and single-carrier receiver. The number of tasks for these applications varies from 7 to 34. They are mixed into the workloads spanning from lower to higher intensity levels, ensuring comprehensive coverage, as detailed in [56] and [57].

Simulation Framework: We evaluated our runtime monitoring framework using an open-source discrete event-based simulator, DS3 [56]. This simulator has been validated against two commercial SoCs, the Odroid-XU3 [58] and the Zynq Ultrascale+ ZCU102 [59]. It enables target application simulations using different schedulers, providing a flexible environment for efficiently implementing new scheduling policies and our framework. Each simulation duration is around 2 s, resulting in a dynamic variation in the number of applications running, ranging from 4000 to 40 000 instances, and an average task count ranging from 50 000 to 500 000.

B. Results Obtained With IL Scheduler

This section delves into the experimental evaluations with IL schedulers. The policy adopted for the IL scheduler comprises a neural network architecture consisting of three dense layers, each with 32 neurons. The neural network is trained using Python and TensorFlow libraries, achieving accuracies ranging between 96.1% and 98.3% against the reference scheduler, ETF [23]. The policy leverages a combination of system, application, and task-level data as features to determine the cluster assignment. Then, the task is assigned to the PE within the chosen cluster, which is either available or set to become available first. We first illustrate the proposed framework as a function of time using single- and multi-application use cases. Then, we summarize our exhaustive accuracy evaluations.

Single Application Use Case Illustration: This illustrative example starts running a domain application represented in

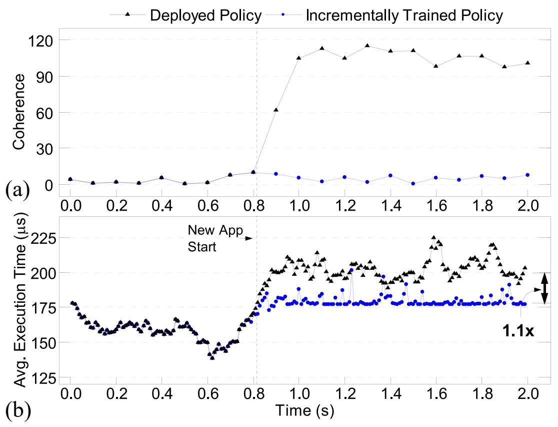

Fig. 6. Illustration of (a) the coherence value and (b) the average execution time for the runtime monitoring framework with IL scheduler using two applications. The first application (WiFi transmitter) runs until the black dotted line. After that, it is replaced by a new application (lag detection) not represented in the training data.

the training dataset. As the test samples from this application arrive at runtime, the coherence value remains low, as shown in Fig. 6(a). We emphasize that the training and test samples are different except that they come from the same application. Since the execution time varies significantly over time, it cannot be used alone to identify significant workload changes. After running for 0.8 s, this application is replaced with a new one not represented in the training dataset. The proposed monitoring framework successfully captures this change, as shown in Fig. 6(a). The coherence increases quickly, indicating the unalignment between the trained policy and the impact of new data samples. If we do not take action (e.g., incrementally train or turn off the scheduler), the coherence remains high, and execution time varies around 200 us. In contrast, incremental training (explained in Section IV-E) successfully adapts the policy to the new application, as revealed by the coherence plot in Fig. 6(a). Furthermore, the execution time reduces on average by 10%. Finally, we note that the incrementally trained policy still runs the first application optimally, i.e., the coherence remains low if it resumes running. This part is not plotted for brevity.

Multiple Application Use Case Illustration: The second example starts running a mix of five applications represented in the training dataset. The coherence computed at runtime remains low, as expected, as shown in Fig. 7(a). After running them for about 0.25 s (marked by a dotted line), these applications halt, and a previously unseen application starts running. The proposed framework successfully tracks the increased coherence after this change. As in the previous example, an elevated coherence indicates that new data samples require updating the policy parameters. If the policy is not updated, coherence remains high, and the execution time rises to about 2.5 ms, as shown in Fig. 7(b). In contrast, the proposed incremental training rapidly reduces coherence to its original value. Moreover, it achieves a remarkable performance boost (12× lower execution time) compared to no training.

Accuracy and Performance Summary: We prepared extensive use case scenarios similar to those illustrated above. They start running a randomly selected subset of application

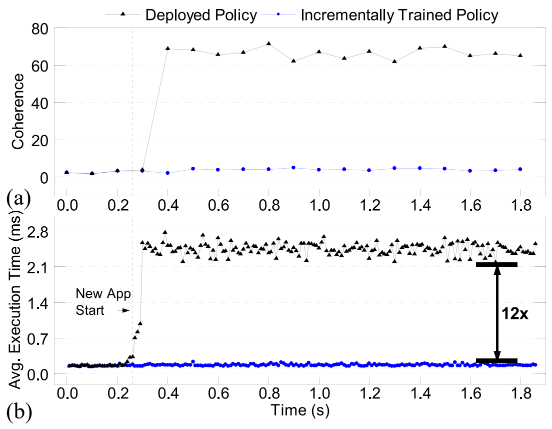

Fig. 7. Illustration of (a) the coherence value and (b) the average execution time for the runtime monitoring framework with IL scheduler using a workload composed of six applications. Five out of six domain applications run concurrently until the black dotted line. After that, the sixth application (single-carrier receiver) is introduced.

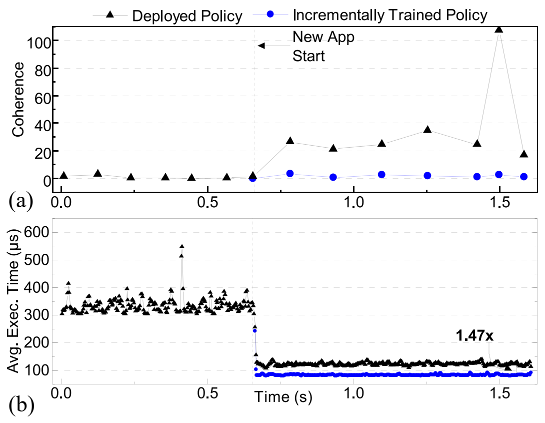

Fig. 8. Result of (a) the coherence value and (b) the average execution time for the runtime monitoring framework with RL scheduler using a workload comprising instances from two applications. Until the black dotted line, the first (WiFi receiver) application instances are in the system. After that, the new application (temporal mitigation) is introduced.

TABLE I
ACCURACY AND EXECUTION TIME IMPROVEMENTS FOR RUNTIME MONITORING FRAMEWORK ON IL AND RL SCHEDULERS ($M = 1024$)

Scheduler	Monitoring Accuracy	False Negative	False Positive	Avg. Exec. Time Improvement
IL	98.39%	0.59%	1.02%	4.21×
RL	88.75%	6.20%	5.05%	1.32×

mixes and then randomly change the applications. Single application examples start running one of the six domain applications randomly and switch to another one after a random duration. We repeated these simulations at different intensities and obtained 1221 batches. 663 out of these 1221 points indicate inputs the ML scheduler does not generalize. The multiapplication experiments start running five out of six applications concurrently (leaving one out). Then, the missing application replaces the original one. These experiments are also repeated to obtain 13 767 batches. 8585 of these 13 767 batches correspond to input the ML scheduler does not generalize. Overall, the combined data set comprises 14 988 batches, of which 9248 batches indicate a significant input change.

The proposed runtime monitoring framework identifies whether the IL scheduler generalizes to new data points correctly 98.39% of the time, as summarized in Table I (the first row). False positives in Table I occur when our monitoring framework detects activity despite there being no new application. False negatives, on the other hand, occur when a nongeneralized application appears but is not detected by our monitoring framework. The IL scheduler's false positive rate is only 1.02%, which means it incorrectly flags a change, although the scheduler generalizes well to the input. More importantly, it almost never misses a significant input change (0.59%). Finally, the proposed framework enables 4.21× lower execution time on average when incremental training is performed. These results present that the proposed framework can effectively detect when the IL scheduler makes unreliable decisions and adapt the scheduler to achieve substantial benefits.

C. Results Obtained With RL Scheduler

This section discusses the performance of the proposed runtime monitoring framework when applied to the RL scheduler. The RL scheduler comprises an actor policy for decision making and a critic network for evaluation. The actor policy is responsible for scheduling decisions and is situated on the critical path of the main process. Therefore, it is implemented using a DDT, enabling scheduling in approximately 0.18 μs on the Nvidia Jetson Xavier NX board [16]. Once a scheduling decision is made, the main process executes tasks on PEs while our framework concurrently monitors these decisions in the background. Actor–critic policies utilize features encompassing task, application, and SoC-level information (similar to the IL scheduler described in Section V-B). These policies are trained using PyTorch with an OpenAI Gym environment [60].

We conducted leave-one-out experiments with all six domain applications for a comprehensive performance evaluation. The RL scheduler generalizes well to five of these applications, even when they are excluded from training. However, it performs poorly when running the last application (temporal mitigation), indicating that the RL scheduler does not generalize to this application and does not make robust decisions. Our monitoring framework confirms this observation, as coherence values remain consistently low even with the arrival of new applications, except for "temporal mitigation," where coherence increases when the RL policy schedules it. Each batch (M) used in monitoring comprises 1024 samples, each divided into eight mini-batches (K) with 128 samples. The rest of this section summarizes the results.

Single Application Use Case Illustration: Like the IL experiments in Section V-B, this scenario begins by executing a single application represented in the training dataset. The coherence computed by the proposed framework is low during this time, as illustrated in Fig. 8(a). Subsequently, it is replaced with a new application, to which the RL scheduler does not generalize. The coherence value sharply increases from approximately zero to over 20 following this change, as shown in Fig. 8(a). Correspondingly, there is a sudden decrease in

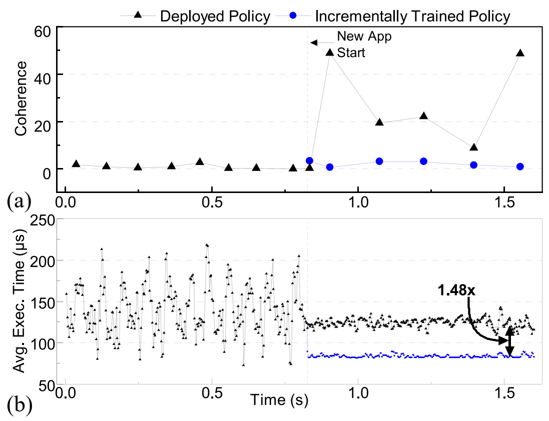

Fig. 9. Result of (a) the coherence value and (b) the average execution time for the runtime monitoring framework with RL scheduler using a workload comprising instances from six applications. Until the black dotted line, five out of six application instances are present in the system. After that, the sixth application (temporal mitigation) is introduced.

execution time, as shown in Fig. 8(b). This decrease occurs because the new application has inherently shorter execution times. However, it is essential to note that a shorter execution time does not necessarily indicate that the RL scheduler has successfully generalized to the new application. As discussed in Section IV-E, the policy undergoes incremental offline training to adjust to a new application. The policy retains its performance for the initial application while being optimized for the new one. With the incrementally trained policy, the average execution time decreases by 1.47×

Multiple Application Use Case Illustration: The multiple application begins running five out of six domain applications, like the IL example. Coherence during this time is low since these applications are represented during training, as shown in Fig. 9(a). Then, a new application not represented in the training replaces the original mix. The proposed framework successfully captures this change, as indicated by the abrupt increase in coherence after the dotted line. The execution time varies widely during the initial period, but it has a similar average value with lower variation after the new application is launched. This behavior shows that execution time is not a reliable indicator of the scheduler's generalizability. Finally, Fig. 9(b) shows training the scheduler incrementally adapts it to the new application, enabling 1.48× lower execution time.

Accuracy and Performance Summary: We conclude this section by summarizing the accuracy and performance benefits of the proposed framework with RL schedulers. Like the IL experiments, we conducted comprehensive simulations with varying application loads. For single-application examples, we assessed the monitoring framework across a total of 3685 batches (each comprising $M = 1024$ tasks). The RL scheduler fails to generalize to 666 of these batches coming from the new application. In the case of multiple applications, we evaluated our monitoring framework for over 1168 batches, with 161 batches indicating a lack of generalization. Overall, we evaluated our monitoring framework for 4853 batches. As discussed previously, the RL scheduler demonstrates inherent generalization to five applications, resulting in fewer instances of nongeneralized cases than the IL scheduler.

TABLE II
MONITORING FRAMEWORK OVERHEAD FOR IL SCHEDULER ON NVIDIA JETSON XAVIER NX BOARD [16]

Batch size (M)	Reference scheduler (ms)	Loss (ms)	Gradient (ms)	Coherence (ms)	Total overhead (ms)
128	3.20	1.25	7.84	0.09	12.38
256	6.40	1.92	13.70	0.21	22.23
512	12.80	3.69	26.48	0.29	43.26
1024	**25.60**	**7.50**	**50.24**	**0.40**	**83.74**

Table I summarizes the proposed monitoring framework's accuracy and performance benefits. It can determine whether the scheduler generalizes to the new inputs or not with 88.75% accuracy. Closing inspection reveals a 6.2% false negative rate, i.e., the frequency of failing to detect a new application. Similarly, it incorrectly flags the lack of generalization to a new application (false positive) for 5.05% of the batches. The values are lower than those obtained with the IL scheduler since there is no ground truth label during the training and monitoring of the RL scheduler. Hence, it only relies on the reward signal, a weaker indication of correctness than the ground truth. The accuracy can be improved by checking more than one batch before flagging a lack of generalization at the expense of more significant overhead. This optimization is one of the potential future research directions. Finally, when the proposed framework identifies a new application, incremental training provides, on average, a 1.32× lower execution time.

Application to Other Schedulers: The proposed framework can also be used with other scheduling algorithms besides the IL and RL schedulers considered so far. For example, Decima [7] is a graph neural network-based job scheduling algorithm targeting data clusters for streaming applications. The authors show that when the model is trained with low throughput workloads, the model poorly generalizes to high throughput workloads, leading to a 1.6× higher average execution time. This example shows a great use case for our monitoring framework. As it successfully detects the changes in the applications, specifically an unseen level of throughput in this case, it can trigger the system to take proper action, such as incremental training. Indeed, when the incrementally trained version performs similarly to a system trained with both high and low throughput workloads, the proposed framework can achieve 1.37× faster execution time. Therefore, our framework is effective for a wide range of hardware platforms and models that utilize gradient descent optimization.

D. Overhead Analysis

The proposed monitoring framework is not on the critical path since it operates in the background. An overhead analysis is still helpful since it helps determine how frequently the monitoring can be triggered. As described in Section III-B, this work considers domain-specific SoC, where applications continuously process streaming inputs for extended durations after launch. The proposed monitoring framework does not need to run continuously. It can be triggered 1) when a new application launches or 2) periodically while sleeping most of the time. The overhead analysis in this section summarizes the execution overhead as a function of the batch size (M). These values determine the shortest possible monitoring period.

TABLE III
MONITORING FRAMEWORK OVERHEAD FOR RL SCHEDULER ON NVIDIA JETSON XAVIER NX BOARD [16]

Batch size (M)	Value estimates (ms)	Loss (ms)	Gradient (ms)	Coherence (ms)	Total overhead (ms)
128	0.02	12.22	20.09	0.10	32.42
256	0.04	24.11	20.20	0.10	44.45
512	0.08	49.01	23.83	0.10	73.03
1024	**0.17**	**92.30**	**24.96**	**0.10**	**117.53**

Table II summarizes the overhead for monitoring the IL scheduler when running on the Nvidia Jetson Xavier NX board [16]. The most time-consuming step is the gradient calculation, varying from 7.84 to 50.24 ms as the batch size grows from 128 to 1024. The second largest contributor is running the reference scheduler, which takes 3.2–25.6 ms. We emphasize that different components of the monitoring framework can be pipelined. For example, the reference scheduler can start running for the next task after the loss calculation begins. Hence, the total execution time in Table II is a loose upper bound. Regardless, our measurements show that the entire monitoring process takes 83.74 ms, even for the largest batch size used in our experiments, highlighted in the table. A smaller batch size can be employed to speed up the monitoring process at the expense of accuracy. The last row (bold) highlights the setting in our experiments, while evaluations shown for all batch sizes are listed in Table II. This means the monitoring can be repeated in the background with this period if needed. However, in practice, we expect a more extended period, on the order of seconds, since the application composition in the target SoCs rarely changes.

Table III summarizes the monitoring and detection overhead for RL schedulers as a function of the batch size (M). The loss and gradient calculations dominate the total execution time for RL schedulers. The loss takes longer than those for the IL scheduler since loss for IL is the mean squared error, but RL requires solving (2). As in the IL scheduler, the value estimate, loss, and gradient calculations can be pipelined. In the worst case, when all steps are performed sequentially, the total execution time varies from 32.42 to 117.53 ms as the batch size grows from 128 to 1024. The last row (bold) highlights the setting in our experiments. Like in the IL case, all batch sizes in the table lead to effective monitoring. Hence, the proposed framework can run as a real-time background task to monitor RL schedulers.

As detailed in Section IV-B, coherence can be calculated using a running sum for $\mathbb{E}_{z\sim\mathcal{M}}[g_z]$ and $\mathbb{E}_{z\sim\mathcal{M}}[g_z \cdot g_z]$ vectors. This approach ensures that the memory requirement does not scale with the batch size M, meaning the memory requirement remains constant, or $O(1)$. Using onboard sensors, we also monitored power utilization and temperature changes on the Jetson Xavier NX. We observed that it consistently consumes less than 1 W of power. So, our monitoring framework requires a maximum of 83.74 mJ for the IL case and 117.53 mJ for the RL case. Due to this very low energy consumption, we observed only a 3 °C–4 °C increase in temperature. This analysis shows that our monitoring framework has a negligible impact compared to the application running on the target SoCs.

VI. CONCLUSION

ML algorithms are increasingly used for runtime decision making in SoC. For example, offline-trained deep neural networks and DDT policies schedule tasks to PEs. Like all ML models that critically depend on training data, these schedulers can exhibit unpredictable behavior when the runtime inputs deviate significantly from the training. Hence, monitoring their robustness and protecting the system from adverse effects is crucial.

This article introduces a novel runtime monitoring framework for domain-specific SoCs. The proposed framework uses the new input samples and policy gradient to compute a coherence metric. Low coherence indicates agreement with the trained policy and new inputs, while elevated coherence shows that the scheduling decisions are unreliable. We also discuss how the policies can be incrementally trained or turned off until they become reliable. Extensive evaluations show that the proposed framework can detect when the scheduler decisions are unreliable with 88.75%–98.39% accuracy. Our experiments also reveal that 1.1×–14× lower execution time is possible by incremental retraining.

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. Dr. Umit Ogras has a financial interest in DASH Tech IC, which is developing products related to the research described in this article. This relationship has been reviewed and approved by the University of Wisconsin–Madison based on its outside activities policies.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019.
[2] D. Green, "Heterogeneous integration at DARPA: Pathfinding and progress in assembly approaches," in *Proc. 68th IEEE ECTC*, 2018, pp. 1–40.
[3] 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. Embedd. Comput. Syst. (TECS)*, vol. 18, no. 5, pp. 1–19, 2019.
[4] J. Mack, S. Hassan, N. Kumbhare, M. C. Gonzalez, and A. Akoglu, "CEDR: A compiler-integrated, extensible DSSoC runtime," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, pp. 1–34, 2022.
[5] J. Ullman, "NP-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, 1975.
[6] 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.
[7] 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.
[8] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.

[9] Z. Tong, X. Deng, H. Chen, J. Mei, and H. Liu, "QL-HEFT: A novel machine learning scheduling scheme base on cloud computing environment," *Neural Comput. Appl.*, vol. 32, pp. 5553–5570, May 2020.

[10] A. Krishnakumar et al., "Runtime task scheduling using imitation learning for heterogeneous many-core systems," *IEEE Trans. CAD Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4064–4077, Nov. 2020.

[11] X. Wang, Z. Ning, S. Guo, and L. Wang, "Imitation learning enabled task scheduling for online vehicular edge computing," *IEEE Trans. Mobile Comput.*, vol. 21, no. 2, pp. 598–611, Feb. 2022.

[12] 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, Jun. 2020.

[13] Y. Wu et al., "Large scale incremental learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 374–382.

[14] F. M. Castro, M. J. Marín-Jiménez, N. Guil, C. Schmid, and K. Alahari, "End-to-end incremental learning," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 233–248.

[15] W. Liu, X. Wang, J. Owens, and Y. Li, "Energy-based out-of-distribution detection," in *Proc. 34th Conf. Neural Inf. Process. Syst.*, 2020, pp. 1–12.

[16] "Jetson xavier NX developer kit." Nvidia. Accessed: Mar. 10, 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/

[17] R. Cordeiro, D. Gajaria, A. Limaye, T. Adegbija, N. Karimian, and F. Tehranipoor, "ECG-based authentication using timing-aware domain-specific architecture," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3373–3384, Nov. 2020.

[18] J. Fickenscher, F. Hannig, and J. Teich, "DSL-based acceleration of automotive environment perception and mapping algorithms for embedded CPUs, GPUs, and FPGAs," in *Proc. Int. Conf. Archit. Comput. Syst.*, 2019, pp. 71–86.

[19] D. Parravicini, D. Conficconi, E. D. Sozzo, C. Pilato, and M. D. Santambrogio, "Cicero: A domain-specific architecture for efficient regular expression matching," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5, pp. 1–24, 2021.

[20] T. Wang et al., "Via: A novel vision-transformer accelerator based on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4088–4099, Nov. 2022.

[21] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, no. 184, p. 4, 2009.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] S. Baskiyar and R. Abdel-Kader, "Energy aware DAG scheduling on heterogeneous systems," *Cluster Comput.*, vol. 13, pp. 373–383, Dec. 2010.

[28] 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.

[29] 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.

[30] L. Benini, D. Bertozzi, and M. Milano, "Resource management policy handling multiple use-cases in MpSoC platforms using constraint programming," in *Proc. 24th Int. Conf. Logic Program.*, 2008, pp. 470–484.

[31] H. Yang and S. Ha, "ILP based data parallel multi-task mapping/scheduling technique for MPSoC," in *Proc. Int. SoC Design Conf.*, 2008, pp. I–134.

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

[33] Z. Hu, J. Tu, and B. Li, "Spear: Optimized dependency-aware task scheduling with deep reinforcement learning," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2019, pp. 2037–2046.

[34] T. Basaklar, A. A. Goksoy, A. Krishnakumar, S. Gumussoy, and U. Y. Ogras, "DTRL: Decision tree-based multi-objective reinforcement learning for runtime task scheduling in domain-specific system-on-chips," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 5, pp. 1–22, 2023.

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

[36] 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.

[37] A. Mallick, K. Hsieh, B. Arzani, and G. Joshi, "Matchmaker: Data drift mitigation in machine learning for large-scale systems," in *Proc. Mach. Learn. Syst.*, 2022, pp. 77–94.

[38] S. Ackerman, E. Farchi, O. Raz, M. Zalmanovici, and P. Dube, "Detection of data drift and outliers affecting machine learning model performance over time," 2020, *arXiv:2012.09258*.

[39] L. Yang et al., "CADE: Detecting and explaining concept drift samples for security applications," in *Proc. 30th USENIX Security Symp. (USENIX Security)*, 2021, pp. 2327–2344.

[40] D. M. Dos Reis, P. Flach, S. Matwin, and G. Batista, "Fast unsupervised online drift detection using incremental Kolmogorov-Smirnov test," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Min.*, 2016, pp. 1545–1554.

[41] A. Haque, L. Khan, and M. Baron, "Sand: Semi-supervised adaptive novel class detection and classification over data stream," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 1652–1658.

[42] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–21.

[43] Z. Shi, E. Jeannot, and J. J. Dongarra, "Robust task scheduling in non-deterministic heterogeneous computing systems," in *Proc. IEEE Int. Conf. Cluster Comput*, 2006, pp. 1–10.

[44] S. Chatterjee and P. Zielinski, "On the generalization mystery in deep learning," 2022, *arXiv:2203.10036*.

[45] D. Kalimeris et al., "SGD on neural networks learns functions of increasing complexity," in *Proc. 33rd Conf. Neural Inf. Process. Syst.*, 2019, pp. 1–11.

[46] S. Fort, P. K. Nowak, S. Jastrzebski, and S. Narayanan, "Stiffness: A new perspective on generalization in neural networks," 2019, *arXiv:1901.09491*.

[47] S. Chatterjee, "Coherent gradients: An approach to understanding Generalization in gradient descent-based optimization," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–13.

[48] F. Torabi, G. Warnell, and P. Stone, "Behavioral cloning from observation," 2018, *arXiv:1805.01954*.

[49] 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.

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[51] "Cross entropy loss." Accessed: Mar. 21, 2024. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[52] K. Bhatti, C. Belleudy, and M. Auguin, "Power management in real time embedded systems through online and adaptive interplay of DPM and DVFS policies," in *Proc. IEEE/IFIP Int. Conf. Embed. Ubiquitous Comput.*, 2010, pp. 184–191.

[53] 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.

[54] 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.

[55] F. M. M. u. Islam, M. Lin, L. T. Yang, and K.-K. R. Choo, "Task aware hybrid DVFS for multi-core real-time systems using machine learning," *Inf. Sci.*, vols. 433–434, pp. 315–332, Apr. 2018.

[56] 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.

[57] "DS3 simulator." Accessed: Mar. 21, 2024. [Online]. Available: https://github.com/segemena/DS3.git

[58] "ODROID-XU3." Hardkernel. 2014. Mar. 10, 2024. [Online]. Available: https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3

[59] "Zynq ZCU102 evaluation kit." Xilinx. Mar. 10, 2024. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

[60] G. Brockman et al. "OpenAI gym," 2016. [Online]. Available: https://github.com/openai/gym