IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 28 July 2022, accepted 22 August 2022, date of publication 1 September 2022, date of current version 22 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3203401

RESEARCH ARTICLE

Deep Reinforcement Learning for System-on-Chip: Myths and Realities

TEGG TAEKYONG SUNG AND BO RYU, (Member, IEEE)
EpiSys Science Inc., Poway, CA 92064, USA
Corresponding author: Bo Ryu (boryu@episyscience.com)

ABSTRACT Neural schedulers based on deep reinforcement learning (DRL) have shown considerable potential for solving real-world resource allocation problems, as they have demonstrated significant performance gain in the domain of cluster computing. In this paper, we investigate the feasibility of neural schedulers for the domain of System-on-Chip (SoC) resource allocation through extensive experiments and comparison with non-neural, heuristic schedulers. The key finding is three-fold. First, neural schedulers designed for cluster computing domain do not work well for SoC due to i) heterogeneity of SoC computing resources and ii) variable action set caused by randomness in incoming jobs. Second, our novel neural scheduler technique, Eclectic Interaction Matching (EIM), overcomes the above challenges, thus significantly improving the existing neural schedulers. Specifically, we rationalize the underlying reasons behind the performance gain by the EIM-based neural scheduler. Third, we discover that the ratio of the average processing elements (PE) switching delay and the average PE computation time significantly impacts the performance of neural SoC schedulers even with EIM. Consequently, future neural SoC scheduler design must consider this metric as well as its implementation overhead for practical utility.

INDEX TERMS Deep reinforcement learning, heuristic scheduler, neural scheduler, resource allocation, system-on-chip scheduling.

I. INTRODUCTION

Approaching the limit of Moore's Law has spurred tremendous advances in System-on-Chip (SoC) which bestows unprecedented gain in computational and energy efficiency for a wide range of applications through an integrated architecture of general-purpose and specialized processors [19]. In particular, the domain-specific SoC (DSSoC), a class of heterogeneous chip architecture, empowers exploitation of distinct characteristics of compute different resources (*i.e.*, CPU, GPU, FPGA, accelerator, etc.) for speed maximization and energy efficiency via intelligent resource allocation [25], [33], [34]. The primary goal of a DSSoC scheduling policy is to optimally assign a variety of hierarchically structured jobs, derived from many-core platforms executing streaming applications from wireless communications and radar systems, to heterogeneous resources or processing elements (PEs). Over the years, researchers have demonstrated effective performance for DSSoC with expert-crafted, heuristic rules [5], [28].

While heuristic schedulers have been dominant in a wide range of domains for resource allocation, recent effort on scheduling algorithm development started undergoing a paradigm shift toward neural approaches as they demonstrated state-of-the-art performance in complex resource management domains [17], [20]. In particular, recent successes in applying deep reinforcement learning (DRL) for scheduling heterogeneous (cloud) cluster resources [11], [32] have further motivated applying similar DRL approaches for task scheduling on DSSoC, obtaining noticeable performance gains over well-known heuristic schedulers under certain operational conditions [37], [38], [40]. Through extensive experimentation with both DRL and heuristic schedulers under extremely wide ranges of DSSoC scenarios, we present an in-depth comparative analysis between neural schedulers and their heuristic counterparts for the DSSoC domain. The key contribution of our research is that the high performance of DRL schedulers previously observed in both cloud cluster

The associate editor coordinating the review of this manuscript and approving it for publication was Amin Zehtabian.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

TABLE 1. Design features of cluster and DSSoC scheduling approaches (DLT: Deep Learning Training).

Algorithm	Application	Approach	Job	Resource	Objective
TetriSched [45]	Cluster	Estimates job run-time heuristically and plans for placement options	MapReduce jobs	Heterogeneous clusters	Minimization of errors in job execution timing
Decima [32]	Cluster	Allocates a quantity of resources to ready tasks using graph-structured information	Spark jobs	Homogeneous clusters	Minimization in avg. job completion time
Gandiva [49]	Cluster	Exploits job predictability to time-slice resources efficiently across multiple jobs	DLT jobs	Heterogeneous clusters	Improvement on cluster utilization
Tiresias [18]	Cluster	Assigns job priority using Gittins index to schedule distributed jobs	DLT jobs	Homogeneous clusters	Minimization in avg. job completion time
Themis [30]	Cluster	Uses a two-level architecture to capture placement sensitivity and ensure efficiency	DLT jobs	Heterogeneous clusters	Improvement on cluster utilization
Allox [27]	Cluster	Transforms the scheduling problem into a min-cost bipartite matching problem	DLT jobs	Heterogeneous clusters	Minimization in avg. job completion time
Gavel [35]	Cluster	Generalizes existing scheduling policies by expressing them as optimization problems	DLT jobs	Heterogeneous clusters	Minimization in avg. job completion time.
DeepRM [31]	Cluster	Includes backlog information on remaining jobs; train the agent using REINFORCE	Cluster jobs	Single cluster	Minimization in avg. slowdown
SCARL [11]	Cluster	Employs attentive embedding and schedules tasks using factorization of action	Cluster jobs	Heterogeneous clusters	Minimization in avg. slowdown
DRM [37]	SoC	Iteratively maps tasks to resources and updates the agent using REINFORCE algorithm	Single synthetic job	Heterogeneous resources	Minimization in job completion time
DeepSoCS [38]	SoC	Re-arranges task orders using graph-structured information and greedily maps tasks to resources	Synthetic and SoC jobs	Heterogeneous resources	Minimization in avg. latency
SoCRATES [40]	SoC	Iteratively maps tasks to resources and aligns post-processed returns to corresponding tasks	Synthetic and SoC jobs	Heterogeneous resources	Minimization in avg. latency

and DSSoC domains is found to be highly sensitive to the ratio of the average PE switching delay and the average PE computation time. Specifically, when this ratio is close to one, neural schedulers tend to outperform their heuristic counterparts under various operational scenarios. On the other hand, when the ratio is much less than one and subject to other operational conditions, the anticipated high performance of neural schedulers does not materialize. We attribute this to two major factors: (i) heterogeneity of SoC computing resources; (ii) variable action set caused by randomness in incoming jobs. When combined, they exacerbate the problem of *delayed reward* because the accumulated rewards are likely to disrupt the backpropagation-based optimization method. With this finding, we present a realistic avenue for future DRL-based resource scheduler design.

A. RELATED WORK

Design of high-performance SoC resource schedulers has been active for many years [5], [28]. Scheduling algorithms are mostly heuristic in nature with specific optimization goals. Examples include First Come First Served (FCFS), Earliest Task First (ETF) [6], Minimum Execution Time (MET) [9], and Hierarchical Earliest First Time (HEFT) [44]. While both MET and STF schedule tasks to PEs which take the shortest amount of execution time, HEFT schedules tasks by considering both task computation time and data transmission delays. A real-time heterogeneity-aware scheduler HetSched [4] with task- and meta-scheduling components having multiple static DAG-represented jobs as input is built for autonomous vehicle applications. A new pruning Monte-Carlo Tree Search (MCTS)-based algorithm [26] has been applied for workflow scheduling. It has improved performance in makespan over the heuristics, Improved Predict Priority Task Scheduling (IPPTS) [15], and a meta-heuristic Genetic Algorithm approach [22]. However, much of the gain depends on the specific heuristics and the nature of job configurations.

Cluster resource management for cloud computing (*e.g.*, YARN [47] or Kubernetes [8]) is another orthogonal approach in resource allocation. It is primarily designed to schedule big-data, time-persistent jobs (*i.e.*, MapReduce [12] or Deep Learning Training (DLT) jobs[1]). A list of scheduling approaches along with their design features is summarized in Table 1. Themis [30] and Tiresias [18] allocate tasks from distributed DLT jobs to homogeneous clusters using two-dimensional scheduling algorithm. Gandiva [49] schedules a set of heterogeneous DLT jobs to a fixed set of GPU clusters. It allows preemption on jobs to share overload jobs to available resources' spaces. AlloX [27] transforms a heterogeneous resources scheduling problem into a min-cost bipartite matching problem in order to provide performance optimization and fairness to users in Kubernetes. TetriSched [45] estimates job run-time heuristically for placement options. Gavel [35] transforms existing scheduling policies to heterogeneity-aware optimization problems for generalization and improves the diversity of policy objectives. Such cluster schedulers enhance run-time performance by exploiting the simulation characteristics.

[1] A neural network represents a job, and each operation, such as matrix multiplication or nonlinear function, acts as tasks.

Neural schedulers have begun to surpass hand-crafted algorithms and show a significant performance gain in the cluster scheduling problem. DeepRM, the first DRL-based cluster scheduler reported in the literature, shows significant reduction in job slowdown[2] over heuristics [31]. In comparison to DeepRM, Decima [32] proposes an end-to-end neural scheduler for more realistic cluster environment with hierarchical cluster jobs. It extracts hierarchical job information with graph neural networks (GNNs) [16] and decides how many resources to execute each task. Decima addresses varying action selection caused by the hierarchical jobs using placeholder implementation [2], but it considers homogeneous clusters. SCARL [11] aims to schedule jobs to heterogeneous resources by exploiting attentive embedding [46] in policy networks. However, SCARL is not able to schedule non-hierarchical jobs, which differs from Decima, and is not applicable to the realistic environment [32]. Spear [21] applies MCTS to plan the task scheduling with a DRL model for guidance in the expansion and rollout steps in MCTS.

Building on the success of neural schedulers for the cluster environment as described above, novel neural approaches have been proposed for the domain of SoC. Deep Resource Management (DRM) [37] is considered the first DRL-based SoC scheduler that schedules hierarchical jobs to heterogeneous resources in the scenario with a single synthetic job. DeepSoCS [38], adapted from Decima, is proposed for handling more realistic SoC scenarios where multiple numbers of both synthetic and real-world SoC jobs are continuously generated. It is a hybrid approach that rearranges the tasks using the graph-structured information extracted by GNNs and maps them to resources using a heuristic algorithm. However, the performance gain achieved by DeepSoCS depends on operational conditions, as it is inherently imitating the expert policy with an exhaustive search employed by heuristic schedulers. In order to explore the feasibility of an end-to-end neural SoC scheduler with the goal of achieving significant performance gain over heuristic schedulers, the authors proposed SoCRATES [40] with a novel technique of Eclectic Interaction Matching (EIM). EIM remedies the concurrency problem in receiving observation and reward gains by matching the time-varying interaction and simulation time steps. Consequently, SoCRATES achieves considerable enhancement in performance over prior neural schedulers [37], [38]. In this paper, we present key insights into how such performance gain is achieved by SoCRATES through extensive comparative experimentation.

B. MOTIVATION

Despite significant performance gains demonstrated by neural schedulers for cluster computing management, they generally suffer from limited extensibility. For example, prior cluster schedulers address non-hierarchical workloads [27], [49] and homogeneous resources [31], [32] that cannot fully exhibit SoC resource allocation. Although a series of research in the cluster application employs heterogeneous machines [27], [30], [45], [49], their complexity is relatively simpler than DSSoC. Schedulers in cluster applications allocate jobs to a set of CPUs or GPUs with different performances, whereas schedulers in DSSoC applications map a range of domain-specific jobs to various types of PEs, *e.g.*, CPUs, GPUs, accelerators, memory, each with different performance and supported functionalities. Cluster schedulers decide how many clusters to execute incoming tasks, whereas SoC schedulers map which SoC computing resource to an incoming task. Hence, the scheduler must be aware of unsupported action for an individual task. Hence, directly applying neural schedulers to SoC is non-trivial due to the disparities in the environment properties, such as the structures of jobs/resources and scheduling mechanisms.

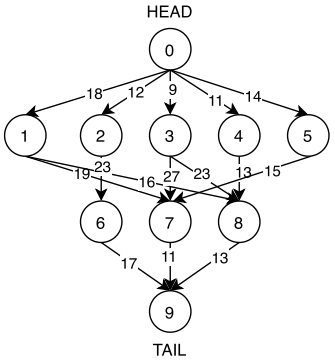
In contrast, heuristic scheduling algorithms in the domain of SoC steadily show state-of-the-art performance. We discovered that their significant performance gains come from rescheduling task assignments with exhaustive searches, such as PE availability checks or gaps between consecutive task assignments (see Section IV-B for more details). However, such rule-based algorithms generally have limited robust performance. For instance, heuristic schedulers are vulnerable to system perturbation from external forces in the setting of a single job execution [37]. Based on these robust and significant performance gains in the domain of cluster computing, we are interested in extending these neural schedulers to SoC. While neural algorithms generally adapt to dynamic system changes and have robust performance [41], subsequent works have motivated and developed in a more complicated and practical scenario with continuous job injection [38], [40]. In this paper, we investigate the challenging standpoints for designing DRL scheduling policy in the domain of SoC. With the recently introduced EIM technique overcoming such challenges, we rationalize the underlying reasons behind the performance improvement in existing neural schedulers by examining PE usages and action designs. Furthermore, we investigate which operation condition impacts the performance of neural SoC schedulers with EIM. To the end, the questions we want to consider in this paper are the following:

- What is the main difference between SoC and other domains?
- How does the neural scheduling policy effort change in SoC domain?
- Under which operational conditions do neural schedulers perform/cannot perform well?
- How does EIM technique improve the performance of neural schedulers?
- What are the strengths/weaknesses of neural scheduler?

II. BACKGROUND AND SYSTEM MODEL

A large body of research in scheduling exists for a broad range of domains. Cluster management in datacenters allocates Spark or DLT jobs to a set of CPU and GPU machines. This paper contributes to the domain of heterogeneous DSSoC

[2]This metric represents a relative value of actual job duration and ideal job duration.



Task	P0	P1	P2
0	14	16	9
1	13	19	18
2	11	13	19
3	13	8	17
4	12	13	10
5	13	16	9
6	7	15	11
7	5	11	14
8	18	12	20
9	21	7	16

FIGURE 1. An illustration of a set of synthetic job and resource profiles. The diagram on the left depicts a job DAG, where a note represents a task by its ID and the edge represents data transmission delay by its weights. The table on the right shows a set of heterogeneous PEs with different computation time for each task.

and its emulator in the form of high-fidelity Domain-Specific SoC Simulator (DS3) [5], [39]. DS3, which supports a heterogeneous SoC computing platform Odroid-XU3 [1], enables the allocation of a set of communication or radar jobs to various types of resources, such as general-purpose cores, hardware accelerators, and memory. The ARM heterogeneous big.LITTLE architecture of the cores enables performance-oriented and energy-efficiency runs (the big cores of 2.1 Ghz Cortex-A15 are performance-oriented, the LITTLE cores of 1.5 GHz Cortex-A7 are energy-efficient). DS3 integrates system-level design features for hierarchical jobs and heterogeneous resources. The job and resource profiles are given as a list specifying properties. The system parses them and generates workloads and PEs using the job and resource models.

A. SYSTEM-ON-CHIP SIMULATION

1) JOB MODEL

We define a job as a collection of interleaved tasks. Jobs in DS3 implement real-world applications of wireless communication and radar processing. The tasks represent operations, such as waveform generator, Fast Fourier Transform, vector multiplication, or decoder [5]. A *job structure* is in the form of a directed acyclic graph (DAG), illustrated in Fig. 1 [44]. A job is denoted as $G = (N, E)$, where N is a set of nodes and E is a set of edges. Each node $n_i \in N$ represents a heterogeneous task in the job, and each directed edge $e_{i,j} \in E$ connects node n_i to node n_j. We interchangeably use the term ''task'' and ''node'' unless there is confusion. The edge encodes task dependency. To be specific, if there exists edge $e_{i,j}$, task n_j can start execution only after task n_i finishes. Here, we call node n_i a parent of node n_j, and n_j a child of n_i. We define a set of parents of n_j (predecessors) by pred(n_j) and a set of child of n_i (successors) by succ(n_i). A node may have multiple parents or children, and nodes can be simultaneously executed. Each edge $e_{i,j}$ has a weight $w_{i,j}$ that represents data transmission delay between n_i and n_j. This delay is added to the task duration when the scheduler selects a different PE to task i from task j. The labels with HEAD and TAIL refer to the root parent node and the terminal leaf node, respectively. Assume a job G has v tasks, $N = \{n_1, \ldots, n_v\}$, then the job is considered complete when all tasks in N have been completed. Here, n_1 is HEAD node and n_v is TAIL node. According to the job model aforementioned, multiple jobs are generated. Each job is generated based on the following parameters [44]:

1) v: the number of tasks in the directed acyclic graph
2) α: the shape parameter of the graph. α controls the width and depth of a graph structure. We sample the average width of each level in a graph from a normal distribution with a mean of $\sqrt{v} \times \alpha$. The depth of a graph is equal to the $\frac{\sqrt{v}}{\alpha}$ (see Appendix A for details on job DAG construction). If $\alpha \gg 1.0$, a shallow but wide graph is generated; if $\alpha \ll 1.0$, a deep but narrow graph is generated.
3) ν: the average value of communication delay. The weight of $e_{i,j}$, representing communication delay, is set to max(1, $\lfloor |w| \rfloor$), where $w \sim \mathcal{N}(\nu, 0)$.
4) CCR: the communication-to-computation ratio. We calculate an average communication cost by the sum of the scheduled PE bandwidth and the weights of edges between the current task and the previous task. An average computation cost is defined in the SoC job profile. If the CCR value in a DAG is high, the job is a communication-intensive workload. Conversely, if the CCR value is low, the job is a computation-intensive workload.
5) d_{in}: an average value of in-degree of nodes
6) d_{out}: an average value of out-degree of nodes

2) RESOURCE MODEL

Resource profile defines the characteristics of PEs, and each PE is defined with a set of different, fixed supported tasks and operating performance points (OPP). OPP is a utilization set for a tuple of power consumption and task run-time frequency. OPP for PE q, for instance, can be defined by a set of voltage-frequency pairs, $\text{OPP}_q = \{(V_1^q, f_1^q), \ldots, (V_O^q, f_O^q)\}$ where O is the number of operating points. Once the frequency parameter is given, the resource model creates the corresponding PE. Since PE running with high frequency, generally, executes tasks faster but consumes more power and energy, a trade-off exists between run-time performance and energy efficiency. Moreover, each PEs has a bandwidth that contribute to the communication delay when the simulator switches over PEs during task execution.

3) OBJECTIVE

DS3 is heavily shaped by the peculiarities of the SoC domain. DS3 comes with real-world reference applications from wireless communications and radar processing domains. Each supported workload consists of various operations (*i.e.*, tasks), which require a short amount of duration. The run-time overhead of each task includes the task duration and

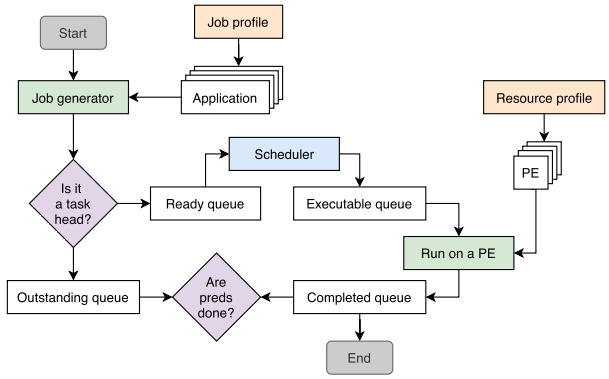


FIGURE 2. An overview of DS3 workflow. At initialization, a set of workloads and PEs are generated for given job and resource profiles. The job generator continuously generates multiple jobs using the set of workloads and distributes them to the task queues. The scheduler takes any tasks in the ready queue and maps each task to one of the PEs. If the PE is idle, it starts task execution. The task dependency graph prescribes which next task to move onto the ready queue after the completion of its predecessors.

data transmission delay. The allocation of different processors at the same time for task n_i and its parent task set $\{n_j\} = \text{pred}(n_i)$ would incur the data transmission delay. Let task n_i is mapped to PE P_i and its task computation time with operating frequency f_o^i by $\text{comp}(n_i|P_i, f_o^i)$. Then, the overall task duration is equated by

$$\text{exec}(n_i) = \mu \cdot \text{comp}(n_i|P_i, f_o^i) + \text{delay}(n_i), \tag{1}$$

where μ indicates a scaling parameter for extending the task execution time. On the right-hand side, the first term is task computation time on a PE, and the second term is data communication delay, given by:

$$\text{delay}(n_i) = \max_{n_j \in \text{pred}(n_i)} \frac{w_{i,j}}{B(P_i, P_j)}, \tag{2}$$

where $w_{i,j}$ is the weight of edges between task i and task j, and $B(P_i, P_j)$ is the PE bandwidth from P_i to P_j. The self-loop bandwidth of the same processors is assumed to be negligible, $B(P_i, P_i) = 0$. Due to the communication delay, frequent resource switching leads to an increasing loss in task completion time. The objective to optimize, the *average latency* minimization, is given by

$$L = \frac{\sum_{G \in \mathcal{G}_{\text{comp}}} \sum_{i \in |G|} \text{exec}(n_i)}{|\mathcal{G}_{\text{comp}}|}, \tag{3}$$

where $\mathcal{G}_{\text{comp}}$ is a set of completed job DAGs, and $|G|$ is the number of tasks in the job G.

Previous work [44] introduced additional evaluation metrics of run-time overhead for a single completed job: Schedule Length Ratio (SLR) and Speedup. The SLR and Speedup metrics are given by

$$\text{SLR} = \frac{\text{makespan}}{\sum_{n_i \in \text{CP}_{\text{MIN}}} \min_{p_j \in Q} \{w_{i,j}\}} \tag{4}$$

Algorithm 1 DS3 Environment

Input: job inter-arrival rate scale, clock signal clk, maximum simulation length CLK, job model M_J, resource model M_R, job capacity C, job queue Q_{job}, ready task queue Q_{ready}, job profile `job`, resource profile `resource`, W number of jobs, Q number of PEs
Output: average latency L
for each episode **do**
  clk $\leftarrow 0$
  $\{G_i\}_{i=1:W} \leftarrow M_J(\texttt{job})$
  $\{P_i\}_{i=1:Q} \leftarrow M_R(\texttt{resource})$
  repeat
    # Generate jobs
    if $|Q_{\text{job}}| < C$ **then**
      $\text{clk}_{\text{inj}} \sim \text{Exp}(\text{scale})$
      $Q_{\text{job}} \leftarrow G$ at clk_{inj}
    end if
    for each task i in Q_{ready} **do**
      # Schedule tasks in ready list to PE
    end for
    if P is idle **then**
      # PE execution
      start P execution corresponding to the scheduled tasks
    end if
    clk $\leftarrow$ clk + 1
  until clk = CLK
  Compute L using (3)
end for

$$\text{Speedup} = \frac{\min_{p_j \in Q} \left\{ \sum_{n_i \in v} w_{i,j} \right\}}{\text{makespan}}, \tag{5}$$

where the denominator of SLR_k represents the ideal lower limit time for scheduling for the job DAG. CP_{MIN} is the minimal critical path of job DAG, and Q is the number of PEs. The nominator of Speedup represents the overall task computation time when each of the v tasks in a job DAG is scheduled onto the same processor. This indicates the ability of the algorithm to schedule tasks to explore parallel performance. The lower SLR and the higher Speedup, the more optimal scheduling performance.

Since this paper seeks to evaluate performance over multiple jobs, we average out SLR and Speedup over the entire completed jobs per simulation length. Let a set of completed jobs by $\{G_i\}_{i=1}^{|\mathcal{G}_{\text{comp}}|}$, each of the completed jobs corresponds to the set of generated workloads at DS3 initialization. Since heterogeneous jobs are generated, each job likely has a different minimal critical path and parallel performance with the same processors. The average SLR and the average Speedup are given by

$$\overline{\text{SLR}} = \frac{\sum_{k=1}^{|\mathcal{G}_{\text{comp}}|} \text{SLR}_k}{|\mathcal{G}_{\text{comp}}|} \tag{6}$$

TABLE 2. A comparison between DS3 and Spark properties. Due to the differences in applicability, DS3 and Spark have different characteristics of job and resource. On DS3, a representative real-world profile, WiFi-TX, is included. Note that the shape parameter controls the diversity in the number of job types and their average graph levels, α.

	DS3		Spark
	synthetic	real-world	
Job characteristic			
Number of tasks	10	27	1137.6
Data transmission delays	16.6 ± 5.0	3.38 ± 2.4	2000
Average job DAG level	4	7	5.8 ± 2.6
Number of types	1	1	154
Average job arrival time	25		25000
Job duration	Varied		1127.3 ± 441.9
Resource characteristic			
Structure	Heterogeneous		Homogeneous
Task computation time	13.3 ± 4.1	40.0 ± 83.6	-
Number of type	4	17	1

$$\overline{\text{Speedup}} = \frac{\sum_{k=1}^{|\mathcal{G}_{\text{comp}}|} \text{Speedup}_k}{|\mathcal{G}_{\text{comp}}|}. \tag{7}$$

The DS3 workflow is given in Algorithm 1 and Fig. 2. After initialization, DS3 continuously generates indefinite hierarchical-structured workloads with respect to the job model at stochastic job inter-arrival rates. While the number of injected workloads is below the job capacity C, the job generator injects a mix of multiple instances of the workloads in a stream fashion, $\mathcal{G} = \{G_1, \ldots, G_W\}$, where $W \leq C$. The workloads are generated at every clk_{inj}, where $\text{clk}_{\text{inj}} \sim$ Exp(scale), where scale is the mean of job inter-arrival rate. A large value of job inter-arrival leads to high frequency in job injection. Then, DS3 loads a set of tasks that have no dependency onto the ready queue, otherwise onto the outstanding queue. Each ready task, which is derived dynamically based on the prior task scheduling, is ready to be scheduled by PEs using a scheduling policy. The task then moves to the executable queue, and the corresponding PE, if idle, starts executing the task. The tasks are non-preemptive, as DS3 runs in a non-preemptive setting during task execution. The job generator, distributed PEs, and simulation kernel, all of which are executed in parallel, share the same clock signal. For convenience, we describe a list of key notations and their definitions in Table 3.

B. SIMULATION ANALYSIS

Elucidating the distinction in simulation behaviors, we compare DS3 against Spark [32], one of the representative realistic simulations for cluster applications. Both simulations support the scheduling of multiple graph-structured jobs, but differ greatly in the mechanism of resource allocation in their domains. We list the job and resource characteristics after normalization in Table 2.

The scheduling policy in Spark decides on how many resource machines to allocate for the ready tasks with respect to the given job profile. DS3 scheduling policy, on the other hand, decides which PE to execute the ready tasks, and the task run-time is solely dependent on the selected PE performance. After task completion, Spark applies a static moving delay, whereas DS3 applies a dynamic data transmission delay. Cluster jobs generally consist of numerous tasks and last for a long time. Spark, for instance, supports 154 types of jobs with approximately 5.8 levels (DAG depth). Alternatively, DSSoC jobs are executed in a short range of duration. DS3 provides several types of real-world job profiles, but this paper focuses on one real-world job, WiFi-TX, and one synthetic job. These jobs have 4 and 7 levels, respectively. Endowing with heterogeneous resources, DS3 has 4 PEs on a synthetic profile and 17 PEs on a real-world profile. Each has a different run-time performance for tasks and different supported functionalities. In that sense, an individual scheduling task must check whether it can be executed in PE. Regarding CCR, the synthetic profile has a similar range of computation and communication costs. In contrast, the real-world profile is chain-structured and compute-intensive. That being said, the communication time for the synthetic job has at most 22x larger than that for the real-world job, and the task computation time for real-world resources is at most 13.4x larger than that for synthetic resources. In practice, we modify the job characteristics using shaping parameters α, μ, and ν to grant more variability. (see Section II-A1 for details on the parameter description). The difference in the

TABLE 3. A list of key notations used in this paper.

Notations	Descriptions
G_i	Job graph i
N, n_i	A set of nodes and its node
$E, e_{i,j}$	A set of edges and its edge
$w_{i,j}$	A weight value of edge $e_{i,j}$
$\text{pred}(n_i)$	A set of parents of node n_i
$\text{succ}(n_i)$	A set of children of node n_i
α	The shape parameter of the job graph
ν	An average value of communication delay
CCR	Communication-to-computation ratio
P_i	Computing PE i
μ	A scaling parameter to PE performance
L	Average latency
$\mathcal{G}$	Number of completed jobs
$\mathcal{T}$	Task queue
$\mathcal{J}$	Job queue
clk	Simulation clock signal
$R(\text{clk})$	Immediate reward at clk
G	Expected cumulative discounted returns
$\hat{G}$	Post-processed expected cumulative discounted returns
θ	Learnable neural network parameter
$\mathcal{L}^{\text{ACT}}(\theta)$	Loss on the policy network
$\mathcal{L}^{\text{CRI}}(\theta)$	Loss on the value network
$\mathcal{L}^{\text{SoC}}(\theta)$	Total loss on the neural SoC scheduler
γ	Discount factor
η	Learning rate
ξ	Entropy coefficient

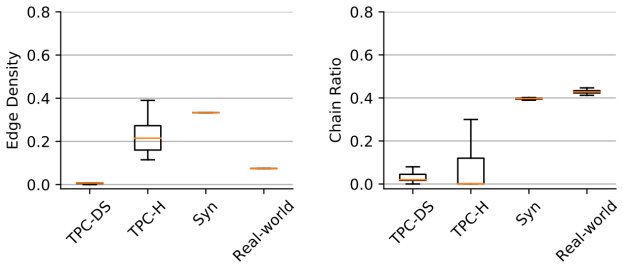


FIGURE 3. The edge density and chain ratio of cluster and SoC workloads. The results of TPC-DS and TPC-H are reproduced by referring to [43].

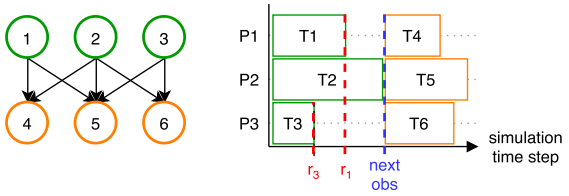


FIGURE 4. An illustration of irregular interactions. *T* refers to *task*, and *P* refers to *processor*. Although tasks 1 and 3 have been completed earlier, the next tasks 4, 5, and 6 are scheduled after task 2 has been completed. As a result, the reward gains for scheduling decisions for tasks 1 and 3 are truncated due to the task dependencies.

mechanisms in two different domains limits the scope of the applicability of each scheduling algorithm, and the extent or range of run-time is largely different.

Additional metrics are given here for hierarchical job DAGs [43]: (i) The edge density measures the sparsity of job DAGs. The density computed by $\frac{2E}{V(V-1)}$, where E denotes the number of edges, V the number of vertices (tasks), and $V(V-1)$ the possible maximum number of edges for a job DAG—the higher density results in denser job DAG with more complexity. (ii) The chain ratio measures the prevalence of chained tasks. The rate is computed by $\frac{C}{V}$, where C denotes the number of chained tasks that have an exact one child and one parent. Fig. 3 reports that the synthetic job DAG is relatively sparse, and SoC jobs have a larger number of chains than cluster jobs.

A major difficulty in designing a DS3 scheduler is that the number of available actions (scheduling decisions) varies over time due to the mixes of incoming heterogeneous jobs and their different task dependency graphs. Fig. 4 illustrates an exemplary scenario where tasks 4, 5, and 6 are a child of task 2. Although task 1 and task 3 have been completed earlier, per the dependency graph, the next observation can be received after completing task 2. Then, the immediate reward signals for tasks 1 and 3 are naturally delayed. With heterogeneous PEs- and dependency-graph-induced variations incurring abrupt dynamic run-time of tasks, it becomes an entangled affair in pairing action and its reward to compute returns properly. Indeed, this entanglement of the task dependency graph and heterogeneous resources leads to the misalignment of the order and timing of observation and reward gains. In that sense, the agent has a mismatched reward timestamp with the actual simulation running clock signal. As a result, the interactions become inconsistent, and rewards (returns) will be incorrectly assessed and backpropagated.

Based on the above analysis, DS3 exhibits dynamic, realistic operational behaviors but differs significantly from other domains of resource allocations. Due to task dependencies from mixes in various jobs, the scheduler must address a variable action set (i), shown in the list below. The distributed PE executes each scheduled task accordingly (ii). By combining (i) and (ii), the agent naturally has delayed rewards likely to disrupt DRL optimization. That is the last difficulty, (iii).

1) **Variable action sets**: Mixes in various jobs with different task dependency graphs cause variable action sets. Since the job queue holds multiple heterogeneous jobs, the agent must recognize multiple job graphs and respond to the fact that action sets are irregular. At every scheduling interaction, the agent receives tasks free of dependencies for the given state.
2) **Heterogeneous resources**: As heterogeneity in both jobs and SoC computing resources, the DS3 scheduler must consider different task execution times and data transmission delays. The SoC scheduler computes which task to be executed on which PE. Based on the task-PE mappings from a scheduling policy, the average job duration becomes highly unpredictable.
3) **Delayed rewards**: A combination of the varying actions caused by randomness in incoming jobs and heterogeneous resources exacerbates the problem of delayed rewards. The accumulated rewards tend to disrupt DRL optimization. With the previous action commitments and a varying number of observations, the returns must match the interaction steps and the actual simulation clock signal.

C. BENCHMARK SCHEDULER

1) RULE-BASED SCHEDULER

The task duration depends on task computation time on a PE and communication delay computed by the PE bandwidth and data transmission delay in the job DAGs, as described in (1). Shortest Time First (STF) and Minimum Execution Time (MET) [7] iteratively schedule ready tasks to the PE that has minimal execution time. After the schedules, MET checks whether the PE is busy or idle. If the scheduled PE is busy, then MET revises the task assignment to alternate PE. Heterogeneous Earliest Finish Time (HEFT) [44] is effective at hierarchical job scheduling. HEFT first sort ready tasks based on the upward rank values, which are importance weights, and greedily map tasks to heterogeneous PEs. An upward rank of a ready task n_i can be recursively calculated by

$$\text{rank}_u(n_i) = \overline{w_i} + \max_{n_j \in \text{succ}(n_i)} (\overline{c_{i,j}} + \text{rank}_u(n_j)), \tag{8}$$

where $\text{succ}(n_i)$ is a set of successors of task n_i, $\overline{c_{i,j}}$ is the average communication cost of edge (i, j), and $\overline{w_i}$ is the average computation cost of task n_i. Essentially, the upward

rank is the length of the critical path from task n_i to the exit task. While the original research seeks the critical path [44], a job DAG in DS3 is deemed complete when all of its tasks are finished. Therefore, the performance of HEFT relies heavily on the heuristic task-PE mapping, which iteratively computes the earliest execution finish time (EFT) of a ready task. EFT of task n_i and processor p_k is equated by

$$\mathrm{EFT}(n_i, p_k) = \max\{\mathrm{avail}[k], \max_{n_j \in \mathrm{pred}(n_i)} (\mathrm{AFT}(n_j) + c_{i,j})\}, \quad (9)$$

where avail$[k]$ is the earliest time at which the processor p_k is ready for task execution, pred(n_i) is the set of predecessor tasks of n_i, and AFT(n_j) is the actual finish time of the task n_j. $c_{i,j} = w_{i,j}/B(p_i, p_j)$, where $w_{i,j}$ is the weight of edge (i, j) and B is bandwidth between given processors, is the data transmission delay as referred to (2). Essentially, EFT algorithm calculates actual delay-aware computation time and exhaustively schedules the task to the PE with minimal cost. HEFT particularly applies an insertion-based policy that seeks whether the scheduled task can be executed prior to the previous task assignment. If HEFT finds residual gaps due to transmission delays associated with previous scheduling decisions, it reschedules the tasks. Recent improvement via execution-focused heuristic in dynamic run-time scenarios resulted in a run-time variant of HEFT, $\mathrm{HEFT}_{\mathrm{RT}}$ [28].

2) NEURAL SCHEDULER

DeepSoCS [38] first introduced in the DSSoC with the realistic setting. It sorts tasks using the topological knowledge extracted by graph neural networks and maps each task to PEs using exhaustive search, EFT algorithm [44], accordingly. DeepSoCS shows a promising result in the SoC application by exploiting the insertion policy and imitating the expert policy, HEFT. However, mapping the tasks to PEs crucially impacts the performance rather than sorting the tasks in DS3 due to counting job completion after all tasks are finished.

SCARL [11] is designed for scheduling a single-level job input to heterogeneous machines in a pre-defined number of injecting jobs. SCARL employs attentive embedding [46] to share representation from the job and resource embedding and allocate each job to the available machine. The scheduler conducted experiments in the extended version of the simple cluster simulation [31].

III. PROPOSED METHOD

The critical challenges for designing a DRL scheduler in DS3 are that the scheduler requires to i) adaptively allocate a varying number of tasks to heterogeneous PEs by considering system dynamics and data transmission delays, and ii) correctly align task returns to scheduling actions according to respective time-varying agent experiences. An overall systematic workflow of DS3 with scheduling policies is depicted in Fig. 5. After the tasks enter the ready queue, the scheduler receives an observation and maps each task to corresponding PEs. For the following subsections, we provide the state, action, and reward statements tailored to DS3. As the set of actions varies in order and time, we provide cases on how to design actions. Also, we delineate a straightforward and effective EIM technique and how this technique addresses the alignment of return.

A. AGENT DESCRIPTION

Applying RL to sequential decision-making problems is natural, as it collects experiences via interactions with the environment. In general, conventional RL is formalized by Markov Decision Processes (MDP), which is consisted of a 5-tuple $\langle \mathcal{S}, \mathcal{A}, R, P, \gamma \rangle$ [41]. Here, $\mathcal{S} \in \mathbb{R}^d$ is the state space, $\mathcal{A} \in \mathbb{R}^n$ is the action space, and $R \in \mathbb{R}$ is the reward signal that is generally defined over states or state-action pairs. $P : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a matrix representing transition probabilities to next states given a state and an action. $\gamma \in [0, 1]$ is the discount factor determining how much to care about rewards in maximizing immediate reward myopically or weighing more on future rewards. RL aims to discover an optimal policy π that maximizes the expected cumulative (discounted) rewards or (discounted) returns. At every interaction, the RL agent samples a (discrete) action from its policy, which is the probability distribution of actions given a state, $a_t \sim \pi(s_t)$. The agent then computes the return with $\mathbb{E}[\sum_{t=0}^{T} \gamma^{t-1} R(s_t, a_t)]$, where t is the interaction time step. In this paper, we assume a finite state, finite action, and finite-horizon problem.

1) STATE

The state representation is designed to capture information of simulation dynamics. Considering the SoC domain-specific knowledge, we select the attributes of the overlapping tasks/jobs and resource information. The observation features at every interaction are

$$\mathrm{Concat}((P_n^G, \mathrm{Stat}_n^G, \mathrm{TWT}_n^G, |\mathrm{pred}_n^G|)_{n=0,G=0}^{v,W}, (\mathrm{Dep}^G, \mathrm{JWT}^G)_{G=0}^{W}, N_{\mathrm{child}}), \quad (10)$$

where n is a task in every job G, v is the number of tasks in job G, and W is the number of job DAGs in job queue. Each of the observation features is described as follows.

- P_n^G, the assigned PE ID.
- Stat_n^G, one-hot embedded task status. Status is classified by one of the labels from ready, running, or outstanding.
- TWT_n^G, the relative task waiting time from the ready status to the current time.
- $|\mathrm{pred}_n^G|$, the number of remaining predecessors.
- Dep^G, the number of hops (levels) for the remaining tasks as referred to task dependency graph.
- JWT^G, the relative job waiting time from injected to the system to the execution time.
- N_{child}, the number of all awaiting child tasks in the outstanding and ready statuses.

Time in observation features refers to the actual clock signal in an SoC simulation. Based on the choices of neural architecture designs, state representation includes graph embeddings that capture topological information using graph neural networks [11], [32], [38].

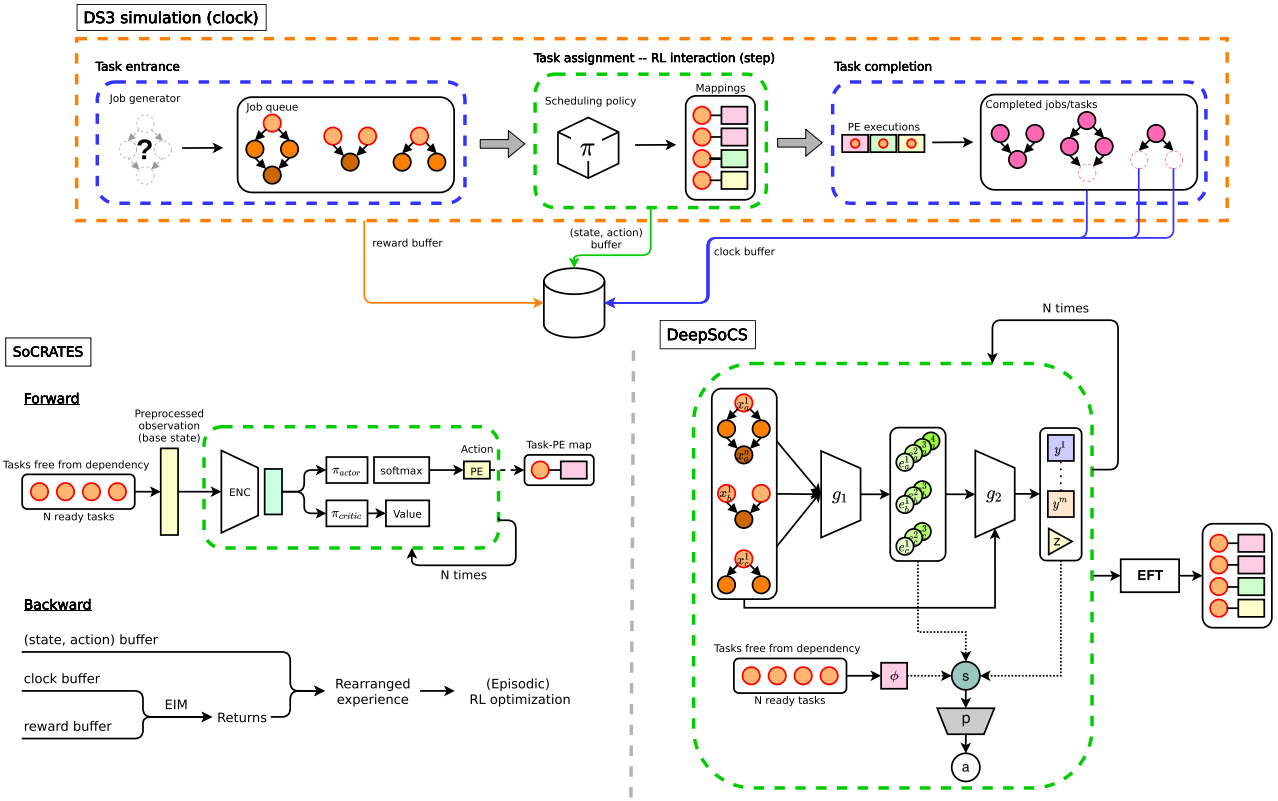


FIGURE 5. The architecture of neural schedulers applied to DS3 simulator. Schedulers receive *N* tasks in the ready queue and map each task to SoC computing resources. Due to the varying number of tasks, scheduling policies feed each task iteratively. SoCRATES applies Eclectic Interaction Matching to post-process the return (bottom-left). DeepSoCS returns sorted tasks and uses the EFT algorithm to map them to resources (bottom-right).

2) ACTION

At every task assignment, shown in the top-middle stage from Fig. 2, the agent performs a scheduling decision on an individual task that is free from dependency. Since the number of ready tasks varies by the previous scheduling decisions following their dependencies, the feasible action set varies. Let the ready tasks by $a_t \in \mathcal{T}_{\text{ready}}$, where $\mathcal{T}_{\text{ready}}$ is a set of ready tasks. For every task $\{a_i\}_{i=1}^{|\mathcal{T}_{\text{ready}}|}$, an action i is sampled from the policy distribution with parameter θ, $a_i \sim \pi_\theta(a|s)$, which can be represented by multinomial distribution, $\pi_\theta(a|s) \overset{d}{=} \text{Multinomial}(p, m)$. Here, $p \in \mathbb{R}^{1\times Q}$ is the probabilities of each PEs, Q is the number of PEs, and $m \in \mathbb{R}^{1\times Q}$ is a masking vector to filter out PEs not supporting the task.

One approach is to consider a set of actions as a group action. The group action at RL interaction time step t can be represented by $a_{i,t} \sim \pi_\theta(s_t)$, where the set of actions are sampled from the same probabilities with respect to policy distribution. In practice, we define the size of the action vector to a large enough number and apply zero-padding whenever the number of ready tasks is less than that [48]. An alternative approach is to treat each ready task as an individual action. In lieu of the group action, the agent pulls out each for a PE selection with respect to different policy distribution iteratively, $a_{i,t} \sim \pi_{\theta,(i)}(s_t)$. In this case, each action is sampled from different probabilities with respect to policy distribution.

3) REWARD

DS3 schedulers aim to minimize average latency over simulation length. As described in (3), the number of completion jobs is largely dependent on the latency. While the negative job duration reward is an adequate reward metric in a cluster environment [32], this is not effective for latency. Minimizing the elapsed time of the completed jobs is a local optimization, while increasing the number of completed jobs is a global optimization to entail latency minimization in overall. Moreover, maximizing the number of tasks is not adequate optimization because leaving one task out of a job did not contribute to the job completion. We state the reward function as follows.

$$R(\text{clk}) = C_1 \cdot |\hat{\mathcal{G}}_{\text{comp}}| + C_2, \tag{11}$$

where $|\hat{\mathcal{G}}_{\text{comp}}|$ is the number of newly completed jobs at clk, C_1 and C_2 are the weights of job completion bonus and penalty for clock signal, respectively. The second term on the right-hand side (C_2) represents a penalty and acts as continuous reward feedback on every running clock signal.

In general, C_2 is a negative value that encourages the agent to complete jobs quickly. C_1 is a positive value that is likely to be higher than previously accumulated penalties. It is non-trivial to verify the relationship between the number of completed jobs and accumulated penalties due to the combinatorics in the task-resource mapping and the combined effect on workload characteristics and stochastic job injection rate. The values of $+50$ for C_1 and -0.5 for C_2 were chosen empirically for our study, which has shown effective performance gain over various workload scenarios. Note that this reward function is computed per clock signal to enable the EIM technique, which is discussed in the following section. For the standard RL approach, the reward is computed per interaction step t instead of clock signal clk.

B. ECLECTIC INTERACTION MATCHING

Conventional RL environments formalized in standard MDP assumption return the next observation and action consequences right after the previous action has been completed. However, as introduced in the example case in Fig. 4, the action in DS3 is performed with the ready tasks, and it is highly not regularly performed due to the variability in task dependency from a mix of incoming jobs. As a result, the next observation is not immediately generated after executing the previous scheduled task. Moreover, treating a reward and the next observation at the same time leads to incorrect reward propagation, because the scheduler assigns multiple tasks at the same time, and each of the scheduled tasks readily be completed at different time due to the different performance of heterogeneous PEs. In that sense, the task dependencies and different task duration inherently cause delayed rewards, and this phenomenon leads to incorrect reward propagation in the optimization updates. Therefore, the scheduling agent must handle a varying number of action sets and the mismatches between the interaction and the action effects during the action decision stage, for which we address right below.

A standard RL experience comes down to a sequence of $\langle s, a, r, s' \rangle$. We first decouple the receiving reward and next state to have i) a sequence of $\{s_i, a_i\}_{i=1}^{T}$, where T is the last interaction step, and ii) a list of rewards collected upon the simulation clock signal $\{r_{\text{clk}}\}_{\text{clk}=1}^{\text{CLK}} = \{R(\text{clk})\}_{\text{clk}=1}^{\text{CLK}}$, where $R(\text{clk})$ is a reward function described in (11). As discussed in Section II-B, an amount of interactions T and entire clock signal CLK are not generally matched due to the different task dependencies in a mix of hierarchical jobs and performances in heterogeneous PEs. We compute the immediate reward per clock signal independent of the interaction step. Additionally, we append the 'start' flag to the state-action tuple and stored in the buffer. While traversing the simulation, at the completion of any scheduled task, we store the 'complete' flag and completed clock signal ω in the buffer. Hence, the experiences in the buffer is described as $\{s_t, \{a_{(n,t)}\}_{n=1}^{n'}, \{\omega_{(n,t)}\}_{n=1}^{n'}\}_{t=1}^{T}$, where n' is the number of ready tasks at interaction step t.

Fig. 6 showcases an exemplary experience of scheduling three ready tasks and return computations using two different strategies. On the upper-left diagram, an agent receives an observation and sequentially selects an action. The state, action, and task starting/completing clock signal are marked green. Next, we compute returns based on the accumulated rewards. In the standard approach performing the Monte-Carlo return with the accumulated rewards [42], partial reward sequences that overlap ongoing tasks and subsequent observation are not counted; the missing sequences incur incorrect return matching and instability in training.

The EIM technique instead aligns Monte-Carlo returns with the committed actions spanning individual task duration, referred to the 'start' and 'end' task signals. The return for each action reflects the length of task duration, and each action correctly matches outcomes without any discarded information. Moreover, task flags and actual clock signals allow the agent to sequentially select actions within a set of varying actions. EIM technique thus enables the agent to receive a correct form of state-action-return triplets, regardless of varying action sets. EIM is a straightforward post-process that is proven effective in training an agent when the agent interaction and simulation clock signal is inconsistent. The bottom diagram of Fig. 6 depicts the task and action with the return computation. The x-axis denotes the simulation clock signal, and the y-axis is the RL interaction time step. Partitions of second and third task duration in the standard approach are discarded for return assignment. EIM, by contrast, properly pairs the state-action tuple with returns by aligning returns to task assignments.

In training, we use the Actor-Critic algorithm [24]. We use shared neural networks on both actor and critic and update parameters with REINFORCE [42]. While the actor network selects actions with respect to the policy distribution, π_θ,

Algorithm 2 SoCRATES Scheduler

Input: clock signal clk, job queue $\mathcal{J}$, ready task queue $\mathcal{T}_{\text{ready}}$
for each episode **do**
  state-action buffer $\mathcal{B}_{\text{SA}} \leftarrow \emptyset$
  clock buffer $\mathcal{B}_{\text{clk}} \leftarrow \emptyset$
  reward buffer $\mathcal{B}_R \leftarrow \emptyset$
  for each task i in $\mathcal{T}_{\text{ready}}$ **do**
    Construct state s_t
    $a_{i,t} \sim \pi_{\theta,(i)}(s_t)$
    Assign $a_{i,t}$ to PE for task i
    $\mathcal{B}_{\text{SA}} \leftarrow (s_t, a_{i,t})$
  end for
  if task i complete **then**
    $\mathcal{B}_{\text{clk}} \leftarrow (i, \omega)$
  end if
  $\mathcal{B}_R \leftarrow r_{\text{clk}}$
end for
# Update the agent model
$\theta \leftarrow \theta + \eta\gamma^t \nabla_\theta \mathcal{L}_t^{\text{SoC}}(\theta)$

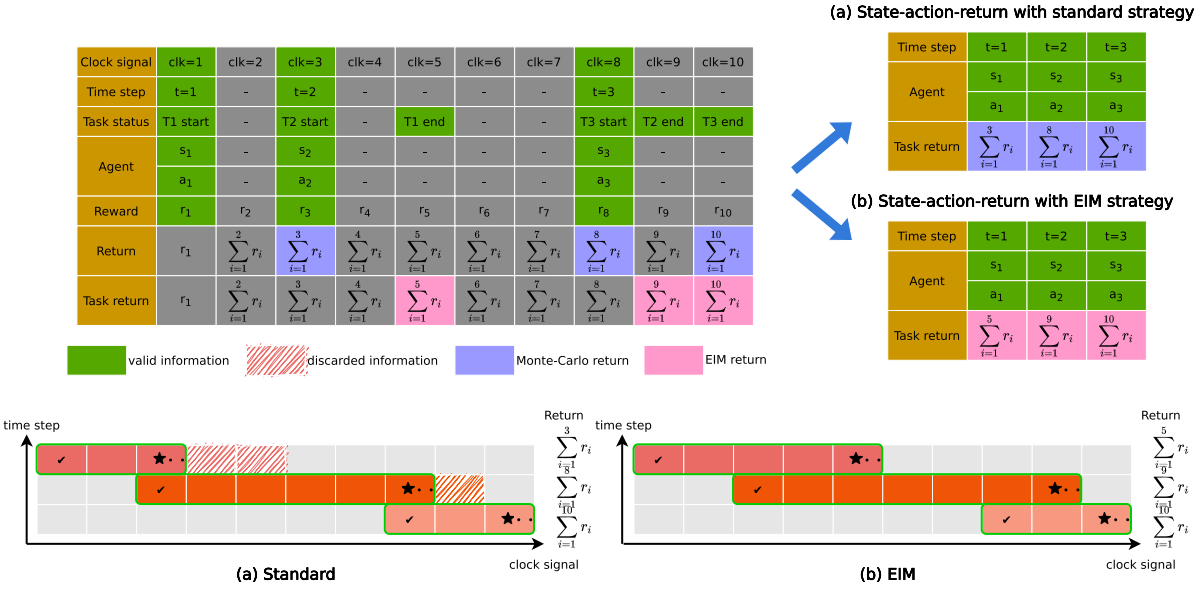


FIGURE 6. An experience with different strategies. Top figure depicts the experiences and rearranged state-action-return sequential triplets after processing different strategies: standard and EIM. EIM preserves the integrity of task execution for return calculation via accounting for returns spanning task duration. Bottom figure describes three tasks coming with concurrency and inconsistent interaction due to task dependency and heterogeneous resources. Two orthogonal axes show interaction time step and actual simulation clock signal. For standard strategy, delayed consequences are discarded, as depicted by shaded regions.

the critic network estimates the value using the value function, $\hat{V}_\theta^\pi$. At the end of the episode, EIM post-processes the expected returns as described in (12):

$$\hat{G}(s_t, \omega) = \sum_{\text{clk}=0}^{\omega} \gamma^{\text{clk}} r_{\text{clk}+1}. \tag{12}$$

The actor loss is equated by

$$\mathcal{L}_t^{\text{ACT}}(\theta) = -\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\Big[\hat{G}(s_t, \omega) - \hat{V}_\theta^\pi(s_t)\Big], \tag{13}$$

and the critic loss is computed by the standard mean squared loss,

$$\mathcal{L}_t^{\text{CRI}}(\theta) = \frac{1}{2}\Big(\hat{G}(s_t, \omega) - \hat{V}_\theta^\pi(s_t)\Big)^2. \tag{14}$$

The overall loss is given as:

$$\mathcal{L}_t^{\text{SoC}}(\theta) = \mathcal{L}_t^{\text{ACT}}(\theta) + \mathcal{L}_t^{\text{CRI}}(\theta) + \xi\mathcal{H}(s_t), \tag{15}$$

where the last term on the right-hand side is the entropy regularization, $\mathcal{H}(s_t) = \mathbb{E}_{\pi_\theta}[\log \pi_\theta(s_t)]$, with its coefficient ξ introduced for exploration. Pseudocode for the proposed algorithm is given in Algorithm 2.

IV. EVALUATION

This section demonstrates the feasibility of neural schedulers in a high-fidelity SoC simulation, DS3. We present evaluations in three ways: (a) we revisit rule-based schedulers and observe their benefits on performance, (b) we verify the efficacy of EIM technique on neural schedulers by investigating PE usage with various reward functions and different action designs, and (c) we empirically validate that neural schedulers can have competitive and generalized performance on run-time overhead in a series of experiments where job DAG topology and PE performance are varied. Specifically, we examine in which operational conditions existing neural schedulers with EIM have significant performance gains.

TABLE 4. A table of hyperparameters used for training neural schedulers.

Hyperparameter	Value
Optimizer	Adam
γ (discount factor)	0.98
η (learning rate)	0.0003
ξ (entropy coefficient)	0.01
α (job structure)	0.8
μ (scale to PE performance)	0.5
ν (avg. comm. delay)	0.0
Number of workloads	200
Simulation length	10,000
C (job capacity)	3
Gradient clip	1
Scale	25

A. EXPERIMENTAL SETUP

Table 4 describes a list of training parameters. We use Adam optimizer [23] and clip the gradients to avoid gradient explosion. To engender more interactions and a more dynamic environment, we randomly inject jobs with a set of 200 workloads. As we empirically discovered that the

job DAG topologies of synthetic or real-world profiles are structured with $\alpha = 0.8$, we synthesize job structures with $\alpha = 0.8$ based on the given job profile. The job inter-arrival rate (scale) is set to 25; the system stochastically injects a job at every 25 clock signals on average. At every episode, the simulation executes until 10,000 clock signals. To reduce the training and evaluation time, we conduct all experiments with the initial condition of quasi steady state, that is, each experiment begins with all jobs already stacked in the job queue [38]. All evaluations have been conducted by 20 trials with different random seeds.

B. REVISITING RULE-BASED SCHEDULERS

Rule-based algorithms have continue to demonstrate state-of-the-art performance in SoC run-time scheduling [5], [28]. In order to establish a baseline for comparative study, first we extensively investigate the run-time performance of existing heuristic schedulers. As described in Section II-C1, the main discrepancy between STF and MET is that MET reschedules the scheduling assignment by checking whether the selected PE is busy or idle. Likewise, HEFT_{RT} iteratively computes the actual run-time of given tasks using computation time and data transmission delay. It then applies an insertion policy, which exhaustively searches for a possible empty slot between each task assignment.

The top plot in Fig. 8 shows an overall run-time performance of different heuristic schedulers using synthetic (Syn) and real-world (RW) profiles. The x-axis indicates CCR, and the y-axis indicates the average latency. The jobs are communicative-intensive, if CCR $\gg$ 1.0, and are computation-intensive, if CCR $\ll$ 1.0. For synthetic profiles structured with a similar range of CCR, STF and MET have comparable performance and surpass HEFT_{RT}. On the other hand, MET and HEFT_{RT} significantly improve performance for real-world profiles, where jobs are computation-intensive, by checking the availabilities in PEs and exhaustively searching the empty slot between task assignments. Particularly, when increasing CCR on real-world profiles, MET and STF show similar performance and surpass HEFT_{RT}. We observe that the increasing gap between computation time and communication delay leads to large variances in the distribution of task run-time. The high variances in profile statistics result in more chances to improve the performance by rescheduling task assignments.

The bottom plot in Fig. 8 shows an experimental result for real-world profile using two types of HEFT_{RT}, with and without insertion policy. The insertion policy effectively seeks better placements due to the divergent distribution of task computation time. Hence, the rescheduling task assignment in the heuristic schedulers instrumentally improves run-time performance. In that sense, rescheduling task assignments can largely improve performance, and HEFT_{RT} can show almost optimal performance within a myopic scope by its exhaustive search when the variations in task run-time are large.

C. PERFORMANCE COMPARISON

This section describes our extensive evaluation of the performance of existing schedulers specifically designed for heterogeneous resources in the SoC domain. We compare two types of representative scheduling algorithms: i) SoCRATES [40], DeepSoCS [38], and SCARL [11] for neural, and ii) STF, MET [7], and HEFT_{RT} [28], [44] for heuristics. Since SCARL does not support hierarchical workloads, we modified SCARL as follows: (1) *State*: We take the same job representation with the SoCRATES. We select PE performance, types of PE, capacity, available time to execute tasks, task remaining execution time, idle rate, and normalized values of PE run-time and expected total task time for features of PE representation. (2) *Action*: Original SCARL selects both workload and resource. Since SCARL does not support task selections for hierarchical workloads, the action maps the selected resource to the task in sequence. (3) *Reward*: We use the same reward function of job completion, described as (11). At the update stage, we compute the returns with the collected rewards after post-processing with EIM.

Throughout the evaluations, we primarily concentrate on *average latency*, which indicates the average run-time performance. We observe how the schedulers behave in a wide range of experiments by varying the types and structures of the jobs, transmission delay, and performance in heterogeneous PEs. Fig. 7 reports the run-time performance using a synthetic workload. The right and left plots depict the experimental results after varying job structures and PE performance. For the former case, we control the job structure parameter α while holding the parameter of PE performance μ, and for the latter case, vice versa. Large α generates shallow but wide job graphs, while small α generates deep but narrow job graphs. All evaluations are conducted with the highest job inter-arrival rate (the smallest scale value), leading to a high frequency of job injection. From the holistic viewpoint, the trends in SLR and Speedup follow the curve of the average latency. Among all other schedulers, we can observe that SoCRATES has surpassed under a wide range of experiment settings. Since the neural schedulers have been evaluated using a single trained model, SoCRATES has generalized and competitive performance in various scenarios in job structures and PE performances. As described in Section IV-B, CCR for the synthetic workload closely reaches 1.0, meaning that the task computation time and data transmission delay lie in a similar range. As a result, the task ordering in heuristics did not impact much, and their performances fell behind the SoCRATES. When the number of tasks was varied, SCARL's attentive embedding of tasks and resources was unable to take advantages of attentive representation and even further deteriorates the overall run-time performance. As a result, SCARL shows comparable performance to random policy.

Synthetic and real-world profiles differ in the number of tasks and resources, job DAG topology, and supported functionalities on individual resources. Table 2 indicates that the real-world profile has a much higher task computation time

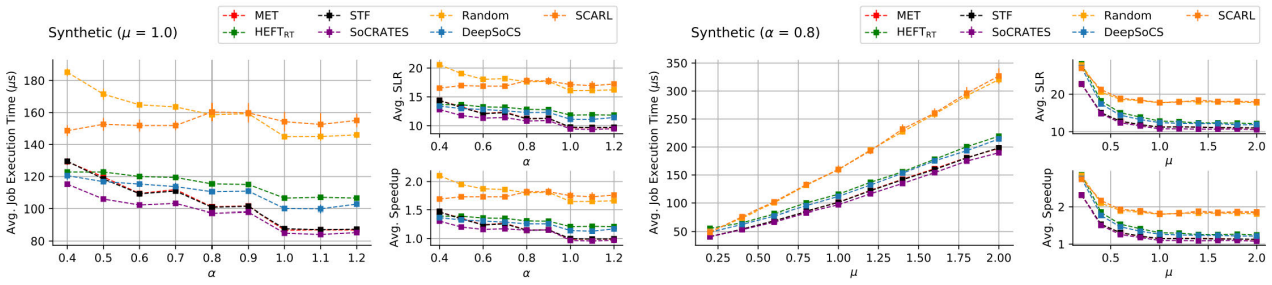


FIGURE 7. Performances evaluation on neural and non-neural schedulers with various job structures and PE performances using synthetic profiles. The left and right figures show run-time performance on varying topologies in job DAGs and PE performances, respectively.

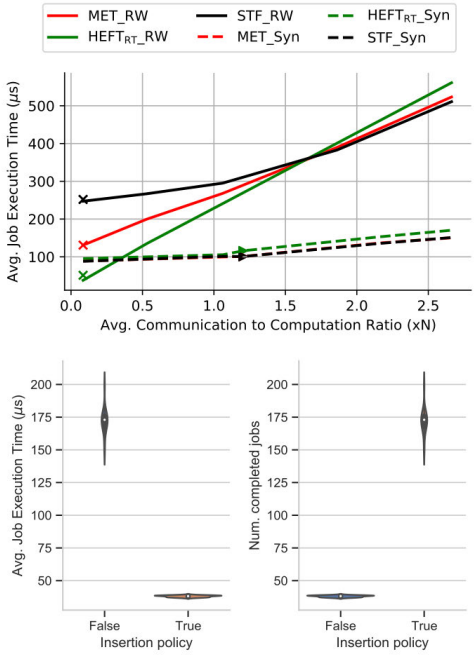


FIGURE 8. An experimental analysis of heuristic schedulers. The top figure compares run-time performance with different data transmission delays. The cross and triangle marks depict fixed job profiles from synthetic and real-world workloads. We report the results with a solid line for real-world (RW) and a dotted line for synthetic (Syn) workloads for the increase in data transmission delay compared to task computation time. Both lines are plotted by varying the job structure with $\alpha = 0.8$. The bottom figure shows the effectiveness of insertion policy in HEFT_{RT} scheduler using violin chart. An insertion policy significantly improves performance for average latency (bottom-left) and increases the number of completed jobs (bottom-right).

cost than the data transmission delay. Hence, the actual task run-time is more varied, and rescheduling task assignments from heuristic schedulers can largely improve the run-time performance. As a result, SCARL significantly improves performance, and HEFT_{RT} shows the most optimal run-time behavior in the real-world profile, as shown in Fig. 9. We observe that SoCRATES surpassed other schedulers in the synthetic profile, but it was limited in the real-world profile. Although EIM remedies fundamental difficulties in DS3 and improves SoCRATES performance, it cannot reduce the performance gap for the optimal task scheduling in a myopic range using an exhaustive search. We hypothesize that the characteristics of the real-world profile, such as various availabilities of task executions in resources, invokes cohesive challenges to designing a DRL scheduler. Additionally, a large number of tasks leads to increased complexity in task dependency composition and large variance because completing all tasks counts as job completion. Hence, the end-to-end neural approach could not surpass HEFT_{RT} when the task duration has high variance and computing resources cannot compute all tasks. Although SoCRATES shows limited performance in the real-world profile, DeepSoCS shows comparable performance to HEFT_{RT} by imitating experiences from the expert algorithm.

In conclusion, Fig. 10 demonstrates the overall evaluation of neural and non-neural schedulers using synthetic and real-world profiles with various job DAG topology by controlling α. The left figure shows that SoCRATES has largely improved behavior when the computation time and communication delay lie in a similar range. The right figure shows that DeepSoCS and HEFT_{RT} shows the most optimistic performance when the composition of task duration has a large variance. Thus, if we adaptively choose a neural scheduler between EIM-based policy and imitated expert policy depending on different scenarios, the neural SoC schedulers can obtain an improved performance over other neural and non-neural schedulers.

D. ANATOMY OF SOCRATES

SoCRATES is the fully differential decision-making algorithm [40]. The crucial component in SoCRATES is EIM technique that alleviates both delayed rewards and variable action selection, caused by hierarchical job graphs, mixes of different jobs, and heterogeneous computing resources. Although the recently introduced EIM technique overcoming such challenges, it lacks the validation of the efficacy of EIM. In this section, we rationalize the underlying reasons behind the performance improvement of EIM by examining PE usages and action designs.

1) ANALYSIS OF ECLECTIC INTERACTION MATCHING

First, we examine how EIM affects the scheduling policy decisions with the PE selection behavior. The top plot in

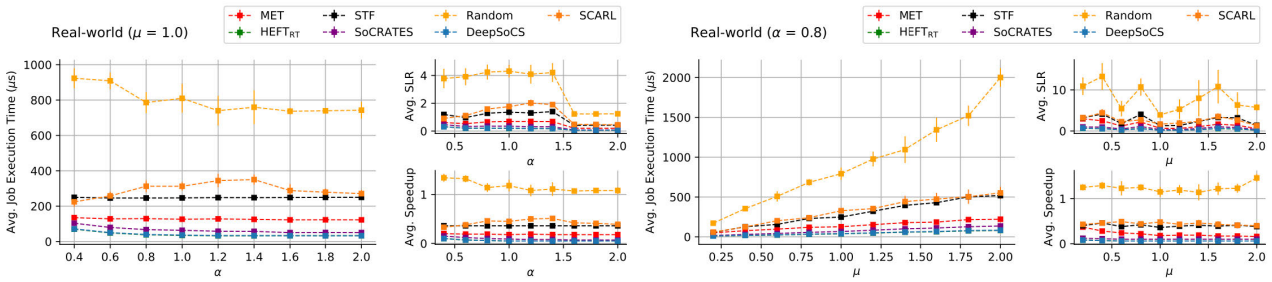


FIGURE 9. Performances evaluation on neural and non-neural schedulers with various job structures and PE performances using real-world profiles. The left and right figures show run-time performance on varying topologies in job DAGs and PE performances, respectively.

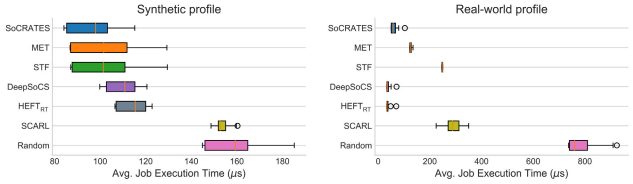


FIGURE 10. An overall result of *average latency* with various job DAG topology by controlling α. The left plot shows the synthetic profile, and the right shows the real-world profile.

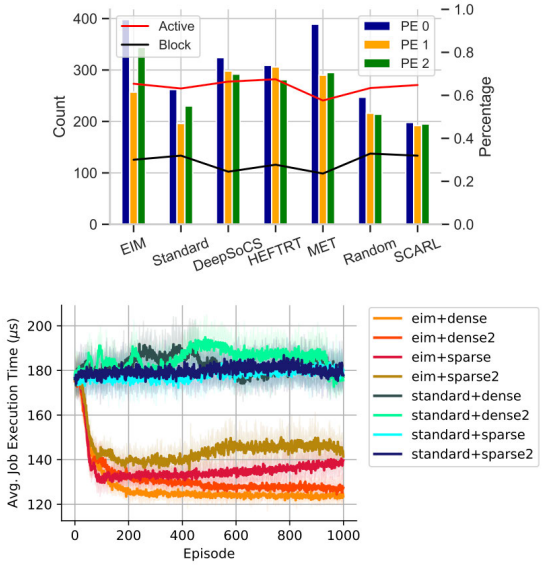


FIGURE 11. Experiments for the analysis of EIM technique. Top plot shows PE selection and active/block time for PEs in different scheduling algorithms. Bottom plot compares latency performance using EIM and standard strategies with various sparse and dense reward functions. All experiments are conducted with synthetic workloads with a fixed job topology.

Fig. 11 shows the counts on each PE execution and the total amount of time for PE in active and blocking time. Active time represents the amount of time in PE execution, and blocking time represents the duration when a PE is busy while the assigned tasks are ready. The simulation clock signal measures each time. Intuitively, optimized performance in PE usage can be achieved when active time increases but blocking time decreases. SoCRATES with EIM technique, in effect, utilizes a greater number of PEs and achieves higher resource active time than that with other policies. Its high blocking time derives from the fact that the policy weights encourage achieving long-term returns while blocking the cost of immediate returns. The same policy without EIM technique also has similar values of active and blocking times. However, its low number of PE counts leads to poor PE utilization and latency behavior. MET also uses a large quantity of PEs, but its low active time in PEs invokes additional bottlenecks in PE usage. Random and SCARL policies show high value in active time. However, their absolute number of PE counts is much lower. As a result, they have poor performance compared to other schedulers.

Next, as shown in the bottom plot, we train SoCRATES using various types of reward functions with and without the EIM technique for generalization. We train the policy using synthetic workload, and each two types of dense and sparse reward functions are used (see Appendix B for more details). The solid line represents the average values, and the shaded region bounds the maximum and minimum values among 8 runs in random seeds. The standard strategy seemingly cannot train the model by observing its steady and straight performance curve. On the other hand, the EIM strategy enables to show learning curve. The EIM iteratively matches each return in actions with respective task duration at the cost of storing extra flags on task start and completion. This additional post-processing is very cheap in operations and achieves substantially better latency performance in any kind of reward function than the standard strategy. From the reward function perspective, the sparse reward apparently exacerbates unstable latency performance due to its limited feedback for an RL agent. Hence, it is commonly modified to dense forms using the shaping technique [36].

2) ACTION DESIGN

To design a DRL agent in an environment with varying actions, one can set an action space to the number of maximal actions and mask out every varying action [48]. In the case of group actions, we distribute the returns, computed by the longest task duration, to the set of actions. It turns out,

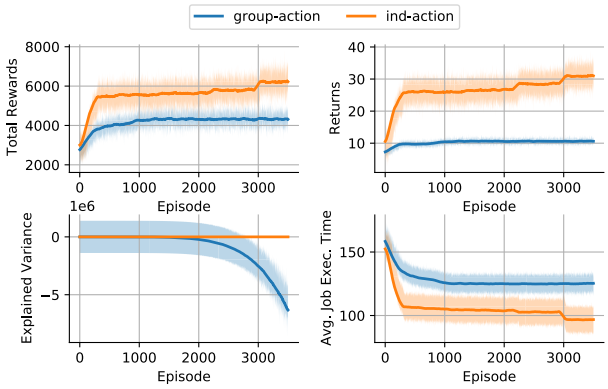


FIGURE 12. A set of performance metrics for selecting a group action and independent actions in the training phase. Top-left: total rewards, top-right: returns, bottom-left: explained variance [13], and bottom-right: average job execution time (μs).

however, that the approach to group actions is inadequate due to the rapid convergence of gradients. The action probabilities quickly devolve to the local minima, and the losses in policy and value increase exponentially. Also, a set of group actions and their respective returns cannot be distributed to individual actions, since each task has a respective reward and estimated return or value in the task duration. On the other hand, individual action selection with the EIM technique converts a varying action problem into a conventional sequential decision-making problem, with no need to be aware of invalid actions from the agent perspective. Fig. 12 shows that the policy with independent actions produces much higher values in the expected returns by 270% and has improved run-time performance by 30%. We applied the EIM technique to both approaches. Additionally, we report the explained variance, EV, denoted by (16).

$$\mathrm{EV}(s_t) = \frac{1 - \mathbb{V}[G(s_t) - \hat{G}(s_t)]}{\mathbb{V}[G(s_t)]}, \tag{16}$$

where $G(s_t)$ is empirical return of state s_t and $\hat{G}(s_t)$ is predicted return of state s_t. EV measures the difference between the expected return and the predicted return [14]. By observing the decreases in explained variance for the group action, we empirically validate that group action does not fully understand the environment while reconciling the experiences.

V. CONCLUSION AND FUTURE WORK

In this paper, we unveil myths and realities of DRL-based SoC job/task schedulers. We identify key practical challenges in designing high-fidelity neural SoC schedulers: (1) varying action sets, (2) high degree of heterogeneity in incoming jobs and available SoC compute resources, and (3) misalignment between agent interactions and delayed rewards. We propose and analyze a novel end-to-end neural scheduler (SoCRATES) by detailing its core technique (EIM) which aligns returns with proper time-varying agent experiences. EIM successfully addresses the aforementioned challenges, endowing SoCRATES with a significant gain in average latency and generalized performance over a wide range of job structures and PE performances. We also rationalize the underlying reasons behind the substantial performance improvement in existing neural schedulers with EIM by examining actual PE usages and disparate action designs. Through extensive experiments, we discover that CCR significantly impacts the performance of neural SoC schedulers even with EIM. At the same time, we find that the action of rescheduling task assignments by heuristic schedulers leads to significant performance gain under certain operational conditions, often outperforming neural counterparts.

With these findings, we intend to investigate further whether EIM technique can bring additional performance gains in other learning-based and planning algorithms, both empirically and theoretically. Further research on analyzing the performance bounds of the EIM technique is being conducted. With the advantage of task rescheduling in heuristic schedulers, we plan to improve neural schedulers by converting such technique to a differential function and integrating it into the optimization. Alternatively, offline reinforcement learning using expert or trace replay [3] is another possible approach to improve neural schedulers. Moreover, leveraging the structure of the underlying action space to parameterize the policy is a candidate approach to tackle a varying action set [10]. We also plan to leverage GNNs to bestow the structural knowledge from job DAGs [50], and demonstrate the performance gain of the improved neural schedulers by using the Compiler Integrated Extensible DSSoC Runtime (CEDR) tool, a successor to DS3 emulator, as it enables the gathering of low-level and fine-grain timing and performance counter characteristics [29].

APPENDIX A
JOB DAG CONSTRUCTION

The simulator can synthesize a variety of workloads given the job profile and hyperparameters, which are described in Section II-A1. First, we compute average values of widths, $\overline{w}$, and depths, $\overline{d}$, with the hyperparameter α based on the job model description in Section II-A1. We compute the number of nodes by $w \sim \max(1, \mathcal{N}(\lfloor\overline{w}\rfloor, 0))$ per $\overline{d} - 2$ job levels. Here, we exclude two levels in which the HEAD and TAIL nodes are located. Then we check whether the total number of nodes matches v (the number of nodes). If the total number of nodes is less or greater than v, then we randomly select nodes from the job DAG and add/subtract them in order to exactly have v nodes. As illustrated in Fig. 13, small α generates deep but narrow job graphs (left figure), and large α generates shallow but wide job graphs (right figure). Next, the job model generates the task dependency by the following iterative process. Let the number of predecessors and the number of nodes at level l by $|\mathrm{pred}(n_i)|$ and $|l|$, respectively. Then, the number of dependent tasks for node i at level l is computed by $\max(1, \min(\mathcal{N}(\frac{|l-1|}{3}, 0), |l-1|))$. We connect n_i to randomly selected $|\mathrm{pred}(n_i)|$ nodes in $l-1$ level.

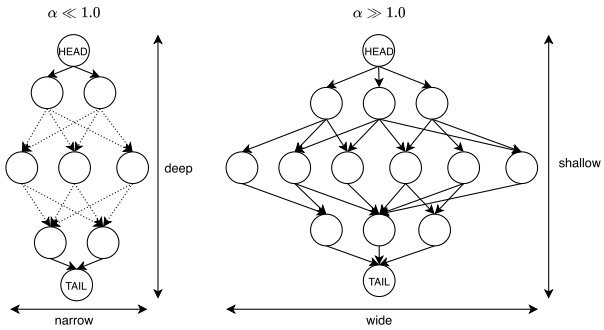


FIGURE 13. An illustration of two types of job DAGs based on α. Left diagram shows that small α generates a deep but narrow job graph. Dotted lines in the middle represent hidden nodes and edges. Right diagram shows that large α generates a shallow but wide job graph.

APPENDIX B
DESCRIPTION OF REWARD FUNCTIONS

Two types of dense and sparse reward functions are used to validate the efficacy of EIM technique. The reward functions are described as follows.

$$R_{\text{dense}}(\text{clk}) = C_1 \cdot |\hat{\mathcal{G}}_{\text{comp}}| + C_2 \tag{17}$$

$$R_{\text{dense2}}(\text{clk}) = C_1 \cdot |\hat{\mathcal{G}}_{\text{comp}}| \tag{18}$$

$$R_{\text{sparse}}(\text{clk}) = 0 \cdot \mathbb{1}_{[\text{clk}<\text{CLK}-m]} + C_1 \cdot |\hat{\mathcal{G}}_{\text{comp}}| \cdot \mathbb{1}_{[\text{clk}\geq\text{CLK}-m]} \tag{19}$$

$$R_{\text{sparse2}}(\text{clk}) = 0 \cdot \mathbb{1}_{[\text{clk}\neq\text{CLK}]} + C_1 \cdot |\hat{\mathcal{G}}_{\text{comp}}| \cdot \mathbb{1}_{[\text{clk}=\text{CLK}]}, \tag{20}$$

where $|\hat{\mathcal{G}}_{\text{comp}}|$ is the number of newly completed jobs at clk, CLK is the end of simulation length, and m is the number of the lastly completed tasks. C_1 and C_2 are the weights of job completion bonus and penalty for clock signal, respectively. We set $+50$ for C_1 and -0.5 for C_2 empirically. All reward functions are computed per simulation clock signal.

REFERENCES

[1] *ODROID-XU3*. Accessed: Mar. 3, 2019. [Online]. Available: https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[3] R. Agarwal, D. Schuurmans, and M. Norouzi, "An optimistic perspective on offline reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, Nov. 2020, pp. 104–114.

[4] A. Amarnath, S. Pal, H. T. Kassa, A. Vega, A. Buyuktosunoglu, H. Franke, J.-D. Wellman, R. Dreslinski, and P. Bose, "Heterogeneity-aware scheduling on SoCs for autonomous vehicles," *IEEE Comput. Archit. Lett.*, vol. 20, no. 2, pp. 82–85, Jul. 2021.

[5] S. E. Arda, A. Krishnakumar, A. A. Goksoy, N. Kumbhare, J. Mack, A. L. Sartor, A. Akoglu, R. Marculescu, and U. Y. Ogras, "DS3: A system-level domain-specific system-on-chip simulation framework," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1248–1262, Aug. 2020.

[6] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids," in *Proc. IEEE Int. Symp. Cluster Comput. Grid*, vol. 2, May 2005, pp. 759–767.

[7] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. D. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, pp. 70–93, Jan. 2016.

[9] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Application*, vol. 24. Springer, 2011.

[10] Y. Chandak, G. Theocharous, C. Nota, and P. Thomas, "Lifelong learning with a changing action set," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, 2020, pp. 3373–3380.

[11] M. Cheong, H. Lee, I. Yeom, and H. Woo, "SCARL: Attentive reinforcement learning-based scheduling in a multi-resource heterogeneous cluster," *IEEE Access*, vol. 7, pp. 153432–153444, 2019.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[13] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," Github, Tech. Rep., 2017. [Online]. Available: https://github.com/openai/baselines

[14] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. (2017). *OpenAI Baselines*. [Online]. Available: https://github.com/openai/baselines

[15] H. Djigal, J. Feng, J. Lu, and J. Ge, "IPPTS: An efficient algorithm for scientific workflow scheduling in heterogeneous computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1057–1071, May 2021.

[16] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1263–1272.

[17] A. A. Goksoy, A. Krishnakumar, M. S. Hassan, A. J. Farcas, A. Akoglu, R. Marculescu, and U. Y. Ogras, "DAS: Dynamic adaptive scheduling for energy-efficient heterogeneous SoCs," *IEEE Embedded Syst. Lett.*, vol. 14, no. 1, pp. 51–54, Mar. 2022.

[18] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2019, pp. 485–500.

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

[20] C. M. Holt, "Novel learning-based task schedulers for domain-specific SoCs," Ph.D. thesis, Dept. Elect. Comput. Eng., Arizona State Univ., Tempe, AZ, USA, 2020.

[21] 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)*, Jul. 2019, pp. 2037–2046.

[22] B. Keshanchi, A. Souri, and N. Navimipour, "An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: Formal verification, simulation, and statistical testing," *J. Syst. Softw.*, vol. 124, pp. 1–21, Feb. 2017.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[24] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 12, 1999, pp. 1–7.

[25] A. N. Krishnakumar, "Design run-time resource management of domain-specific systems on chip (DSSoCs)," Ph.D. thesis, Dept. Elect. Comput. Eng., Univ. Wisconsin-Madison, Madison, WI, USA, 2022.

[26] H.-L. Kung, S.-J. Yang, and K.-C. Huang, "An improved Monte Carlo tree search approach to workflow scheduling," *Connection Sci.*, vol. 34, no. 1, pp. 1221–1251, Dec. 2022.

[27] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "AlloX: Compute allocation in hybrid clusters," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.

[28] J. Mack, S. E. Arda, U. Y. Ogras, and A. Akoglu, "Performant, multi-objective scheduling of highly interleaved task graphs on heterogeneous system on chip devices," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 9, pp. 2148–2162, Sep. 2022.

[29] J. Mack, S. Hassan, N. Kumbhare, M. C. Gonzalez, and A. Akoglu, "CEDR—A compiler-integrated, extensible DSSoC runtime," *ACM Trans. Embedded Comput. Syst. (TECS)*, 2022, pp. 1–10.

[30] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2020, pp. 289–304.
[31] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, Nov. 2016, pp. 50–56.
[32] 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.*, Aug. 2019, pp. 270–288.
[33] M. Mettler, M. Rapp, H. Khdr, D. Mueller-Gritschneder, J. Henkel, and U. Schlichtmann, "An FPGA-based approach to evaluate thermal and resource management strategies of many-core processors," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 3, pp. 1–24, Sep. 2022.
[34] K. Moazzemi, "Runtime resource management of emerging applications in heterogeneous architectures," Ph.D. thesis, Dept. Comput. Eng., Univ. California, Irvine, CA, USA, 2020.
[35] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 481–498.
[36] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proc. ICML*, vol. 99, Jun. 1999, pp. 278–287.
[37] T. T. Sung, V. Chockalingam, A. Yahja, and B. Ryu, "Neural heterogeneous scheduler," 2019, *arXiv:1906.03724*.
[38] T. T. Sung, J. Ha, J. Kim, A. Yahja, C.-B. Sohn, and B. Ryu, "DeepSoCS: A neural scheduler for heterogeneous system-on-chip (SoC) resource scheduling," *Electronics*, vol. 9, no. 6, p. 936, Jun. 2020.
[39] T. T. Sung and B. Ryu, "A scalable and reproducible system-on-chip simulation for reinforcement learning," 2021, *arXiv:2104.13187*.
[40] T. T. Sung and B. Ryu, "SoCRATES: System-on-chip resource adaptive scheduling using deep reinforcement learning," in *Proc. 20th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2021, pp. 496–501.
[41] R. S. Sutton and A. G. Barto, "*Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[42] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 12, 1999, pp. 1–7.
[43] H. Tian, Y. Zheng, and W. Wang, "Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba cloud," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2019, pp. 139–151.
[44] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proc. 8th Heterogeneous Comput. Workshop (HCW)*, 1999, pp. 3–14.
[45] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proc. 11th Eur. Conf. Comput. Syst.*, Apr. 2016, pp. 1–16.
[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.
[47] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, Oct. 2013, pp. 1–16.
[48] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, and J. Quan, "StarCraft II: A new challenge for reinforcement learning," 2017, *arXiv:1708.04782*.
[49] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, and Q. Zhang, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2018, pp. 595–610.
[50] Y. Yu, J. Chen, T. Gao, and M. Yu, "DAG-GNN: DAG structure learning with graph neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 7154–7163.

TEGG TAEKYONG SUNG received the Ph.D. degree from Kwangwoon University, South Korea. He has worked as a Research Assistant at Kwangwoon University and a Visiting Researcher with the Electronics and Telecommunications Research Institute, South Korea. He is currently an Expert in the field of deep reinforcement learning with over three years of research and industrial experience. He is also working as a Senior Research Scientist at EpiSys Science Inc. He has authored or coauthored more than ten publications. His research interests include real-world reinforcement learning, machine learning, and graph neural networks.

BO RYU (Member, IEEE) received the Ph.D. degree from Columbia University. He is currently the Founder and the President of EpiSci, a trusted AI autonomy development company whose research and development projects are sponsored by NASA and various the Department of Defense agencies, including DARPA. Prior to founding EpiSci, he worked in various technical positions at the Hughes Research Laboratories, Boeing, the San Diego Research Center, and Argon ST. He has authored or coauthored more than 50 publications and holds 15 U.S. patents.