GCN-based reinforcement learning approach for scheduling DAG applications

Julius Roeder[1][0000−0003−0455−1283], Andy D. Pimentel[2][0000−0002−2043−4469], and Clemens Grelck[3,4][0000−0003−3003−1388]

[1] University of Amsterdam, Amsterdam, The Netherlands j.roeder@uva.nl
[2] University of Amsterdam, Amsterdam, The Netherlands a.d.pimentel@uva.nl
[3] University of Amsterdam, Amsterdam, The Netherlands c.grelck@uva.nl
[4] Friedrich Schiller University Jena, Jena, Germany clemens.grelck@uni-jena.de

Abstract. Applications in various fields such as embedded systems or High-Performance-Computing are often represented as Directed Acyclic Graphs (DAG), also known as taskgraphs. DAGs represent the data flow between tasks in an application and can be used for scheduling. When scheduling taskgraphs, a scheduler needs to decide when and on which core each task is executed, while minimising the runtime of the schedule. This paper explores offline scheduling of dependent tasks using a Reinforcement Learning (RL) approach. We propose two RL schedulers, one using a Fully Connected Network (FCN) and another one using a Graph Convolutional Network (GCN). First, we detail the different components of our two RL schedulers and illustrate how they schedule a task. Then, we compare our RL schedulers to a Forward List Scheduling (FLS) approach based on two different datasets. We demonstrate that our GCN-based scheduler produces schedules that are as good or better than the schedules produced by the FLS approach in over 85% of the cases for a dataset with small taskgraphs. The same scheduler performs very similar to the FLS scheduler (at most 5% degradation) in almost 76% of the cases for a more challenging dataset.

Keywords: DAGs, static scheduling, reinforcement learning, graph convolutional networks

1 Introduction

Directed Acyclic Graphs (DAGs) can be used in various fields (e.g. embedded systems, High-Performace-Computing) to represent applications. In a DAG, the nodes represent tasks and the edges between nodes represent the data dependency between tasks. Applications that can be represented as DAGs include, among others, augmented reality (AR) and AI applications in robotics and automotive [2], computer vision applications for precision agriculture [17], big data analytics applications that are implemented with Hive, Spark or Tez.

These applications are executed on multi-core and many-core platforms. In the area of embedded systems the Nvidia Jetson lineup [1] and the Odroid-XU4

[10] are good examples. To fully utilise the hardware we need to ensure that our applications can take advantage of the multiple CPU cores, the GPU and other available accelerators. Targeting the different available Compute Units (CU) (e.g. CPU, GPU) can be done during the scheduling of an application.

It is advantageous for scheduling to represent applications as DAGs because the DAG naturally provides options for concurrency and thus simplifies targeting multiple CUs. However, this concurrency also provides a challenge, as the scheduler also has to adhere to the data dependency inferred from the DAG (i.e. partial ordering). This process of deciding when and where a task of an application is executed is called scheduling. Scheduling can be done online (i.e. dynamically at runtime) or offline (i.e. statically). In this paper, we focus on static scheduling as dynamic scheduling can introduce significant overhead which may be problematic, especially for embedded systems.

Previous research on scheduling DAG applications mainly focused on using heuristics such as Forward List Scheduling (FLS) or Integer Linear Programming (ILP) (for surveys see [3, 8, 23, 22]). The later provides an optimal solution (e.g. minimising makespan, also known as execution time or run-time)), but does not scale well with increasing state space. The state space of a scheduling problem can depend on many different criteria such as the number of tasks in an application and the number of CUs available. Optimising for energy consumption adds another dimension to the state space. Heuristics scale better with the problem size. However, most scheduling heuristics require a total order [19, 18, 24, 13]. Thus, the first step in scheduling heuristics is to rank all tasks in a DAG (while preserving the partial order) and then scheduling them one-by-one in a greedy fashion. It has been shown that no ranking strategy always outperforms all other ranking strategies (see [19, 18]). Finding near-optimal static schedules for larger DAGs within a reasonable solving time is still an open problem.

Artificial intelligence (AI) may provide a feasible solution. However, supervised learning is not a suitable approach because it requires (large) datasets of examples with solutions (labels). This is especially problematic for large state-spaces, as for example in [18] where finding an optimal solution for DAGs containing more than 15 tasks takes more than 24 hours. Extrapolating the execution times from [18] shows that finding an optimal solution for a DAGs of 100 tasks would take approximately 35 years. Thus, we could never make a large enough dataset for more complex problems. Additionally, the authors [18] show that, other AI approaches, such as evolutionary algorithms, are also not well suited for the task at hand. Reinforcement Learning (RL), however, may still provide a promising approach. Recent advances in Reinforcement Learning have enabled computers to find good solutions for a variety of challenges, e.g., RL methods can build near-optimal solutions (up to 100 nodes) for combinatorial problems such as the Travelling Salesmen Problem [15]. RL approaches combined with supervised learning can schedule DAG task graphs and outperform heuristics such as Heterogeneous Earliest Finish Time (HEFT), Critical-Path-on-a-Processor (CPOP) and Graphene [26, 13]. The combination of RL and supervised methods was required to stabilise the training. However, by pre-training

neural networks in a supervised manner, they learn to imitate heuristics, whereas it has been shown that learning from scratch can outperform both heuristics and humans [21].

We propose a Deep Q-learning (DQN) approach that learns to build offline schedules from scratch (i.e., not relying on supervised learning to learn to imitate a heuristic). We propose two different RL-agents: one with a Fully Connected Neural Network (FCNN) backbone, and one with a Graph Convolutional Neural Network (GCN) backbone that can leverage graph information. To train and evaluate the models, we build two large datasets of 11,000 DAG taskgraphs each: one with small and simple graphs, and one with larger and complex graphs. Our experiments show that a Q-learning approach quickly learns static DAG scheduling without pre-training. Our experiments further show a comparison between a Forward List Scheduling (FLS) algorithm and the schedules generated by the two RL schedulers. Lastly, we show that incorporating graph information via GCN layers significantly improves the scheduling of DAG applications. Both the code[5] and the two datasets[6] are open source.

The remainder of this paper is structured as follows. In Section 2, we give a high level explanation of our RL-based scheduling approach, explain our DAG task model, layout the target architecture and discuss the most important components of our novel RL-scheduler. In Section 3, we discuss our experimental setup, followed by the results in Section 4. Section 5 discusses the related work. Lastly, Section 6 presents our conclusions and future work.

2 RL Scheduling

Let us start with a high-level overview of our RL scheduling approach. Figure 1 shows the interactions between the RL agent and the scheduling environment.

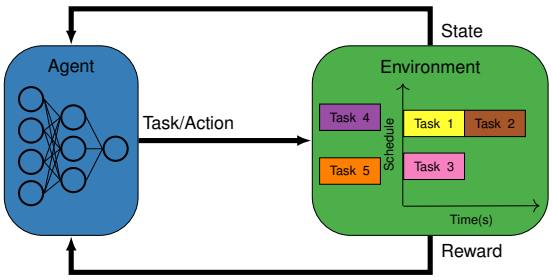


Fig. 1: Reinforcement Learning Framework

The environment updates the ready-queue at the beginning of each scheduling step, i.e., it collects all tasks that can be scheduled. Only tasks whose prede-

[5] https://bitbucket.org/jroeder/simple_rl_scheduling
[6] https://bitbucket.org/jroeder/gnn_tgff_data

cessors have already been scheduled can be scheduled. Hence, at any given point in time, we might have multiple tasks that can be scheduled.

Then the features of all tasks (states) in the ready-queue are collected and passed to the agent. The agent evaluates all eligible tasks and selects the task/action pair with the highest expected reward.

The task/action pair is returned to the environment, and the environment adds the task to the schedule in a *as-soon-as-possible* fashion, respecting both predecessor run-times and preventing tasks from being scheduled to execute at the same time on the same CU. The environment then returns the reward for the task/action pair and the states for the next tasks in the ready-queue. This is repeated until all tasks in all applications have been added to the schedule.

2.1 System Model

Task Model. We consider applications represented as Directed Acyclic Graphs (DAG). In a graph, $G = (\tau, E)$ the set of nodes/vertices τ represents the tasks, and the set of edges E represents data dependencies between tasks, i.e., a source task needs to be completed before the corresponding sink task may start executing. Our task model supports multiple sources and sinks. Additionally, we support a multi-graph setup (i.e., multiple applications).

A task is a sub-part of an application that needs a certain input, then executes without additional input until it finalises and passes its output to the following tasks in the application. Each task has a runtime, also called worst-case execution time (WCET). Our model does not limit the number of incoming or outgoing edges of a task. We assume that multiple tasks cannot run concurrently on one processing unit.

Architecture Model Our approach is fully platform-independent and can be applied to a wide range of homogeneous system architectures. The number of Compute Units (CU) can be altered via a parameter (the number of actions the agent can make). In this paper, we only focus on homogeneous quad-core systems to determine the feasibility of RL based schedulers. However, the model could easily be extended to heterogeneous systems by increasing the number of CU and including additional features.

2.2 RL Scheduler Components

Next, we give a short introduction to the various components of our reinforcement learning scheduler. One main part of the scheduler is the neural network agent that makes all the decisions. We investigate two different agents: one based on a Fully Connected Neural Network (FCNN), and an extention of the first one by incorporating Graph Convolutional (GCN) layers to leverage additional information inherent to the graph structure of the DAG.

Environment. The environment contains the task graph, the schedule and a representation of the target architecture. It can evaluate the impact of different scheduling decisions and update its internal states.

Fully Connected Agent. The main backbone of our agent consists of a fully connected neural network (FCNN). The network consists of *4* layers having *2048, 2048, 4096, 4096* neurons, respectively. We use ELU activation functions after each layer [5]. This network architecture performed the best across a range of different configurations while searching the hyperparameter space. The hyperparameter space search was performed using the Bayesian sweep function provided by *Weights and Biases*[4].

The input to a neural network depends on the type of neural network used. For our FCNN based approach, the state is a list of features for a task (i.e. node-specific and global features that are common to all tasks). The node-specific features are: (1) runtime of a task, (2) best start time at which a task can start (i.e. the end time across all predecessors), (3) actual start time of a task if it has been scheduled, and (4) target core if a task has been scheduled. The global features are: (1) normalised values of the min, max and mean of all tasks in the ready queue, (2) normalised values of the min, max and mean runtime of all tasks in the done queue, (3) number of tasks in the DAG, (4) number of tasks available for scheduling, and (5) number of tasks that still need to be scheduled. All normalised features are normalised with the maximum runtime of any task in the graph. This results in a total of 13 features for each task.

Graph Convolutional Network Agent. As our problem is in the form of a graph it was natural to turn to GCNs [14] in order to attempt to leverage the additional information inherent to the graph structure. The input to our GCNs consist of the same node features as for the FCNN, plus the edge information (i.e. which nodes are connected). We propose 3 different types of edge information: a node's predecessors, all previously scheduled nodes (i.e. they may hold information about gaps in the schedule) or all successors of a node. We create three different GCNs, that each take as input the node features and one of the three edge information. Each GCN consists of 4 SAGEConv [9] layers with *8, 16, 32, 64* neurons per layer respectively. Each layer is followed by an ELU activation function. The input to each GCN are the node information (runtime, best start time, actual start time and target core). However, the edge information for each GCN differ slightly depending on whether it is supposed to learn about predecessors, previously scheduled nodes or successor nodes. The output of the three GCNs is then concatenated to the original node information and to the same global features as for the FCNN agent. All this information is fed into the same FCNN agent as above (4 layers with *2048, 2048, 4096, 4096* neurons, respectively) and ELU activation functions to return what is the most valuable action.

Actions. The action is the CU on which a given task is scheduled. For example, in the case of a quad-core system, the action space is between 0 and 3. The number of possible actions depends on the target system.

Rewards. The reward function (Equation (1)) returns the value of a given state s_t. In our case, the reward is the negative release time ($-rt_0$) of the action (i.e. start time of a task) plus the expected reward of future actions, where γ is the discount rate ($[0, 1)$) of future actions. We used a γ of 0.65. There are

no positive rewards; the best possible reward is 0. If a task (t_0) starts at the 5 second mark, the reward is -5 minus the expected reward of future actions. That means if we expect the next task (t_1) to start at the 8 second mark, then the reward for t_0 is -13.

$$V(s_0) = -rt_0 + E_{t=1}^{\infty}\left[\gamma^t \times (-rt_t)\right] \tag{1}$$

RL approach description. We use a double DQN approach [11] with fixed Q-targets, where two networks (*NN1* and *NN2*) are initialised with the same weights. *NN1* and *NN2* are used to update each other. A simplified representation of a single training step is shown in Figure 2. During training, the environment passes a state (S_t) to *NN1*, which predicts the *expected reward* of all actions at step *t*. The action with the highest *expected reward* is selected and passed to the environment, which evaluates it and computes a *reward*. At the same time, the environment passes the updated state S_{t+1} to *NN2*, which predicts the *expected reward* for the new state. The *reward* at t is combined with the *expected reward* at $t+1$ to form the *actual reward* at t. The *actual reward* is then used to update the weights of *NN1*. The weights of *NN2* are updated every τ steps with the weights of *NN1*. *NN2* is the network used for inference.

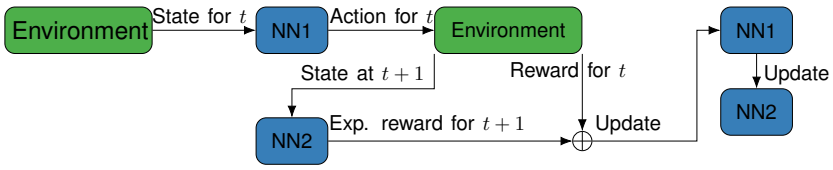


Fig. 2: RL agent training pipeline.

Furthermore, our approach uses prioritised experience replay [20], where training samples of higher impact are more likely to be in the training batch. The impact of a sample is the absolute percentage difference between the predicted and the actual reward.

3 Experiments

Data. We use Task Graphs For Free (*TGFF*) [7] to generate random DAG task graphs. TGFF generates 10,000 tasks, where each task has a different runtime. Using a random selection of the tasks, different DAGs are generated. We generate two datasets to run our experiments. One dataset with smaller, less diverse and simple graphs, and a second dataset with large, diverse and complex graphs.

The main difference between the two datasets is the number of tasks per graph. The small DAGs (Dataset 1) are set to 10 tasks with a multiplier of 1. This does not mean that all graphs have 10 tasks, as the number of tasks also

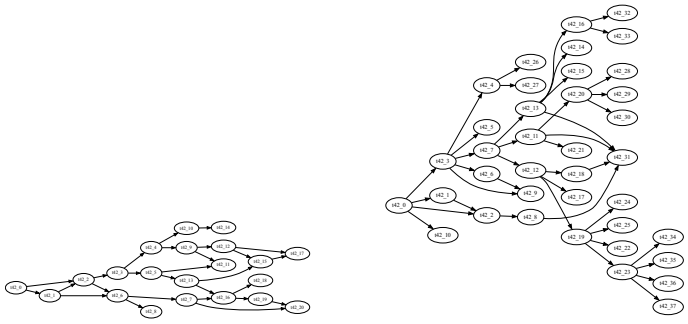
depends on other characteristics. The larger taskgrahs (Dataset 2) are set to an average of 20 tasks with a multiplier of 5. This means that the larger DAGs are more challenging as they, for example, contain more potential parallelism, which is especially important as the target system only has 4 cores. All datasets are roughly uniformly distributed with respect to the number of tasks in a DAG.

Both datasets consist of a training dataset with 10,000 task graphs and a test dataset containing 1,000 task graphs. The graphs in the test and training datasets were generated separately with different seeds. Table 1 contains a summary of the graph statistics for the small and large DAG datasets. The training and test dataset do not differ much with respect to the number of tasks in the DAGs.

Table 1: The table shows the statistics with respect to the number of tasks in the train and test sets that contain the small and large DAGs.

	Mean	Min.	Max.	Std.
Small Train	12.7	6	24	6.3
Small Test	13.0	6	24	6.4
Large Train	25.5	9	55	13.1
Large Test	25.5	9	54	13.4

The difference between the type of graphs generated for the two datasets can be well seen when comparing Figure 3a and Figure 3b.



(a) Example DAG from the small DAG test dataset.

(b) Example DAG from the large DAG test dataset.

Fig. 3: Comparing example DAGs from the two generated datasets.

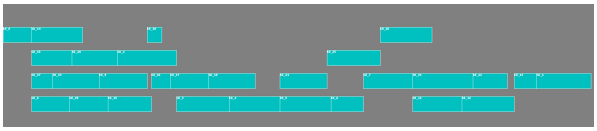
Comparison with existing method. We compare the RL generated schedules to schedules generated by a Forward List Scheduler (FLS) [6]. FLS first orders the tasks and then adds them one by one to the schedule without backtracking. FLS iteratively computes the impact on the makespan (i.e. run-time)

of scheduling a task on a specific compute unit (CU) (e.g. CPU, GPU) and greedily selects the best CU with respect to the makespan. The performance of FLS heavily depends on the initial ranking of the tasks. Thus, it is common practice to try multiple ranking algorithms as none consistently outperforms the others [19]. In this case, we use 3 different rankings: BFS, DFS and BFS with Laxity.

4 Results

In general, the RL scheduler learns to schedule DAGs quickly. Figure 4a shows the schedule produced by an untrained, randomly initialised RL agent. We can see that all 27 tasks from the original graph are simply put after each other on a single core. However, after some training the scheduler improves. Figure 4b shows a schedule produced for the same graph by the same RL agent after some training (before convergence). The decisions are not necessarily optimal but we can clearly observe that the scheduler learns that distributing tasks over different cores is better (i.e. increases its rewards).

(a) Schedule of a taskgraph with 27 tasks produced by one of our untrained RL schedulers.



(b) Schedule of a taskgraph with 27 tasks produced by one of our trained RL schedulers.

Fig. 4: Comparing schedules generated by an untrained and a trained RL-based scheduler.

In Sections 4.1 to 4.4, we discuss the performance of the two different schedulers (FCNN and GCN) with regard to the two different datasets (Dataset 1 & Dataset 2). All four combinations were allowed to train for a similar number of epochs and the best performing neural network was selected.

4.1 Dataset 1 - FCNN Agent

The FCNN agent performs fairly well on the dataset consisting of smaller DAGs. The degradation distribution between the FCNN agent and the FLS scheduler can be seen in Figure 5. In 69.6% of the cases the FCNN agent produces schedules that are the same or better. In 90.0% of the DAGs the FCNN agent results in

schedules that perform similarly (at most 5% degradation) or better. The average degradation is 1.2% and at best the resulting schedule is 6.9% shorter than the schedule generated by the FLS approach. At worst, the FCNN scheduler results in a 19.7% higher makespan. Despite this good performance the FCNN scheduler had a L1Loss of 35.9 which is higher than the L1Loss of the GCNN scheduler.

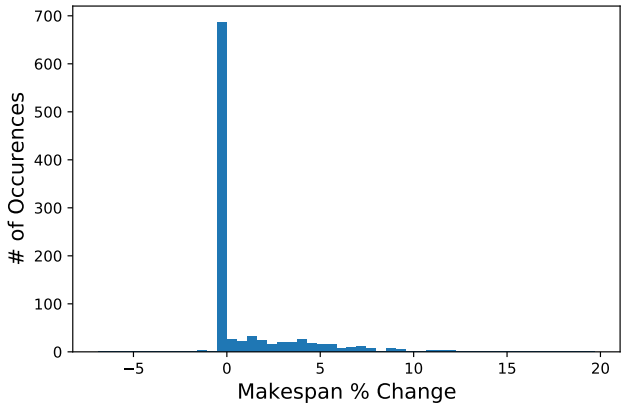


Fig. 5: Makespan degradation of the small DAG test dataset between the FCNN generated schedules and the FLS schedules.

4.2 Dataset 2 - FCNN Agent

The FCNN agent performs significantly worse for the dataset containing larger DAGs than for the dataset of small DAGs. The degradation spread is shown in Figure 6. Overall, the FCNN agent only manages to produce schedules that are the same or better in 18.9% of the DAGs. Additionally, it finds schedules that perform similarly (at most 5% degradation) or better in only 42.3% of the cases. Overall, the degradation is 7.1%. And at best, the generated schedule results in 6.5% lower makespan but at worst we see a degradation of 35.3%. The L1Loss (45.5) is higher than the L1Loss in Section 4.1. Showing that the additional complexity of the large DAGs and possibly the larger variance of DAGs may require a more advanced approach.

4.3 Dataset 1 - GCN Based network

On Dataset 1, the GCN agent performs better than the FCNN agent. The degradation is shown in Figure 7. The distribution looks similar to the one shown in Figure 5. Overall, the GCN approach generates schedules that are the same or better in 85.2% of the cases. And it finds schedules that perform similarly (at

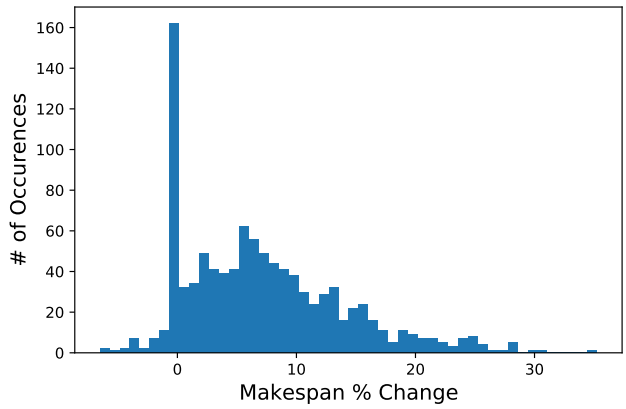


Fig. 6: Makespan degradation of the large DAG test dataset between the FCNN generated schedules and the FLS schedules.

most 5% degradation) or better in 98.1% of the cases. The average degradation is 0.29%. At best the schedule is 6.9% shorter and at worst the found schedule has a 20.4% longer makespan. One more difference between the FCNN scheduler and the GCN scheduler is the much lower L1Loss, which dropped to 5.9. This clearly shows that the three GCNs provide valuable information, even though, the information do not appear to add much value in the case of the smaller DAG dataset.

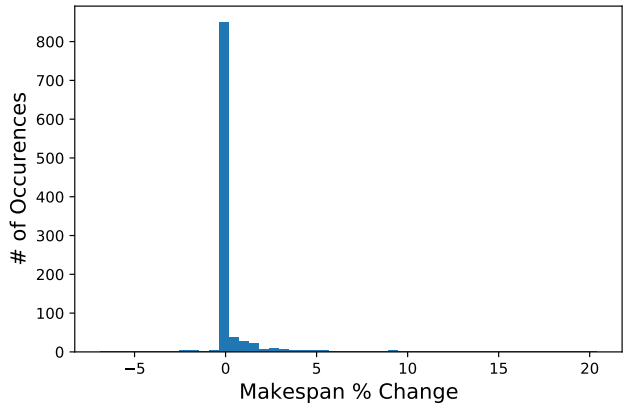


Fig. 7: Makespan degradation of the small DAG test dataset between the GCN based RL scheduler and the FLS scheduler.

4.4 Dataset 2 - GCN Based network

We can see a clear improvement in the schedules generated by the GCN scheduler in comparison to the FCNN scheduler for the dataset of large DAGs. This improvement can also be seen when comparing the degradation distributions in Figure 8 (GCN scheduler) and Figure 6 (FCNN scheduler). In total, we find that the GCN agent generates schedules that are the same or better in 38.7% of the cases. Furthermore, the GCN scheduler finds schedules that perform similarly (at most 5% degradation) or better in 75.6% of the cases. The average degradation drops from 7.1% for the FCNN agent to 2.8% for the GCNN agent. At best we see schedules that are 11.2% shorter and at worst the schedules are 34.4% longer than the FLS generated schedules. The final L1Loss is 11.0. In comparison, to the GCN approach on smaller DAGs this L1Loss is slightly higher. However, the L1Loss is also significantly lower than the L1Loss of the FCNN agent. This clearly shows that the additional information provided by the GCN layers is valuable.

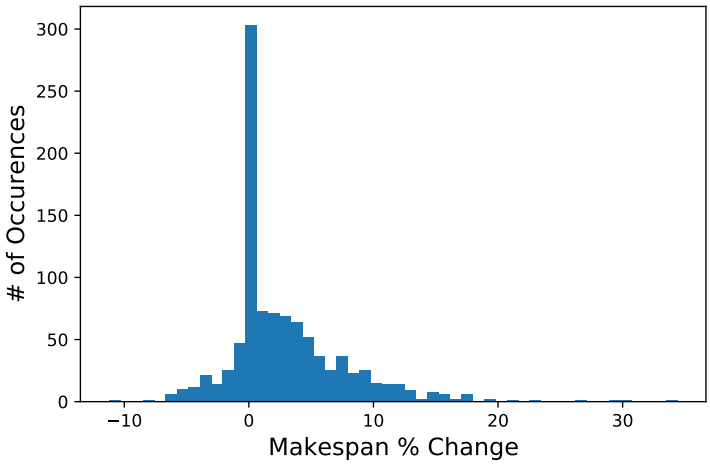


Fig. 8: Makespan degradation of the large DAG test dataset between the GCN based RL scheduler and the FLS scheduler.

Across all four experiments, we cannot draw conclusions on whether the number of tasks in a taskgraph impacts the performance of a RL-scheduler, i.e., a larger taskgraph does not necessarily lead to a higher degradation.

5 Comparative analysis with existing algorithms

Wu et al. [26] use the REINFORCE agent [25] from 1992 to schedule DAG taskgraphs. The paper shows that this approach outperforms Heterogeneous

Earliest Finish Time (HEFT) and Critical-Path-on-a-Processor (CPOP) by up to 25%. However, REINFORCE agents tend to be unstable in the training process. More modern approaches like our approach address this stability issue. Furthermore, the approach by Wu et al. depends on the original ranking of the tasks in the task graph.

Hu, Tu and Li [13] have proposed a new approach (called *Spear)* that uses Monte Carlo Tree Search (MCTS) combined with RL. *Spear* outperforms the Graphene heuristic by 20%. *Spear* determines the ranking of the tasks, i.e., it determines in what order the tasks are scheduled, whereas we use RL to schedule the task end-to-end. Additionally, *spear* initialises the network with supervised learning, i.e., it learns to imitate the behaviour of a heuristic. This means that the agent might learn undesirable behaviour from the heuristic. And is exactly the opposite of what we want, as it has been shown that RL agents are capable of learning strategies on their own and, in some cases outperforming both humans and heuristics [21].

Mao et al. [16] use Reinforcement Learning to schedule independent tasks, whereas we focus on dependent tasks. Hu et al. [12] introduce an RL agent for online scheduling of dependent tasks. Our approach focuses on offline scheduling as online scheduling can incur a high overhead on high-performance embedded systems.

6 Conclusion

Finding near-optimal static schedules for large DAGs in a reasonable solving time is still an open problem. To the best of our knowledge, we are the first to use DQN Reinforcement Learning to tackle this problem in an end-to-end fashion.

We show that RL-based schedulers can outperform FLS-based schedulers. The resulting schedules are up to 11.2% shorter than the corresponding FLS generated schedules. For the small DAG dataset (Dataset 1) our GCN approach generates schedules that are at most 5% worse in 98.1% of the cases. For the large DAG dataset (Dataset 2) our GCN approach generates schedules that are at most 5% worse in 75.6% of the cases. Furthermore, our experiments show that the additional information obtained by the GCN layers add value to our RL-based scheduler. However, this additional information only seems to result in significantly better schedules (on average) if the target dataset is more diverse or contains larger taskgraphs. Furthermore, we show that the selected reward function works (i.e. lower loss = better performance).

In the future, we plan to investigate deeper networks, the performance of the RL scheduler for heterogeneous systems and the use of sparse rewards. Additionally, we plan to investigate which one of the three GCNs adds most value. Lastly, we plan to experiment with policy learning instead of action-value learning.

Acknowledgement

We would like to thank the reviewers for their time and feedback.

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project).

This article is based upon work from COST Action CERCIRAS, supported by COST (European Cooperation in Science and Technology)

References

1. Nvidia Jetson. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`, accessed: 2023-01-21
2. Andreozzi, M., Gabrielli, G., Venu, B., Travaglini, G.: Industrial challenge 2022: A high-performance real-time case study on arm. In: ECRTS 2022. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2022)
3. Bambagini, M., Marinoni, M., Aydin, H., Buttazzo, G.: Energy-Aware Scheduling for Real-Time Systems. TECS **15**(1) (2016)
4. Biewald, L.: Experiment tracking with weights and biases (2020), `https://www.wandb.com/`, software available from wandb.com
5. Clevert, D., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289 (2015)
6. Cooper, K.D., Schielke, P.J., Subramanian, D.: An Experimental Evaluation of List Scheduling. TR98 **326** (1998)
7. Dick, R., Rhodes, D., Wolf, W.: Tgff: task graphs for free. In: 6th CODES/CASHE. IEEE (1998)
8. Gerards, M., Hurink, J., Hölzenspies, P.: A survey of offline algorithms for energy minimization under deadline constraints. J. of Scheduling **19**(1) (2016)
9. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)
10. Hardkernel Co., Ltd.: Odroid-XU4. `https://wiki.odroid.com/odroid-xu4/odroid-xu4`, accessed: 2019-09-06
11. Hasselt, H.: Double Q-learning. 24th NIPS **23**, 2613–2621 (2010)
12. Hu, Y., de Laat, C., Zhao, Z.: Learning workflow scheduling on multi-resource clusters. In: 2019 NAS. pp. 1–8. IEEE (2019)
13. Hu, Z., Tu, J., Li, B.: Spear: Optimized Dependency-Aware Task Scheduling with Deep Reinforcement Learning. In: 39th ICDCS. pp. 2037–2046. IEEE (2019)
14. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
15. Kool, W., van Hoof, H., Welling, M.: Attention, Learn to Solve Routing Problems! In: 7th ICLR (2019)
16. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource management with deep reinforcement learning. In: Proceedings of the 15th ACM workshop on hot topics in networks. pp. 50–56 (2016)
17. Patrício, D.I., Rieder, R.: Computer vision and artificial intelligence in precision agriculture for grain crops: A systematic review. Computers and electronics in agriculture **153**, 69–81 (2018)

18. Roeder, J., Rouxel, B., Altmeyer, S., Grelck, C.: Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems. In: 2021 36th SAC. pp. 501–510 (2021)
19. Rouxel, B., Skalistis, S., Derrien, S., Puaut, I.: Hiding communication delays in contention-free execution for spm-based multi-core architectures. In: 31st ECRTS19 (2019)
20. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized Experience Replay. arXiv:1511.05952 (2015)
21. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. Nature **550** (2017)
22. Singh, A.K., Dziurzanski, P., Mendis, H.R., Indrusiak, L.S.: A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. ACM Computing Surveys (CSUR) **50**(2), 1–40 (2017)
23. Singh, A.K., Shafique, M., Kumar, A., Henkel, J.: Mapping on multi/many-core systems: Survey of current and emerging trends. In: 2013 50th DAC. pp. 1–10. IEEE (2013)
24. Ullah Tariq, U., Ali, H., Liu, L., Panneerselvam, J., Zhai, X.: Energy-efficient Static Task Scheduling on VFI based NoC-HMPSoCs for Intelligent Edge Devices in Cyber-Physical Systems. TIST **1**(1) (2019)
25. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning **8**(3-4), 229–256 (1992)
26. Wu, Q., Wu, Z., Zhuang, Y., Cheng, Y.: Adaptive DAG Tasks Scheduling with Deep Reinforcement Learning. In: 19th ICA3PP. Springer (2018)