Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling

Cheng-Hsiang Chiu
Departmet of ECE
University of Wisconsin-Madison
chenghsiang.chiu@wisc.edu

Chedi Morchdi
Department of CSE
Texas A&M University
chedi.morchdi@tamu.edu

Yi Zhou
Department of CSE
Texas A&M University
yi.zhou@tamu.edu

Boyang Zhang
Department of ECE
University of Wisconsin-Madison
bzhang523@wisc.edu

Che Chang
Department of ECE
University of Wisconsin-Madison
cchang289@wisc.edu

Tsung-Wei Huang
Department of ECE
University of Wisconsin-Madison
tsung-wei.huang@wisc.edu

2024 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3503-8713-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPEC62836.2024.10938506

***Abstract*—Dynamic task graph scheduling (DTGS) has become a powerful tool for parallel and heterogeneous applications, such as static timing analysis and large-scale machine learning. DTGS allows applications to define the task graph structure on-the-fly, enabling concurrent task creations and task executions. However, to schedule tasks, DTGS relies on applications to define a topological order for the task graph. Existing algorithms for generating this order primarily rely on heuristics like level-by-level sorting, which lack adaptability to dynamic computing environments. This paper proposes a novel method that leverages reinforcement learning to generate topological orders for DTGS systems. We will delve into the details of our design and present a real-world use case. For instance, when scheduling a large task graph with 3.9 million tasks and 7.4 million dependencies in a large-scale static timing analysis workload, our method achieves a speedup of up to $1.52\times$ compared to the baseline.**

***Index Terms*—Dynamic Task Graph, Topological Orders, Reinforcement Learning.**

I. Introduction

Dynamic task graph scheduling (DTGS) has emerged as a powerful technique for processing parallel and heterogeneous applications, such as static timing analysis [1]–[25] and large-scale machine learning problems [26]–[29]. Unlike traditional loop-based models that explores parallelism across loops, DTGS represents function calls as tasks and dependencies between them as edges in a task graph. DTGS empowers applications to perform top-down optimization within complex parallel decomposition strategies involving numerous tasks and dependencies. A DTGS runtime then efficiently schedules these dependent tasks across a large pool of execution units with dynamic load balancing [30]. As a result, the parallel computing community has seen the rise of numerous successful DTGS libraries catering to various applications, such as OpenMP [31], Kokkos-DAG [32], PaRSEC [33], [34], HPX [35], Taskflow [36], [37], and AsyncTask [23].

To leverage the power of DTGS, applications define the task graph structure dynamically according to runtime variables and control-flow results. As tasks and dependencies are created on-the-fly, DTGS allows the task creation time to overlap with the task execution time, as shown in Figure 1. Thus, DTGS is flexible when dealing with many algorithms that frequently incorporate dynamic control flow in implementing irregular parallel decomposition strategies, such as electronic design automation (EDA) algorithms [1].

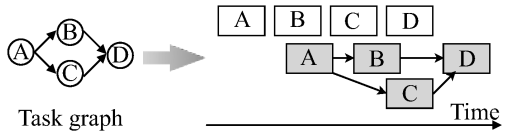

Fig. 1: Scheduling a dynamic task graph with four tasks and four edges. White rectangles denote the task creations and gray the task executions. Tasks are created in the topological order A-B-C-D. Task creations overlap with task executions.

To schedule tasks under the task dependency constraints, DTGS runtime requires applications to create tasks in a topological order of the task graph. For example, in Figure 1, four tasks should be created either in the topological order A-B-C-D or A-C-B-D. To obtain the order, topological sorting algorithms, such as Kahn's algorithm [38], are widely used. These heuristic-based algorithms generate orders primarily based on the graph structures, such as level-by-level sorting. However, such heuristic-based approaches have limitations. First, solely relying on graph structure lacks adaptability to dynamic changes in the computing environment. This can lead to suboptimal scheduling and can consume large scheduling resources due to the randomness involved in DTGS runtime's dynamic load balancing [30]. Second, heuristic algorithms generate deterministic orders. But, topological orders of a task graph are not unique and different topological orders for the same task graph can lead to substantial performance differences, as shown in Figure 2.

To overcome the limitations of heuristic approaches, we leverage recent advancements in reinforcement learning (RL) [20], [39], [40] and propose a method to interact with the computing environment while generating topological orders. We summarize our technical contributions as follows:

Fig. 2: Runtime of DTGS system finishing one EDA application with three different topological orders.

- We have proposed a reinforcement learning-based method to generate topological orders for the dynamic task graph scheduling applications. With the method, we are able to adapt to the computing environment throughout the whole decision-making process, allowing us to generate the topological order that achieves better runtime performance than the baseline.
- We have designed a new task graph encoding method to support our RL model. This technique allows us to encode the task graph dynamically according to the runtime status instead of encoding the whole graph.
- We have designed a new decision space categorization method to support our RL model. This method categorizes the timing-varying decision space into fixed number of categories, allowing us to reuse the trained RL model for various applications without the need for retraining.
- We have evaluated our method for generating the topological orders for a large-scale industrial static timing analysis (STA) application. With our proposed method, we are able to generate topological orders that achieve up to $1.52\times$ speedup over the baseline.

II. Background

We target scheduling a large-scale static timing analysis (STA) application, one of the most important steps in the entire EDA flow, and describe a STA workload as a task graph. The task graph consists of multiple nodes and edges, which represent the tasks and the dependencies among the tasks, respectively. Every task has a known workload and the task dependencies constrain the execution order of the tasks. Take the task graph shown in Figure 3 as an example. We describe an example circuit as a task graph of four tasks and four edges (or dependencies). The task dependencies require that task A must execute before task B and task C, and task D must execute after task B and task C.

To efficiently schedule the task graph, we build a DTGS system using a recently released dynamic task graph library, AsyncTask [23]. AsyncTask provides a clear and concise graph description language for applications to easily explore dynamic task graph parallelism. The expressiveness of AsyncTask's programming model improves our productivity when coding large and complex task graphs for the STA workloads. Listing 1 shows AsyncTask implementation of the task graph in Figure 3. We create four tasks in the topological order A-B-C-D and use AsyncTask's dependent_async API to create each task. Every task defines its own lambda as the first argument which is followed by a list of dependent tasks. Upon returning from dependent_async, we obtain a pair consisting of an instantiated task object and a future object holding the execution result of that task. After constructing all tasks, we call future_D.get to wait for task D to finish. Since we construct tasks in the order A-B-C-D, the completion of task D in turn signifies the completion of all preceding tasks.

```
int main(){
  Executor executor;
  // create four asynchronous tasks
  auto[A, future_A]=executor.dependent_async(
    [](){ printf("Running task A\n"); });
  auto[B, future_B]=executor.dependent_async(
    [](){ printf("Running task B\n"); }, A);
  auto[C, future_C]=executor.dependent_async(
    [](){ printf("Running task C\n");}, A);
  auto[D, future_D]=executor.dependent_async(
    [](){ printf("Running task D\n"); }, B, C);
  // wait for the task graph to finish via future_D
  future_D.get();
}
```

Listing 1: AsyncTask implementation of the task graph in Figure 3.

In Listing 1, we must create tasks in a topological order because we can not create a task until certain tasks it depends on have existed. For example, we need to create task A before task B. That explains why AsyncTask requires applications to create tasks in a topological order. To obtain a topological order of a task graph, we propose a method that incorporates a reinforcement learning (RL) model in DTGS. Figure 3 illustrates the system overview. After we describe a circuit as a task graph, the trained RL model reads in the task graph and outputs a topological order of the task graph. Then the DTGS creates all of the tasks based on the order and overlaps the task executions. We give the details of the trained RL model in Section III.

III. Proposed Method

Runtime status governs the macro-scale performance in a task scheduling system [20]. To take the runtime status into account, we propose a reinforcement leaning model to interact with the computing environment while generating a topological order for the DTGS application. In this section, we first formulate the dynamic task graph scheduling problem as a reinforcement learning (RL) problem and then apply the Deep Q-Learning algorithm [41] to train a good RL policy. Figure 4 shows the training process in the DTGS application.

A. Reinforcement Learning Formulation

To formulate the dynamic task graph scheduling problem as a reinforcement learning problem, we need to define four major components as follows:

- **State**. A state encodes the information that the RL agent needs to suggest an action. We encode the normalized workloads of ready tasks (tasks whose dependencies

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

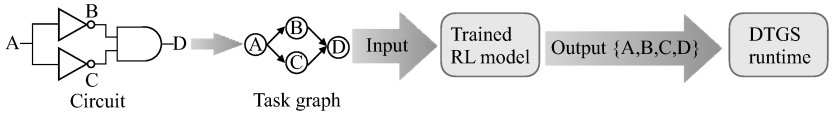

Fig. 3: System overview. An example circuit is described as a task graph of four tasks and four edges. The trained RL model reads in the task graph and generates a topological order A,B,C,D. The DTGS runtime creates the four tasks in the order and executes them under the dependency constraint.

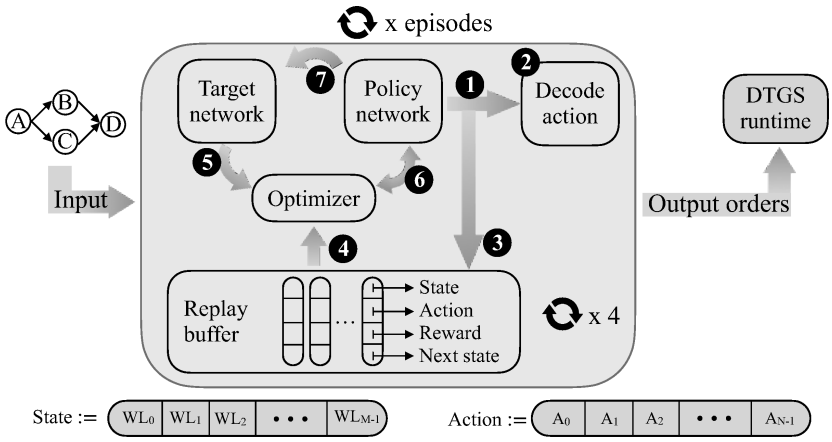

Fig. 4: Overview of the training process. The input is a task graph of four tasks and four edges. The output is a topological order of the four tasks. The training process consists of seven steps, which iterates four times as there are four tasks in the input task graph. The whole training would iterate the task graph for several episodes.

are resolved) and the graph structure in a vector of M coordinates as the state called State. M denotes the number of maximum fanout (or the successor tasks) of a task in the task graph. Each coordinate encodes the total workloads (WL) of ready tasks of the same number of fanout. For example, in Figure 4, task A is the ready task and has two fanouts – task B and task C. The current state would include task A's workload in the third coordinate, State[2], and the other coordinates are zero.

- **Action**. An action describes the operation that the RL agent suggests. In this paper, the RL agent needs to select the next task from all the ready tasks. We note that the number of the ready tasks may vary over time. However, the RL agent can only select a task from a *fixed* number of ready tasks. To accommodate the constraint, we categorize the ready tasks to N groups based on the workloads. For example, in Figure 4, when the RL agent needs to select the next task between task B (suppose being categorized to group 0) and task C (suppose being categorized to group 1), the RL agent selects the next task from a certain *group* (say group 0) rather than selecting the next task directly. This allows us to maintain a consistent action space size regardless of the number of ready tasks at any given time.
- **State Transition**. After an action is performed (i.e., the next task is created by the DTGS runtime), the current state will transfer to a new state. For example, in Figure 4, after task A is created by the DTGS runtime, the new state will have the sum of task B's and C's workloads at the second coordinate (State[1]) because both task B and C are now ready tasks and have one fanout.
- **Reward**. After the RL agent takes an action, we receive a reward feedback from the computing environment. This reward signal guides the agent's learning process towards actions that optimize a specific resource utilization metric. Here, we focus on minimize free memory space (FMS) to keep all computing units as busy as possible. Therefore, we design the following reward to reflect this objective:

$$\text{reward} = -(FMS^{\text{after}} - FMS^{\text{before}}), \qquad (1)$$

where FMS^{before} denotes the normalized FMS before performing the action and FMS^{after} denotes the normalized FMS after the action. Note that minimizing FMS is equivalent to maximizing the reward.

Next, we discuss how to train a good RL policy using the Deep Q-Learning algorithm to maximize the accumulated reward over time.

B. Deep Q-Learning

Deep Q-Learning is a popular algorithm that aims to learn the optimal policy that maximizes the expected accumulated reward over time [41]. We apply this algorithm to solve our dynamic task graph scheduling problem which is now formulated as a RL problem. Figure 4 illustrates the Deep Q-Learning algorithm for our scheduling system in seven steps.

❶ **Generate an action**. Based on the current state, the policy network generates an action. The policy network is a feed forward neural network, as shown in Figure 5. The

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

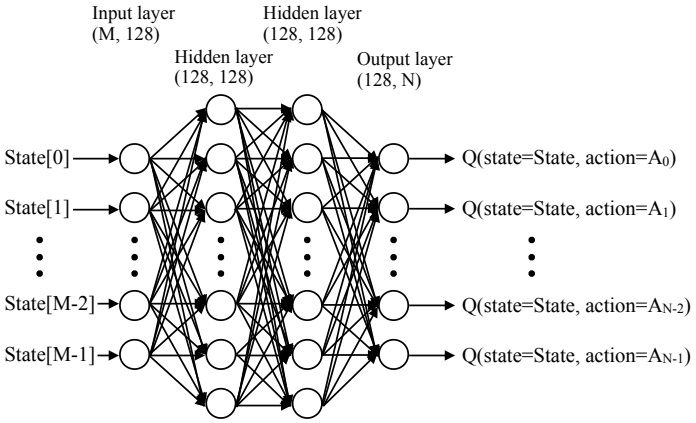

Fig. 5: Illustration of the policy network architecture. There is one input layer of dimension $M * 128$, two hidden layers of dimension $128 * 128$, and one output layer of dimension $128 * N$. The activation function is ReLU [42].

network reads in the current state vector with dimension M as the input. It then processes the information through two hidden layers, each with a dimension of 128×128. Finally, the network outputs a set of Q-values corresponding to each of the N possible actions. A Q-value represents the expected reward associated with taking a specific action in a given state. In essence, it estimates the long-term benefit of choosing an action. To balance exploration and exploitation during learning, we employ the ϵ-greedy strategy. During exploration (with probability ϵ), we randomly select an action from the available options. This helps the RL agent discover potentially better actions outside of its current knowledge. In contrast, during exploitation (with probability $1-\epsilon$), we select the action with the highest Q-value, favoring actions predicted to yield the best rewards in the current state.

The parameter ϵ gradually decays over time, following the equation below,

$$\epsilon = \epsilon^{end} + (\epsilon^{start} - \epsilon^{end}) * e^{-1 * steps/\epsilon^{decay}}, \quad (2)$$

where $steps$ records the number of processed steps so far, ϵ^{start} denotes the initial value, ϵ^{end} denotes the final value, and ϵ^{decay} denotes the decay rate. This decay encourages exploration in the initial stages to learn the environment and transitions to exploitation later for optimal performance.

As discussed in Section III-A, an action represents a group of ready tasks with the same range of workloads. If the selected group is empty (meaning no ready tasks fall within that workload category), we implement the following strategies. We randomly select another action at the exploration phase, or select the group with the next highest Q-value at the exploitation phase.

❷ **Decode the action to the next task**. Upon selecting a non-empty group (i.e., containing ready tasks), we randomly select a task from that group and forward the selected task to the DTGS runtime for task creation and execution.

❸ **Store the transition information in the replay buffer**. To improve the agent's learning efficiency, we utilize a replay buffer. This buffer stores transitions as tuples, allowing the agent to revisit and learn from past experiences multiple times. Each tuple in the replay buffer contains four key elements:

- Current State (s): The state representation captures information relevant to the decision-making process at a specific point in time.
- Selected Action (a): The action is selected by the agent based on the current state.
- Reward (r): The reward signal is received from the environment after taking the selected action. This feedback guides the agent towards actions that optimize resource utilization (here, we minimize the free memory space).
- Next State ($s^{'}$): The state representation after the selected action is executed, reflecting the updated environment.

By revisiting these transitions during training, the agent can learn from a broader set of experiences and improve its decision-making capabilities.

❹ **Sample the replay buffer**. During training, we leverage batch sampling to learn from its past experiences stored in the replay buffer. This involves randomly selecting a mini-batch of data points (size denoted by B) from the buffer. By randomly selecting data points, we help to de-correlate the training samples. This is important because consecutive transitions in the replay buffer might be highly correlated, potentially hindering the learning process. De-correlated samples provide a more diverse set of experiences for the agent to learn from, improving the efficiency and effectiveness of training.

❺ **Calculate the expected Q-value**. We incorporate a target network, which mirrors the architecture of the policy network (as illustrated in Figure 5). However, we do not update the target network's weights as frequently as the policy network's. This separation is crucial for reducing overestimation bias [43]. The target network addresses this issue by providing an unbiased estimate of the Q-value for the next state. We achieve this by using the target network to evaluate the expected Q-value of the action selected by the policy network, as shown in the following equation:

$$Q^{expected} = r + \gamma \max_{a} Q(s', a) \quad (3)$$

where γ represents the discount factor, which balances the importance of immediate rewards against the potential value of future rewards.

❻ **Optimize the model**. During training, we calculate the difference (loss) between the estimated Q-value for the current state and action (denoted as $Q(s, a)$) and the target network's unbiased estimate of the expected Q-value for the next state (denoted as $Q^{expected}$). This loss represents how well the policy network's predictions align with reality, and we aim to minimize it for effective learning. The equation for calculating loss δ is the following,

$$\delta = Q(s, a) - Q^{expected}. \quad (4)$$

To address the issue of outliers in noisy Q-value estimates, we employ the Huber loss function [44]. Unlike the standard quadratic loss, the Huber loss is less sensitive to extreme

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

values, making it a robust choice for this scenario. The mathematical definition of the Huber loss is shown below,

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \mathcal{L}(\delta), \tag{5}$$

where

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{if } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

Once we calculate the loss $\mathcal{L}$, we leverage back-propagation to propagate the error signal through the policy network. This process guides the network in adjusting its internal parameters to minimize the loss in future predictions. We utilize the Adam optimizer [45] with a learning rate of LR.

❼ **Update the target network**. To further enhance the stability and performance of the learning process, we employ *soft update* for the target network [46]. This approach balances the target network's weights between those of the policy network and its own past values. The equation to update the parameters of the target network is the following,

$$\theta' = \tau\theta + (1 - \tau)\theta', \tag{6}$$

where θ' denotes the weights of the target network, θ denotes the weights of the policy network, and τ denotes the update rate of the target network.

In Figure 4, the task graph consists of four tasks. We iterate through the seven steps four times, constituting one episode for the agent. By running multiple episodes episodes (iterating through the entire process several times), the agent has the opportunity to learn more effectively from the environment and refine its policy for optimal task scheduling.

IV. Evaluation

We used the recently released dynamic task graph programming library *AsyncTask* [23] to implement the dynamic task graph scheduling system. We trained the RL model using Pytorch and compiled programs using g++11.4 with -std=c++20 and -O3 enabled to schedule the task graphs. We evaluated the runtime performance of scheduling tasks in the topological orders generated by the reinforcement learning model. We ran all the experiments on a Ubuntu 22.04.3 machine with 16 Intel i7-11700 CPU at 2.50 GHz and 125 GB RAM. All data is an average of 10 runs.

A. Baseline

We selected a heuristic approach as the baseline. This approach involves traversing a task graph and identifying a task whose dependencies have all been resolved. If multiple such tasks exist, we employ one of two pre-defined heuristics throughout the entire process. In heuristic 1, a task is selected randomly from the set of ready tasks. In heuristic 2, the task with the highest number of outgoing dependencies (fanout) is selected. In addition, we used Kahn's algorithm [38] as the third heuristic. It's important to note that we don't dynamically switch between these heuristics. Algorithm 1 details the implementation of the chosen approach.

Algorithm 1: baseline($task\_graph$)

Input: $task\_graph$: a task graph
Output: $order$: a topological order

```
order ← ∅;
array ← ∅;
/* push ready task to array */
for task ∈ task_graph do
    /* in_degree denotes the number of the fanin */
    if task.in_degree == 0 then
        array.push(task);
    end
end
while order.size < task_graph.tasks.size do
    /* Used either Heuristic 1 or 2 throughout the code */
    /* Heuristic 1 */
    task ← pop one task randomly in array;
    /* Heuristic 2 */
    task ← pop one task with most fanout in array;
    order.push(task);
    /* Resolve dependencies for fanout tasks */
    for ftask ∈ task.fanout do
        ftask.in_degree ← ftask.in_degree − 1;
        if ftask.in_degree == 0 then
            array.push(ftask);
        end
    end
end
return order
```

B. Static Timing Analysis Workload

We used the industrial static timing analysis (STA) as the workload [1], [2], which exploits task graph parallelism to parallelize graph-based analysis. STA is representative of many analysis-driven EDA applications, and is a critical step in the overall EDA flow because it verifies the expected timing behavior of a circuit design and reports the critical paths that violate the given timing constraints (e.g., set-up time, hold time). As our system schedules task graphs, we used the state-of-the-art open-source STA engine, OpenTimer [47], to generate task graphs for us. OpenTimer formulates the graph-based analysis (GBA) algorithm into a task graph. The task graph represents the corresponding circuit graph and can contain millions of tasks and dependencies for large designs. Each task computes the required timing information at its corresponding node in the circuit graph (e.g., slew, delay, arrival time), while each edge represents a dependency between two tasks. Table I lists the statistics of the 12 task graphs we used. $\|V\|$ denotes the number of the tasks in a task graph and $\|E\|$ denotes the number of the edges.

C. Training and Hyper-parameters

We trained the RL policy with six task graphs, tv80, ac97_ctrl, des_perf, usb_phy, c1355, and s1196. It's important to note that the des_perf graph used in training was

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

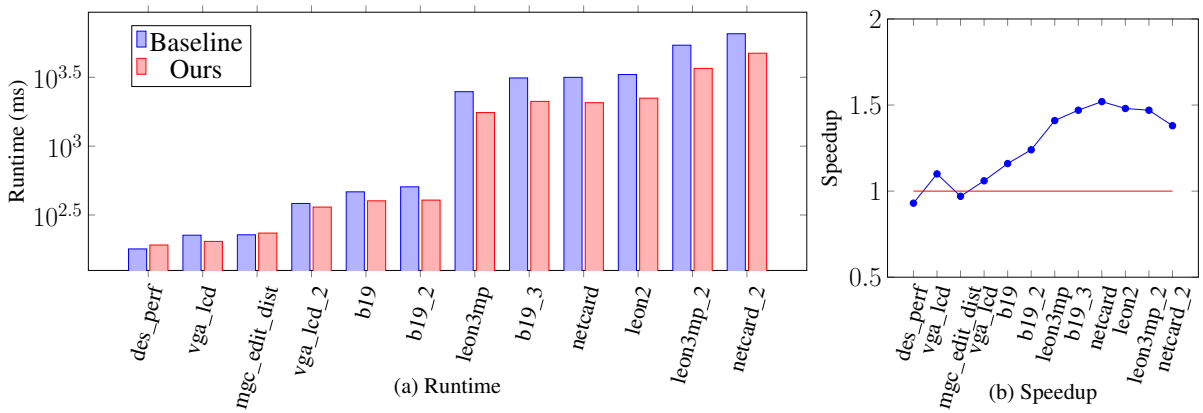

Fig. 6: Performance comparison between the baseline and our RL approach on running 12 task graphs. (a) Runtime comparison. (b) Speedup of our approach over the baseline. The red horizontal line denotes the speedup of one.

TABLE I: Task ($\|V\|$) and edge ($\|E\|$) counts of 12 task graphs.

Graphs	$\|V\|$	$\|E\|$	$\|V\| + \|E\|$
des_perf	371,587	464,810	836,397
vga_lcd	397,809	498,863	896,672
mgc_edit_dist	450,354	566,527	1,016,881
vga_lcd	679,258	823, 034	1,502,292
b19	782,914	1,048,609	1,831,523
b19_2	782,914	1,048,609	1,831,523
leon3mp	3,376,832	6,277,562	9,654,394
b19_3	3,914,570	5,243,045	9,157,615
netcard	3,999,174	7,404,006	11,403,180
leon2	4,328,255	7,984,262	12,312,517
leon3mp_2	6,753,664	8,297,576	15,051,240
netcard_2	7,998,348	9,806,794	17,805,142

a different instance from the one used in testing. The hyper-parameters we used for training are the followings:

- The update rate of the target network, τ, is 0.005.
- The value of discount factor, γ, is 0.99.
- The number of training iterations, $episodes$, is 50.
- The initial value of ϵ, ϵ^{start}, is 0.9.
- The final value of ϵ, ϵ^{end}, is 0.05.
- The decay rate of ϵ, ϵ^{decay}, is 1000.
- The number of input neurons, M, is 20.
- The number of output neurons, N, is 6.
- The number of batch size, B, is 128.
- The learning rate of Adam optimizer, LR, is $1e-4$.

D. Scheduling Task Graphs

After obtaining the topological order for a task graph using the trained RL policy and the baseline, we used *AsyncTask*'s *dependent_async* API to create the tasks in the generated topological order. This API takes two arguments. The first argument is the callable function representing each task. This function could perform various operations like parasitic calculations, slew adjustments, delays, or arrival time calculations in STA. The second argument is a list of dependent tasks.

E. Runtime Performance Comparison

Figure 6 compares the runtime performance of our approach against the baseline. We only report the best runtime performance between two heuristic methods for the baseline. We can see that the baseline exhibits faster execution only for small-sized task graphs. For example, the baseline is $1.06\times$ and $1.02\times$ faster in des_perf and mgc_edit_dist, respectively, although these differences are minor. Conversely, our RL model outperforms the baseline in all other 10 task graphs. For example, ours is $1.24\times$, $1.52\times$, and $1.48\times$ faster than the baseline in b19, netcard, and leon2, respectively. We believe this advantage stems from our RL model's ability to optimize for free system memory. Our approach adapts to changing computing environments and reduces scheduling resource consumption associated with AsyncTask's dynamic load balancing. In contrast, the baseline relies solely on the task graph structure, neglecting runtime information. The heuristic-based baseline can achieve good performance on small-sized graphs (i.e., des_perf and mgc_edit_dist). However, as graphs grow larger and more complex, with more concurrent tasks and scheduling resource consumption, the baseline's static strategy hinders adaptation, leading to performance degradation. Figure 7 visualizes the speedup achieved by our RL approach over the baseline. The speedup increases with graph size and complexity, further highlighting the superiority of our method particularly in modern industrial EDA flow that frequently requires analyzing the same task graphs multiple times.

V. Conclusion

In this paper, we have proposed a reinforcement learning model to generate topological orders for a dynamic task graph scheduling application. In the future, we will apply our approach to other parallel graph algorithms [48]–[51].

Acknowledgement

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

REFERENCES

[1] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," in *IEEE TCAD*, 2021, pp. 776–789.

[2] T.-W. Huang and M. D. F. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM ICCAD*, 2015, p. 895–902.

[3] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2023, pp. 1–12.

[4] G. Guo, T.-W. Huang, and M. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM DATE*, 2023, pp. 1–6.

[5] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism Using Control Taskflow Graph," in *ACM HPDC*, 2022, p. 283–284.

[6] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results," in *ACM/IEEE DAC*, 2022, p. 1388–1389.

[7] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[8] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[9] ——, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *ACM/IEEE DAC*, 2021, pp. 715–720.

[10] Y. Zamani and T.-W. Huang, "A High-Performance Heterogeneous Critical Path Analysis Framework," in *IEEE HPEC*, 2021, pp. 1–7.

[11] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE DAC*, 2020, pp. 1–6.

[12] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-Accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020.

[13] E. Dzaka, D.-L. Lin, and T.-W. Huang, "Parallel And-Inverter Graph Simulation Using a Task-graph Computing System," in *IEEE IPDPSw*, 2023, pp. 923–929.

[14] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," in *ACM/IEEE DAC*, 2023.

[15] T.-W. Huang and L. Hwang, "Task-Parallel Programming with Constrained Parallelism," in *IEEE HPEC*, 2022, pp. 1–7.

[16] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE HPEC*, 2022, pp. 1–2.

[17] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System," *IEEE TCAD*, vol. 41, no. 5, pp. 1448–1452, 2022.

[18] T.-W. Huang, "qTask: Task-parallel Quantum Circuit Simulation with Incrementality," in *IEEE IPDPS*, 2023, pp. 746–756.

[19] C.-X. Lin, T.-W. Huang, T. Yu, and M. D. F. Wong, "A distributed power grid analysis framework from sequential stream graph," in *GLVLSI*, ser. GLSVLSI '18, 2018, p. 183–188.

[20] C. Morchdi, C.-H. Chiu, Y. Zhou, and T.-W. Huang, "A Resource-efficient Task Scheduling System using Reinforcement Learning," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.

[21] C.-H. Chiu, Z. Xiong, Z. Guo, T.-W. Huang, and Y. Lin, "An efficient task-parallel pipeline programming framework," in *ACM International Conference on High-performance Computing in Asia-Pacific Region (HPC Asia)*, 2024.

[22] T.-W. Huang, B. Zhang, D.-L. Lin, and C.-H. Chiu, "Parallel and heterogeneous timing analysis: Partition, algorithm, and system," in *ACM International Symposium on Physical esign (ISPD)*, 2024.

[23] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms," in *IEEE/ACM ICCAD*, 2023.

[24] C. Chang, C.-H. Chiu, B. Zhang, and T.-W. Huang, "Incremental critical path generation for dynamic graphs," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[25] C.-H. Chiu and T.-W. Huang, "An experimental study of dynamic task graph parallelism for large-scale circuit analysis workloads," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[26] S. Jiang, T.-W. Huang, B. Yu, and T.-Y. Ho, "SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU," in *ACM ICPP*, 2023.

[27] D.-L. Lin and T.-W. Huang, "Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism," *IEEE TPDS*, vol. 33, no. 11, pp. 3041–3052, 2022.

[28] ——, "Efficient GPU Computation Using Task Graph Parallelism," in *Euro-Par*, 2021.

[29] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads," in *Euro-Par Workshop*, 2022.

[30] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An efficient work-stealing scheduler for task dependency graph," in *IEEE ICPADS*, 2020, pp. 64–71.

[31] "OpenMP," https://www.openmp.org/.

[32] "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," in *Journal of Parallel and Distributed Computing*, 2014, pp. 3202–3216.

[33] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," in *Computing in Science Engineering*, 2013, pp. 36–45.

[34] R. Hoque, T. Herault, G. Bosilca, and J. J. Dongarra, "Dynamic task discovery in parsec: a data-flow task-based runtime," 2017, pp. 1–8.

[35] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *PGAS*, 2014, pp. 1–11.

[36] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale," *IEEE TCAD*, vol. 40, no. 8, pp. 1687–1700, 2021.

[37] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, 2022, pp. 1303–1320.

[38] Kahn's algorithm for Topological Sorting. [Online]. Available: https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/

[39] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, and M. E. Papka, "Deep reinforcement agent for scheduling in hpc," in *IEEE IPDPS*, 2021, pp. 807–816.

[40] F. G. Blanco, E. Russo, M. Palesi, D. Patti, G. Ascia, and V. Catania, "Deep reinforcement learning based online scheduling policy for deep neural network multi-tenant multi-accelerator systems," in *Design Automation Conference (DAC)*, 2024.

[41] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *Arxiv.org/abs/1312.5602*, 2013.

[42] Rectifier (neural networks). [Online]. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

[43] Q. Lan, Y. Pan, and A. F. andMartha White, "Maxmin q-learning: Controlling the estimation bias of q-learning," in *The International Conference on Learning Representations (ICLR)*, 2020.

[44] Huber loss. [Online]. Available: https://en.wikipedia.org/wiki/Huber_loss

[45] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2014.

[46] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations*, 2016.

[47] "OpenTimer," https://github.com/OpenTimer/OpenTimer.

[48] D.-L. Lin, T.-W. Huang, J. S. Miguel, and U. Ogras, "TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2024.

[49] C.-C. Chang, B. Zhang, and T.-W. Huang, "GSAP: A GPU-Accelerated Stochastic Graph Partitioner," in *ACM ICPP*, 2024, p. 565–575.

[50] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei, "FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array," in *ACM ICPP*, 2024, p. 388–399.

[51] J. Tong, L. Chang, U. Y. Ogras, and T.-W. Huang, "BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

979-8-3503-8713-1/24/$31.00 ©2024 IEEE