A Resource-efficient Task Scheduling System using Reinforcement Learning

Invited Paper

Chedi Morchdi
Dept. of Electrical and Computer Engineering
University of Utah
Salt lake City, USA
u1402320@utah.edu

Cheng-Hsiang Chiu
Dept. of Electrical and Computer Engineering
University of Wisconsin at Madison
Madison, USA
chenghsiang.chiu@wisc.edu

Yi Zhou
Dept. of Electrical and Computer Engineering
University of Utah
Salt lake City, USA
yi.zhou@utah.edu

Tsung-Wei Huang
Dept. of Electrical and Computer Engineering
University of Wisconsin at Madison
Madison, USA
tsung-wei.huang@wisc.edu

***Abstract*—Computer-aided design (CAD) tools typically incorporate thousands or millions of functional tasks and dependencies to implement various synthesis and analysis algorithms. Efficiently scheduling these tasks in a computing environment that comprises manycore CPUs and GPUs is critically important because it governs the macro-scale performance. However, existing scheduling methods are typically hardcoded within an application that are not adaptive to the change of computing environment. To overcome this challenge, this paper will introduce a novel reinforcement learning-based scheduling algorithm that can learn to adapt the performance optimization to a given runtime (task execution environment) situation. We will present a case study on VLSI timing analysis to demonstrate the effectiveness of our learning-based scheduling algorithm. For instance, our algorithm can achieve the same performance of the baseline while using only 20% of CPU resources.**

***Index Terms*—Reinforcement Learning, Task Scheduling**

I. Introduction

Computer-aided design (CAD) tools typically incorporate thousands or millions of functional tasks and dependencies to implement various synthesis and analysis algorithms [1]–[21]. For instance, [4] describes timing analysis algorithms in a top-down *task graph* where each task represents a function and each edge represents a functional dependency. Efficiently scheduling these tasks in a computing environment that comprises manycore central processing units (CPUs) and graphics processing units (GPUs) is critically important because it governs the macro-scale performance [22]–[32]. However, existing scheduling solutions either resort to general-purpose heuristics (e.g., work stealing [33]–[36]) or a custom scheduling method (e.g., hardcoded [37]). These solutions are typically not adaptive to the change in the computing environment and often consume large scheduling resources due to the randomness involved in dynamic load balancing.

Recent advances in machine learning have inspired us to design a new scheduling framework that learns to interact with a computing environment [38]. Despite exciting progress in learning-based scheduling solutions, most of them target *independent* job batches in a high-performance computing (HPC) cluster. These solutions are not suitable for CAD problems where the goal is to find a *resource-efficient* scheduling plan for running dependent tasks using minimal CPU resources. This type of scheduling plan is very important because many CAD task graphs are much larger and more complex than conventional HPC workloads. For instance, a timing propagation task graph can compose up to 500M tasks and 700M dependencies to complete a full-timing analysis of a large design [4], [5]. Due to the sparsity, we may just use a few CPU cores to optimally complete the task graph, which in turn improves the resource utilization and the overall system throughput performance.

To this end, we introduce in this paper a resource-efficient task-scheduling system by harnessing the power of reinforcement learning. We summarize our technical contributions below:

- **Scheduling Algorithm.** We have introduced a reinforcement learning-based task scheduling algorithm to adapt the performance optimization to the computing environment. With our scheduling algorithm, applications are able to schedule tasks with few execution contexts while achieving a comparable runtime performance to existing solutions.
- **Generalizability.** We apply our RL-based scheduling algorithm to schedule a wide range of task graphs and show its superior performance. Surprisingly, our experimental results show that the RL-based scheduling policy learned

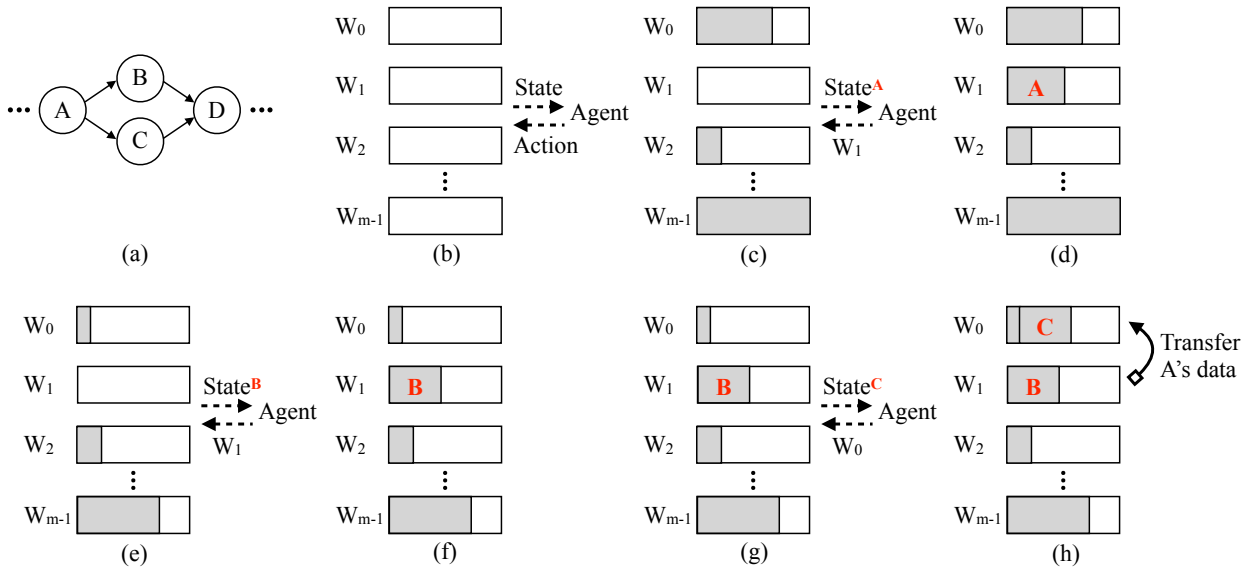


Fig. 1: Illustration of our task scheduling system. Gray rectangles denote the workloads of workers. (a) A task graph. (b) The task scheduler. (b) The scheduler asks `Agent` for task `A`'s action. `Agent` suggests W_1. (d) W_1 has `A` in its queue. (e) The scheduler asks `Agent` for task `B`'s action. `Agent` suggests W_1. (f) W_1 has `B` in its queue. (g) The scheduler asks `Agent` for task `C`'s action. `Agent` suggests W_0. (h) W_0 has `C` in its queue. `A`'s data is transferred to W_0.

from limited classes of task graphs, can generalize well to a wide range of diverse task graphs. Therefore, our RL-based scheduling algorithm provides a 'universal' scheduling solution to multi-core systems.

- **Extensible State Representations.** We have introduced an easy-to-extend state representation to accommodate new computing environment statistics, such as power consumption. With this state representation, applications are able to quickly switch to a different scheduling algorithm based on their specific needs.

We have evaluated our framework on a real static timing analysis (STA) workload that executes a task graph to complete full timing analysis. Compared to the popular heuristic-based scheduler that assigns tasks to all 40 workers uniformly at random, our RL-based scheduler achieved a slightly lower runtime on multiple task graphs using only 7-8 workers.

II. Background

A. System Overview

Our system targets at static timing analysis (STA) application, one of the most important steps in the entire EDA flow, and describes a STA workload as a task graph. The goal is to efficiently schedule the tasks of the task graph. The task graph consists of multiple nodes and edges, which represent the tasks and the dependencies among the tasks, respectively. In particular, the task dependencies not only constrain the execution order of the tasks, but also determine the data flow among them. Take the task graph shown in Figure 1(a) as an example. The task dependencies require that `A` executes before `B` and `C`, and `D` executes after `B` and `C`. Moreover, the execution of `B` and `C` needs `A`'s data, and the execution of `D` needs both `B` and `C`'s data. To schedule tasks across the execution contexts (e.g., CPUs) in a non-stationary computing environment, we propose a reinforcement learning (RL)-based task scheduler as illustrated in Figure 1(b). Next, we provide a high-level overview of this RL-based scheduler, and leave all the technical details to Section III.

We consider a multi-core system and denote the i-th worker as W_i. Each worker has its own task queue to store the tasks assigned to it. We denote the general computing environment as `State`, which includes the total workloads assigned to the workers and some information about the task to be scheduled (see Section III-B for the details). Then, based on the current `State`, the reinforcement learning `Agent` determines an `Action` that assigns the task to a certain worker. Figure 1(c)-(h) illustrate how our task scheduler schedules the three tasks `A`, `B`, and `C` from the task graph in Figure 1(a). To explain, in (c), the `Agent` first assigns `A` to worker W_1 based on the current $State^A$. In (d), `A` is inserted into W_1's queue. After W_1 completes `A`, `Agent` assigns `B` and `C` to W_1 and W_0 according to $State^B$ and $State^C$, respectively, as shown in Figure1(e) and (g). As `B` is assigned to the same worker (W_1) as `A`, there is no data transfer cost for `B` shown in Figure 1(f). In contrast, `C` requires an extra data transfer cost shown in Figure 1(h).

III. Reinforcement Learning-Based Scheduling

In this section, we reformulate the task scheduling problem as a reinforcement learning (RL) problem, and apply the Deep Q-Learning algorithm [39] to train a good RL policy for autonomous task scheduling.

A. Reinforcement learning and Markov Decision process

Reinforcement learning (RL) is a powerful machine learning framework for learning optimal decision-making in a so-called

Markov decision process (MDP) [40]–[44]. In RL, an agent interacts with a complex environment through an MDP, and the interaction data are used to further improve the agent's decision-making. Specifically, MDP is an abstract sequential decision-making process that consists of the following key elements.

- **State** s_t: At any time t, the agent observes the global state s_t of the environment, which contains all information that the agent needs to take an action.
- **Policy** π **and action** a_t: Based on the current state s_t, the agent follows its policy $\pi(\cdot|s_t)$ to take an action a_t. Here, policy π is regarded as a probability distribution over all possible actions conditioned on the state s_t.
- **State transition kernel** P: After action a_t is taken, the global state s_t transfers to a new state s_{t+1} following the environment's transition kernel $\mathsf{P}(\cdot|s_t, a_t)$, which is a distribution over all possible states conditioned on s_t, a_t.
- **Reward** r_t: The agent receives a reward signal r_t after the state transition at time t. Here, the reward r_t generally depends on s_t, a_t and s_{t+1}.

Equation (1) below illustrates the evolution of MDP. In RL, the goal of the agent is to learn the optimal policy π^*, following which yields the highest accumulated reward when interacting with the environment through the MDP.

$$\text{(MDP):}\quad s_0 \overset{\pi(\cdot|s_0)}{\longrightarrow} a_0 \overset{\mathsf{P}(\cdot|s_0,a_0)}{\longrightarrow} (s_1, r_0) \overset{\pi(\cdot|s_1)}{\longrightarrow} a_1 \cdots \tag{1}$$

B. Formulate Task scheduling as an MDP

We view task scheduling as an MDP. Specifically, consider a multi-core system with m workers. Denote T_k as the k-th task to be assigned to the workers, and denote $P(T_k)$ the set of parent tasks of T_k. Then, the elements of this MDP can be specified as follows.

State: Consider the time when the k-th task T_k is ready for scheduling, the state of the RL agent is a vector containing $2m+1$ coordinates. The first m coordinates record the current total workload of the queues of the m workers (each coordinate records the workload of one queue). The next m coordinates record the distribution of the total workload of the parent tasks $P(T_k)$ over the m workers, i.e., each coordinate records the amount of workload of $P(T_k)$ done by one worker. Finally, the last coordinate of the state vector records the total workload of the task T_k to be scheduled. We can see Figure 2 for the state vectors for task `A` and `B` from the task graph in Figure 1(a). These state information are queried from the system whenever a new task is ready for scheduling, and we take logarithm to reduce the scale of large numerical values in this state. In particular, these information is directly related to the balance of workload assigned to the workers and the data transfer cost, which are two critical factors that affect the overall system performance.

Policy and action: There are m possible actions since the task T_t will be assigned to one of the m workers. The policy is specified based on a so-called state-action value table $Q(s, a)$, which evaluates the expected return of taking action a in state s. In the next subsection, we describe the deep Q-learning algorithm used to learn $Q(s, a)$ in a data-driven way.

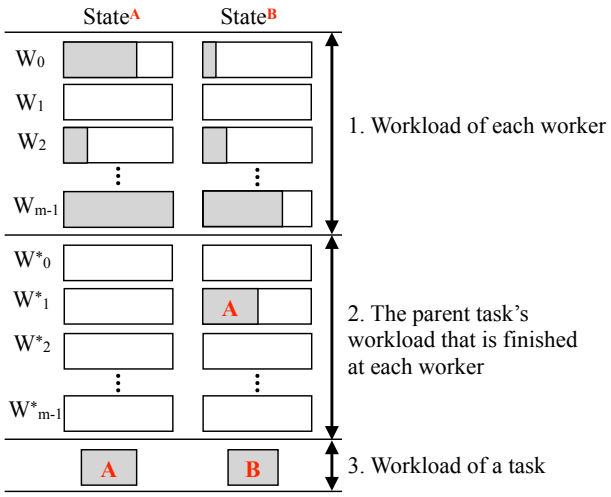


Fig. 2: Illustration of the state representations when scheduling task `A` and `B` in the task graph of Figure 1(a). The first column refers to the `State` for task `A` and the second for `B`. Gray rectangles denote the workloads. Every state includes 1) the workload of each worker, 2) the parent task's workload that is finished at each worker, and 3) the workload of a task. The first m rows of $State^A$ and $State^B$ correspond to Figure 1(c) and 1(e), respectively. As task `A` is executed by W_1, W_1^* for $State^B$ is the workload of task `A`. The workload of other W^* is empty.

State transition: Once task T_k is assigned to a worker based on the action generated by the policy, the workload of that worker's queue will change. This will further lead to a new system state. We note that the new state depends only on the previous state and the action taken, which satisfies the Markov property required by MDP.

Reward: After each state transition, the agent receives a reward signal, which is designed based on two system performance-related characteristics: the balance of total workload assigned to the workers and the data transfer cost induced by scheduling the task T_k.

- The balance of workload is defined as the gap between the maximum queue load and the minimum queue load among the workers, which quantifies the level of imbalance of the workers' queues.
- For the data transfer cost, suppose T_k is assigned to worker i, then the induced data transfer cost is the total data load of its parent tasks $P(T_k)$ that are not assigned to worker i (these parent tasks' data need to be transferred to worker i in order to execute task T_k).

Mathematically, the reward signal is defined as follows.

$$r_t := -\log_{10}(\text{workload balance}) - \alpha \log_{10}(\text{transfer cost}), \tag{2}$$

where $\alpha > 0$ is a hyper-parameter, and we take logarithm to reduce the scale of large numerical values. Intuitively, the

above reward design penalizes taking actions that would cause imbalanced queue workloads and high data transfer cost.

C. Deep Q-Learning Algorithm

Deep Q-learning is a popular algorithm that aims to learn the optimal policy that maximizes the expected accumulated reward [39]. This problem is formulated as follows.

$$\max_{\pi} \; J(\pi) := \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \big| \pi\Big],$$

where $\gamma \in (0, 1)$ is a pre-selected discount factor. In particular, given state s_t at time t, the policy generates an action a_t based on a state-action value function Q according to

$$\pi(a_t|s_t) = \arg\max_a Q(s_t, a). \tag{3}$$

Intuitively, $Q(s, a)$ evaluates the expected return of taking action a in state s, and the policy π simply suggests the action that leads to the highest Q value in a given state. It is well-known that the Q-function satisfies the following Bellman equation [41].

$$Q(s_t, a_t) = r_t + \gamma \mathbb{E}\Big[\max_a Q(s_{t+1}, a)\Big]. \tag{4}$$

The main idea of deep Q-learning is to parameterize the Q-function using a deep neural network $Q_\theta(s, a)$, where θ denotes the network parameters. This network takes state as the input and outputs the Q values associated with all possible actions, as illustrated in Figure 3. Specifically, the deep Q-learning algorithm is summarized in Algorithm 1, and it consists of the following key steps.

- **Data collection:** At each time step t, the agent takes an action a_t following an ϵ-greedy strategy, i.e., a_t is chosen uniformly at random with probability $\epsilon(t)$ (called exploration), otherwise, it is chosen based on the Q-network as $a_t = \arg\max_a Q_\theta(s_t, a)$ (called exploitation). Here, $\epsilon(t)$ is a pre-defined parameter that decays over t to encourage exploration at the beginning and exploitation later on. After taking the action a_t, we collect the transition data (s_t, a_t, r_t, s_{t+1}) and add it to the Experience Replay memory, which stores the latest N transition data and will be used in the training phase.
- **Data sampling:** In each iteration of the training phase, we query B random samples from the Experience Replay memory. Then, for each sampled transition data (assume collected at time T), we compute the following target based on the Bellman equation in Equation (4).

  $$y_T = r_T + \gamma \max_a Q_{\theta'}(s_{T+1}, a), \quad \forall\, T \in B.$$

  Here, $Q_{\theta'}$ corresponds to the so-called target Q-network, whose parameters θ' are copied from the original Q-network Q_θ every K time steps. The purpose is to decouple the original Q-network from the target and allows to properly use automatic back propagation in the model update later.
- **Model update:** With the computed targets, we build the following loss function associated with the sampled data.

  $$L = \frac{1}{B} \sum_{T \in B} \big(y_T - Q_\theta(s_T, a_T)\big)^2.$$

  Then, we update the Q-network's parameters θ using back-propagation through the computed loss L.

Algorithm 1 Deep Q-Learning Algorithm

Initialize: Q-network θ, copy to target network $\theta' \leftarrow \theta$
for *Iterations* $t = 0, 1, \ldots$ **do**
- ▶ Take action a_t following the $\epsilon(t)$-greedy policy.
- ▶ Get data (s_t, a_t, r_t, s_{t+1}) and add to replay buffer.
- ▶ Sample a batch of B samples from the replay buffer and compute the target

  $$y_T = r_T + \gamma \max_a Q_{\theta'}(s_{T+1}, a), \quad \forall\, T \in B.$$

- ▶ Update θ via back-propagation via the following loss.

  $$L = \frac{1}{B} \sum_{T \in B} \big(y_T - Q_\theta(s_T, a_T)\big)^2.$$

- ▶ if t mod $K = 0$, $\theta' \leftarrow \theta$.

end

IV. Experiments

We evaluated the performance of our reinforcement learning-based task scheduling system on an industrial static timing analysis (STA) application [1], [4] that exploits task graph parallelism to parallelize graph-based analysis (GBA). STA 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, hold). As our system schedules task graphs, we used the state-of-the-art open-source STA engine, OpenTimer [45], to generate a task graph for us. OpenTimer formulates the 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. After OpenTimer generates a task graph, our scheduler directly performs the scheduling on the task graph. Table I lists the statistics of the nine circuits we used.

We compiled programs using gcc-12 with `-std=c++17` and `-O3` enabled. We ran all the experiments on a Ubuntu 19.10 (Eoan Ermine) machine with 80 Intel Xeon Gold 6138 CPU at 2.00GHz and 256 GB RAM.

A. Baseline and Deep Q-Learning

For comparison, we implemented a baseline method based on the random action (RA) engine, which assigns each task to one of the 40 workers uniformly at random. Such a baseline method is widely used to schedule STA workloads.

It randomly assigns the tasks to the workers without adapting to the dynamic and non-stationary computing environment.

To learn our RL-based task scheduler, we implemented Algorithm 1 to train the RL policy using a mixed graph composed of the following three graphs: aes_core, tv80 and c6288. Specifically, we implemented Algorithm 1 with the following set of hyper-parameters: batch size $B = 64$, target network synchronization period $K = 10$, reward discount factor $\gamma = 0.95$, reward weight $\alpha = 0.01$, and experience replay memory size $N = 10$k. In particular, for the $\epsilon(t)$-greedy policy, we adopt the initialization $\epsilon(0) = 1.0$ (at $t = 0$) and multiply $\epsilon(t)$ by a factor of $\epsilon\_decay = 0.99998$ after every iteration. Also, we used a four-layer fully-connected neural network to parameterize the Q-function [46], and the network architecture is illustrated in Figure 3. To update the model parameters θ via back-propagation, we use the standard Adam optimizer with learning rate $\eta = 1e-4$ [47].

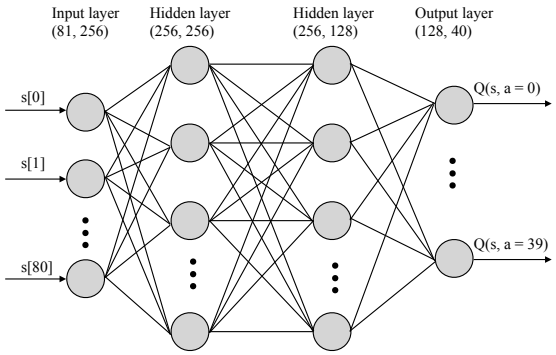


Fig. 3: Illustration of the Q-network architecture. The network takes in the state vector as input (input dimension=81), then propagates it forward through 2 hidden layers and finally outputs the Q-values corresponding to each of the 40 possible actions (output dimension=40). The task at hand is then scheduled to the worker corresponding to the highest Q-value.

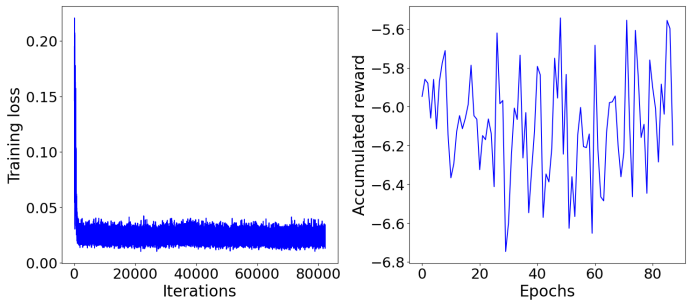


Fig. 4: Left: Training loss v.s. iterations in the training. Right: Accumulated reward v.s. epochs in the training. Every epoch consists of 1K iterations, and the accumulated reward for each epoch is calculated by $R = \sum_{t=1}^{1000} \gamma^t r_t$.

Figure 4 plots the training loss (left figure) and the accumulated reward (right figure) achieved by the RL policy during the training process. From the left figure, it can be seen that the training loss decays quickly, indicating that the learned policy performs well on the training data. Moreover, the right figure shows that the RL policy eventually achieved an accumulated reward at around -6.0. Next, we further test the trained RL policy on some unseen test graphs and demonstrate its superior generalization performance.

B. Performance Comparison

We tested and compared the runtime performance of the baseline RA scheduler and our RL-based scheduler on various graphs with different configurations, and the results are summarized in Table I. We note that the mixed graph used in the test phase is composed of the same three types of graphs (aes_core, tv80 and c6288) as those used in the training phase, but with different configurations. Hence, the mixed graph used in the test phase is very different from the one used in the training phase.

From Table I, it can be seen that the total runtime of our RL-based scheduler is consistently slightly lower than that of the RA scheduler for all the test graphs. Surprisingly, these runtime results are achieved by our RL-based scheduler using only 7-8 workers, which are much more efficient compared to the RA scheduler that utilizes all of the 40 workers. Thus, the experimental results clearly demonstrate the advantage of our RL-based scheduler, indicating its superior memory efficiency and energy efficiency. This also implies that, through the reinforcement learning framework and our specialized reward design, the RL-based scheduler successfully learned a policy that enhances workload balance among the workers and reduces unnecessary data transfer cost.

In Figure 5, we further plot the distribution of tasks assigned to each worker under the RA scheduler and our RL-based scheduler for two task graphs (aes_core and mixed graphs). Specifically, one can observe from Figure 5(a) that, in order to schedule the tasks of the aes_core graph, the RA scheduler assigns tasks uniformly to all the 40 workers. As a comparison, our RL-based scheduler assigns tasks to only 8 workers, and moreover, most of the tasks are actually assigned to only 4 workers. One can make very similar observations in the Figure 5(b) for the mixed graph.

V. Conclusion

We have introduced a resource-efficient reinforcement learning-based task scheduling system to adapt the performance optimization to the computing environment. We have evaluated our task scheduling system on an industrial static timing analysis workload. Compared to the popular heuristic-based scheduler, our RL-based scheduler achieved a lower runtime on all task graphs while using only 20% of workers. Our future work plans to extend our framework to a distributed environment [48]–[53] and consider GPU task graphs [54]–[56] into our state model.

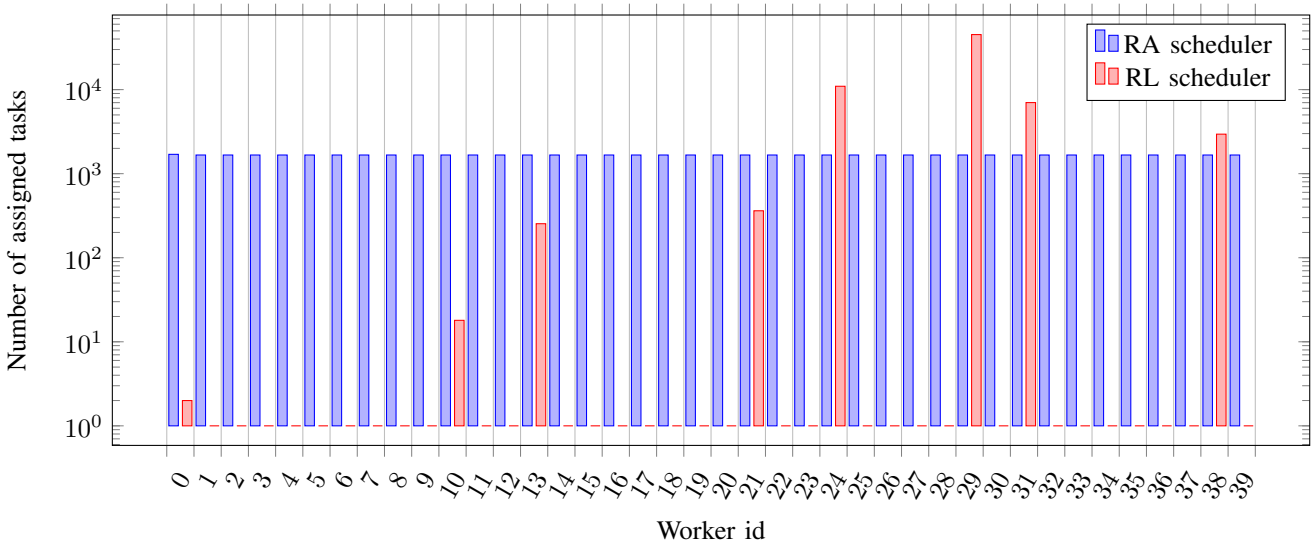
Acknowledgment

We are grateful for the support of National Science Foundation (NSF) grants, CCF-2349141, CCF-2349582 (CAREER), CCF-2106216, DMS-2134223, ECCS-2237830 (CAREER), OAC-2349143, and TI-2229304.

TABLE I: Runtime comparison between the random action (RA) scheduler and our reinforcement learning (RL) scheduler. $\|V\|$ and $\|E\|$ respectively denote the number of nodes and edges of a graph. Improvement denotes the performance of RL over RA.

Graph	$\|V\|$	$\|E\|$	Runtime (Seconds)			# Workers		
			RA	RL	Improvement	RA	RL	Improvement
mixed graph	88,626	115,777	38.44	**38.29**	0.39%	40	**7**	471%
aes_core	66,751	86,446	29.55	**28.89**	2.28%	40	**8**	400%
ac97_ctrl	42,438	53,558	18.92	**18.09**	4.59%	40	**8**	400%
tv_80	17,038	23,087	7.76	**7.04**	10.22%	40	**8**	400%
wb_dma	13,125	16,593	5.52	**5.33**	3.56%	40	**8**	400%
c6288	4,837	6,244	2.01	**1.98**	1.52%	40	**8**	400%
c7552_slack	3,802	4,791	1.75	**1.60**	9.38%	40	**7**	471%
usb_phy_ispd	2,447	2,999	1.11	**1.00**	11%	40	**7**	471%
s1494	2,292	2,925	1.04	**0.97**	7.22%	40	**7**	471%

(a) aes_core graph



(b) Mixed graph

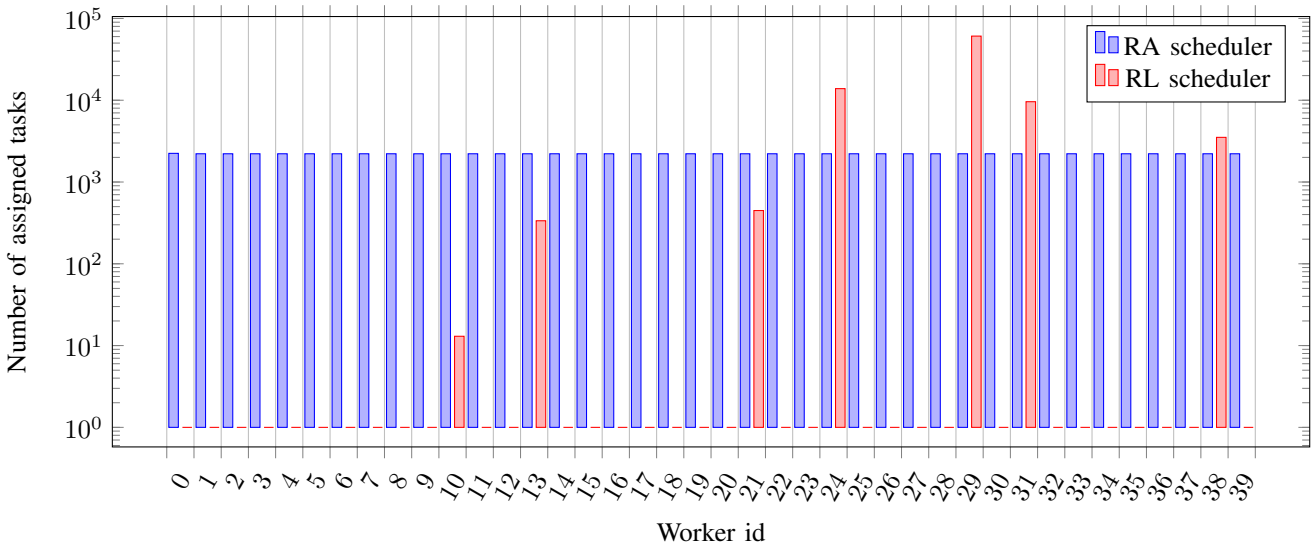


Fig. 5: Histogram of assigned tasks per worker for two graphs (aes_core and mixed graphs).

REFERENCES

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

[2] ——, "Accelerated path-based timing analysis with mapreduce," in *ACM ISPD*, 2015, p. 103–110.

[3] ——, "On fast timing closure: speeding up incremental path-based timing analysis with mapreduce," in *ACM/IEEE SLIP*, 2015, pp. 1–6.

[4] 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.

[5] 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.

[6] 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.

[7] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," 2019, pp. 974–983.

[8] T.-W. Huang, "A General-Purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM ICCAD*, 2020.

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

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

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

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

[13] D.-L. Lin and T.-W. Huang, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," in *IEEE HPEC*, 2020, pp. 1–7.

[14] ——, "Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism," *IEEE TPDS*, vol. 33, no. 11, pp. 3041–3052, 2022.

[15] 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.

[16] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Fast Path-Based Timing Analysis for CPPR," in *IEEE/ACM ICCAD*, 2014, p. 596–599.

[17] ——, "UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm," in *IEEE/ACM ICCAD*, 2014, p. 758–765.

[18] T.-W. Huang and M. D. F. Wong, "UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, 2016.

[19] C.-C. Chang and T.-W. Huang, "uSAP: An Ultra-Fast Stochastic Graph Partitioner," in *IEEE HPEC*, 2023, pp. 1–7.

[20] S. Jiang, T.-W. Huangand, and T.-Y. Ho, "GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction," in *IEEE HPEC*, 2023, pp. 1–7.

[21] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A General Cache Framework for Efficient Generation of Timing Critical Paths," in *ACM/IEEE DAC*, 2019.

[22] 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.

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

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

[25] ——, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," *IEEE TCAD*, 2023.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Essential Building Blocks for Creating an Open-Source EDA Project," in *ACM/IEEE DAC*, 2019.

[33] 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.

[34] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM MM*, 2019, p. 2284–2287.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[41] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[42] P. Dayan and C. Watkins, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[43] V. R. Konda and V. S. Borkar, "Actor-critic–type learning algorithms for markov decision processes," *SIAM Journal on Control and Optimization*, vol. 38, no. 1, pp. 94–123, 1999.

[44] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. International Conference on Neural Information Processing Systems*, 1999, p. 1057–1063.

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

[46] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[47] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA, 2015.

[48] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "DtCraft: A distributed execution engine for compute-intensive applications," in *IEEE/ACM ICCAD*, 2017, pp. 757–765.

[49] ——, "DtCraft: A High-Performance Distributed Execution Engine at Scale," *IEEE ICAD*, vol. 38, no. 6, pp. 1070–1083, 2019.

[50] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "A General-Purpose Distributed Programming System Using Data-Parallel Streams," in *ACM MM*, 2018, p. 1360–1363.

[51] T.-W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *ACM/IEEE DAC*, 2016, pp. 1–6.

[52] 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.

[53] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "Distributed Timing Analysis at Scale," in *ACM/IEEE DAC*, 2019.

[54] 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.

[55] D.-L. Lin and T.-W. Huang, "Efficient GPU Computation Using Task Graph Parallelism," in *Euro-Par*, 2021.

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