Boosting the Performance of Reinforcement Learning-based Task Scheduling using Offline Inference

Chedi Morchdi
Department of Computer Science and Engineering
Texas A&M University
chedi.morchdi@tamu.edu

Cheng-Hsiang Chiu
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
chenghsiang.chiu@wisc.edu

Yi Zhou
Department of Computer Science and Engineering
Texas A&M University
yi.zhou@tamu.edu

Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
tsung-wei.huang@wisc.edu

***Abstract*—Modern computer-aided design (CAD) tools leverage complex algorithms incorporating millions of interdependent functional tasks. Scheduling these tasks efficiently across CPUs and GPUs is paramount, as it directly governs overall performance. However, existing scheduling approaches are typically hardcoded within applications, limiting their adaptability to non-stationary computing environments. To address this challenge, a recent paper introduced a novel reinforcement learning-based online inference task scheduling algorithm. While this approach can learn to adapt the performance optimization in dynamic environments, it integrates online task execution and online task inference, leading to significant overheads, such as querying the system status for each task. To address the concern, we propose a reinforcement learning-based offline inference task scheduling system. Our system separates task execution from inference, performing the inference offline to avoid the overheads. We will evaluate our approach on a VLSI static timing analysis workload and demonstrate that our approach is consistently faster than the online inference method, albeit with slightly increased resource consumption.**

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

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]. For example, [2] details timing analysis algorithms structured as top-down *task graphs*, where each task represents a function and each edge represents a functional dependency. Efficiently scheduling these tasks across a computing environment composed of multicore central processing units (CPUs) and graphics processing units (GPUs) is crucial, as it directly impacts overall performance. However, existing scheduling approaches fall short in adaptability to the computing environments. They either rely on general-purpose heuristics like work stealing [3] or custom methods like hardcoding [4]. These solutions often struggle to adapt to changes in the computing environment and can consume significant scheduling resources due to the randomness inherent in dynamic load balancing.

TABLE I: Runtime comparison between the online inference [5] and our offline inference task scheduling.

Approach	Online task inference	Task execution
[5]	5724.7 seconds	0.298 second
Ours	0 second	0.146 second

A recent study [5] addressed this challenge by proposing a resource-efficient task scheduling system that leverages reinforcement learning (RL) for adaptability to the change in the computing environment. Their system comprises two entities: workers and an RL agent. The RL agent dynamically assigns one of the workers to execute a task based on real-time system data, such as queue loads of workers. This approach is known as *online* inference task scheduling, as task assignment (inference) happens during task execution. However, while the online inference approach in [5] requires only 20% resources for scheduling all tasks, we identified two concerns. First, the online per-task inference incurs significant overheads due to frequent system information queries. Second, inter-process communication (IPC) [6] interleaves between workers (implemented in C++) and the RL agent (implemented in Python) can negatively impact overall runtime performance.

To address these limitations, we propose a reinforcement learning-based *offline* inference task scheduling system. This approach separates task execution from task inference. Specifically, we perform the task inference offline, generating a predetermined task assignment for online task execution. This eliminates the overhead associated with per-task inference during online task execution, leading to a significant performance boost for the RL-based scheduling system. Table I compares the runtime performance of our offline inference approach

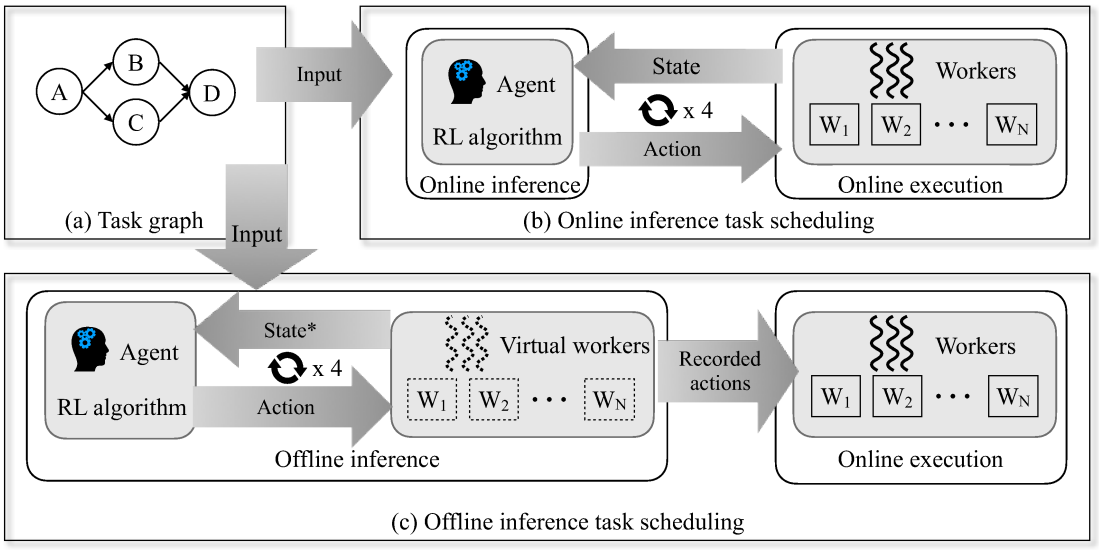

Fig. 1: Illustration of the online inference task scheduling system [5] and the proposed offline inference task scheduling system. (a) An example task graph of four tasks and four edges. (b) The online inference task scheduling [5] consists of the online task inference and the online task execution. (c) The offline inference task scheduling consists of offline task inference and the online task execution.

with the online inference approach from [5]. While the online approach performs both task inference and task execution online, our solution only requires online task execution, leveraging the pre-determined task assignment generated offline. We summarize our technical contributions below:

- **Scheduling Algorithm**. We have introduce a reinforcement learning-based offline inference task scheduling algorithm to boost performance compared to its online counterpart. By performing task inference offline, our scheduling algorithm eliminates the overhead of per-task inference during online task execution. This leads to a significant overall runtime improvement.
- **Parameterized Offline Environment**. We have developed a parameterized offline environment to simulate online interactions between workers and the reinforcement learning agent. This allows applications to easily fine-tune parameters, mimicking the behavior of various reinforcement learning-based online inference task scheduling systems.
- **Performance Studies**. We have conducted experiments to evaluate our offline inference approach against the online method on a real static timing analysis (STA) workload. Our new approach consistently achieves a significantly lower runtime while requiring slightly more CPU resources.

II. Reinforcement Learning-based Online Inference Task Scheduling

In this section, we give an overview of the reinforcement learning-based online inference task scheduling system presented in the paper [5].

A. System Overview

The RL-based online inference task scheduling system [5] targets at static timing analysis (STA) application – one of the most important steps in the entire CAD flow, which describes a STA workload as a task graph and schedules 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. Each task performs a STA operation. The task dependencies not only constrain the execution order of the tasks, but also determine the data flow among them. For example, consider the task graph in Figure 1(a). Task `A` must execute before tasks `B` and `C`, while task `D` depends on the completion of both `B` and `C`. Additionally, tasks `B` and `C` require data from task `A`, and task `D` requires data from both `B` and `C`.

To schedule tasks across the execution contexts (e.g., CPUs or workers) in a non-stationary computing environment, the paper [5] presents a RL-based online inference task scheduling system, as shown in Figure 1(b), and functions as follows:

- **Workers Report State**: Workers `Workers` send the current state information `State` to the RL agent `Agent`. `State` encodes the real-time system data, such as the queue loads of workers.
- **RL Agent Makes Decision**: The RL agent `Agent` analyzes the received state information `State`. Based on this data, it determines the optimal action `Action`.
- **Task Assignment**: `Agent` communicates its decision `Action` back to the workers `Workers`, essentially assigning a specific worker to execute a particular task.
- **Iterative Process**: This process iterates for each task in the task graph. In the example of Figure 1(a), with four tasks, the communication between workers `Workers`

and the RL agent `Agent` would repeat four times.

B. Problem Formulation and Training

To solve the task scheduling problem, the paper [5] formulates the task scheduling problem as a RL problem through a so-called *Markov Decision Process* (MDP) [7]–[11], and defines the four key elements of the MDP as follows:

- **State**. The state represents the current system status, including details like the queue load of each worker, and is queried by the workers whenever a new task is ready for scheduling. This information is crucial for balancing workload assigned to the workers and minimizing data transfer costs, both of which significantly impact the overall system performance. In [5], the state is a vector with $2 * N + 1$ coordinates. The first N coordinates represent the queue loads of all N workers. The next N coordinates represent the workload of the new task's parents that is completed at each worker. The final coordinate represents the workload of the new task itself.
- **Action**. The action denotes which worker will execute the new task.
- **State transition**. Once the new task is assigned to a worker based on the action, the queue load of that worker will change. This will further lead to a new system state.
- **Reward**. After each state transition, the RL agent receives a reward signal based on system performance-related characteristics. In the paper [5], the reward consists of the queue load and the data transfer cost.

After formulating the scheduling problem as a RL problem through MDP, the paper [5] applies the Deep Q-learning algorithm to train a good RL policy that maximizes the accumulated reward over time.

C. Limitations of Online Inference Scheduling

While the paper [5] achieves a comparable runtime performance using only 20% of the computing resources, it has two key limitations. First, the per-task online inference, where the RL agent determines worker assignment for each individual task, incurs significant overheads. This is because the agent needs the current system status (encoded as a state) to make a decision for every task. Querying this state information for every task incurs significant overhead. When the task graph gets bigger, more overheads are incurred.

Second, the inter-process communication between the RL agent (implemented in Python) and the workers (implemented in C++) introduces additional overhead. Since they are separate processes, each task inference requires two communications (one for state and one for action), as shown in Figure 1(b). This overhead becomes more significant for larger task graphs.

In summary, the online inference task scheduling system that interleaves task inference and execution between two separate processes (RL agent and workers) can lead to performance degradation due to factors like CPU migrations. Moreover, the frequent inter-process communication between the RL agent and workers adds to the overall runtime cost. To address these limitations, we propose an offline inference task scheduling system. This approach separates task inference from task execution. We perform the task inference offline, generating a pre-determined task assignment for task execution. This eliminates the need for per-task online inference and the associated overhead, leading to improved performance.

III. Reinforcement Learning-based Offline Inference Task Scheduling

The RL-based online inference task scheduling in [5] integrates task inference with task execution (as shown in Figure 1(b)), leading to significant overhead. To address this, we propose separating task inference from execution, moving the task inference offline. This creates a two-stage process, offline task inference and online task execution. The offline task inference stage focuses solely on task inference. It analyzes the system state and generates an optimal task assignment. The online task execution stage executes the tasks based on the pre-determined task assignment generated offline. This separation eliminates the need for per-task online inference and its associated overhead, resulting in improved performance. Figure 1(c) illustrates this two-stage approach.

A. The Offline Task Inference Stage

At the offline inference stage, we perform the task inference. Unlike the online inference [5] that only determines the worker ID for each task, our offline inference for a task determines ❶ the execution order for that task and ❷ the worker ID that is going to execute that task. We need to additionally determine the execution order because of the following reason. The execution order is supposed to be determined by the workers dynamically as they know when a task is finished and when a new task is available to be scheduled, as the online inference task scheduling system does. However, at the offline task inference stage we are not executing the tasks across the workers yet and thus the workers can not determine the execution order for us. That is the reason why we need to additionally decide the execution order of the tasks.

❶ To determine the execution order of tasks, we perform a topological sorting algorithm on the task graph. This order is crucial because of task dependency constraints. For example, in Figure 1(a), task `A` must precede tasks `B` and `C`, and task `D` can only start after both `B` and `C` finish. Therefore, a valid execution order is either `A-B-C-D` or `A-C-B-D`, which corresponds to the topological order of the task graph. There are two most popular topological sorting algorithms: Breadth-First Search (BFS)-based and Depth-First Search (DFS)-based. The BFS-based algorithm prioritizes tasks at the same depth level, exploring them all before moving on to deeper levels. Conversely, the DFS-based algorithm explores tasks as deep as possible along a chosen branch before backtracking and exploring another branch. While the BFS- and DFS-based algorithms generate same order for Figure 1(a), we can use a more complex task graph (like Figure 2) to illustrate the detailed differences between BFS- and DFS-based topological sorting.

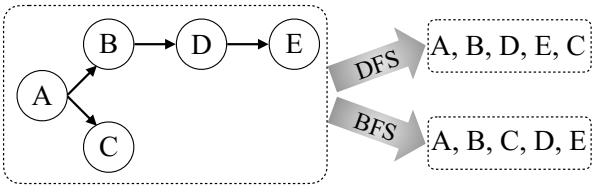

Fig. 2: The topological orders of a task graph using DFS-based and BFS-based topological sorting algorithm.

❷ To determine which worker executes which task (i.e., perform task inference), we leverage the trained reinforcement learning (RL) policy from the existing approach [5]. Here's how it works:

- **Task Order Determination**: We first establish the execution order for all tasks using a topological sorting algorithm (as explained previously).
- **Synthetic State Generation**: For each task in the order, we create a synthetic state `State`* to approximate the actual system state `State` that would be available online, as shown in Figure 1(c). This synthetic state `State`* is a vector with $2 * N + 1$ coordinates, similar to the online approach. The first N coordinates represent the queue loads of N `Virtual workers`. We use virtual workers `Virtual workers` to represent the workers `Workers` because we don't directly interact with `Workers` as we are offline. The next `N` coordinates respectively record the amount of workload of the task's parent tasks completed by the `N` virtual workers. The last coordinate represents the workload of the current task itself.
- **Worker Assignment and State Update**: We feed the synthetic state `State`* to the trained RL policy. The policy then recommends an action `Action`, which essentially assigns the task to a specific virtual worker. This worker assignment is stored for later online execution. To simulate task execution progress, we update the synthetic state `State`*. We add the current task's workload to the assigned virtual worker's queue load in `State`*. We also decay the queue load of all virtual workers by dividing their respective coordinates in `State`* by their average, mimicking the progress of actual workers.

B. The Online Execution Stage

At the online execution stage, the workers are responsible for executing the tasks based on the recorded actions forwarded from the offline inference stage. The recorded actions dictate three key aspects for each task: 1) task ID, 2) the execution order of the task, and 3) the worker ID that will execute the task. We read in and store the recorded actions in a data structure. Next, we directly query the data structure without doing any task inference and assign the tasks in the recorded order to the responsible workers.

TABLE II: The statistics of the six task graphs we used. $\|V\|$ and $\|E\|$ respectively denote the number of nodes and edges of a task graph.

Graph	$\|V\|$	$\|E\|$	$\|V\| + \|E\|$
synthetic	88,626	115,777	204,403
aes_core	66,751	86,446	153,197
ac97_ctrl	42,438	53,558	95,996
tv80	17,038	23,087	40,125
wb_dma	13,125	16,593	29,718
c6288	4,837	6,244	11,081

IV. Experimental Results

We compared the runtime performance of our reinforcement learning-based offline inference task scheduling system and its online inference counterpart [5]. 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. VLSI Timing Analysis Workloads

We evaluated the performance on an industrial VLSI static timing analysis (STA) application [2], [12], that leverages task graph parallelism for parallelizing graph-based analysis (GBA) workloads. STA plays a critical role in the overall CAD 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 and hold time). We chose the state-of-art open-source STA engine, OpenTimer [13], as the experimental tool. OpenTimer formulates the GBA algorithm into a task graph. The task graph represents the corresponding circuit design and can contain millions of tasks and dependencies for large designs. Each task computes the required timing information at its corresponding node in the task graph (e.g., slew, delay, arrival time), while each edge represents a dependency between two tasks.

Table II lists the statistics of the six task graphs we used. Five of them are transformed from the circuit descriptions into task graphs and are dumped by OpenTimer. We synthesized one more task graph called `synthetic` by concatenating three task graphs `aes_core`, `tv80`, and `c6288`. Using a synthetic graph improves the model's performance, generalizability, and robustness by providing a more diverse training set for better generalization.

B. Training and Testing

We trained the RL policy on the graph `synthetic` following the Deep Q-learning algorithm presented in the paper [5]. We then tested the trained policy on the six test graphs. We note that the graph `synthetic` 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 using different instances. Hence, the graph `synthetic` used in the test phase is very different from the one used in the training phase.

C. Performance Comparison

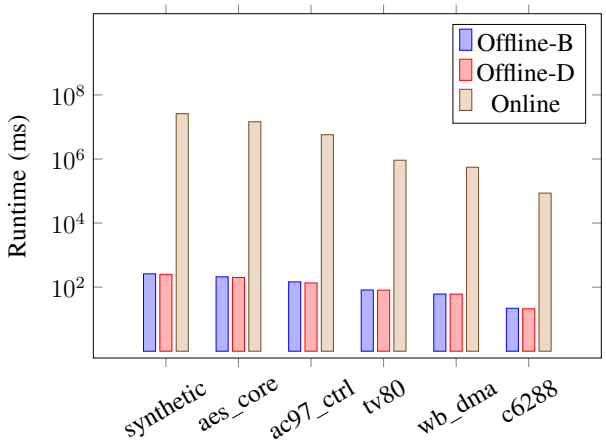

Fig. 3: Elapsed time comparison between offline inference scheduling with BFS order (denoted as Offline-B), offline inference scheduling with DFS order (denoted as Offline-D), and online inference scheduling (denoted as Online) on the six graphs, respectively.

Figure 3 shows the elapsed time comparison. We can see that the offline approach (with both BFS and DFS order) is consistently better than the online approach for all the test graphs. For example, for the graph `aes_core`, the offline approach with DFS order finishes the analysis in 0.2 second while the online approach needs 4 hours. The substantial time difference comes from the following reasons. First, the online inference performs the per-task inference on the fly, which incurs significant overhead. Second, our offline approach separates task inference from task execution and utilized a predetermined task assignment, eliminating the need for online inference decisions for each task. These factors contribute to the clear performance advantage of our offline inference task scheduling system over the online inference counterpart, as demonstrated by the experimental results.

In Figure 4, we report the total number of used workers and the number of the top 95% workers used by each approach. Here, we define the number of top 95% workers as the number of workers that handle the top 95% (vast majority) of the total number of tasks, where the workers are ranked in a descending order based on the total number of tasks assigned to them. Our findings reveal that the offline approach uses a higher total number of workers compared to the online approach. This occurs because our offline approach interacts with virtual workers during offline inference, making it less sensitive to real-time changes in the computing environment and less adaptive than the online approach. However, both online and offline approaches utilize a similar number of top 95% workers. This indicates that, regardless of the inference strategy (online or offline), most tasks are assigned to a relatively small group of workers. This highlights the resource efficiency of both RL-based approaches.

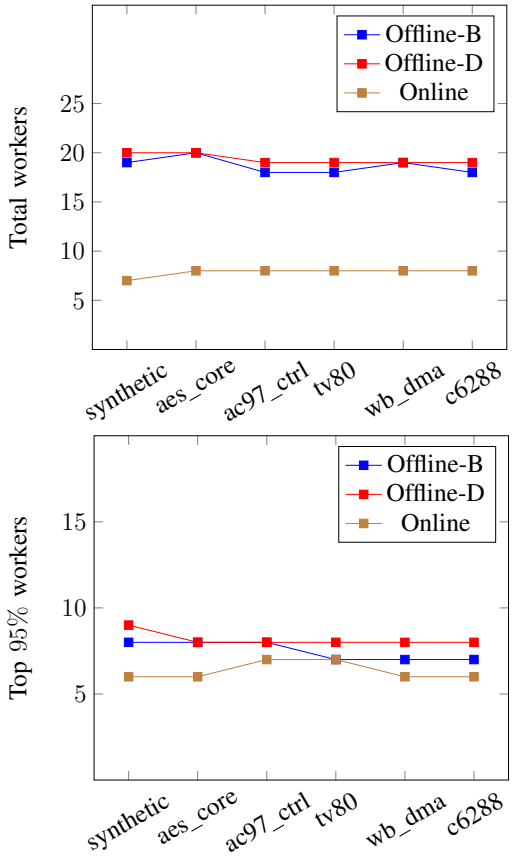

Fig. 4: Histogram of the number of utilized workers and the top 95% workers between offline inference scheduling with BFS order (denoted as Offline-B), offline inference scheduling with DFS order (denoted as Offline-D), and online inference scheduling (denoted as Online) on the six graphs, respectively. The number of top 95% workers denotes the number of workers that handle the top 95% of the total number of tasks, where the workers are ranked in a descending order based on the total number of tasks assigned to them.

D. Runtime Breakdown

To further demonstrate the advantages of our offline approach over the online approach, we show the runtime breakdown of the online approach in Table III. As the table shows, online task inference consumes significantly more time than online task execution for all six graphs. For instance, task inference takes over four hours, while task execution requires only 0.15 second when running `aes_core`. This breakdown in Table III clearly highlights the substantial overhead associated with online inference in the online inference task scheduling system. By eliminating this online inference cost incurred with each test run, our offline inference approach offers a significant runtime advantage.

E. Benefits of Offline Over Online Inference

In this section, we conclude two main benefits of our offline inference task scheduling over the online inference

TABLE III: Runtime breakdown of the online inference scheduling approach measured in seconds.

Graph	Online task inference	Online task execution
synthetic	25961.84	0.1453
aes_core	14527	0.15
ac97_ctrl	5724.7	0.298
tv80	915.8	0.117
wb_dma	552.557	0.093
c6288	86.1	0.029

task scheduling [5]. First, our offline approach demonstrates significant performance improvements over [5], particularly for running large-scale industrial workloads like static timing analysis (STA). As shown in Figure 3, the offline approach finishes all STA workloads in under one second, while the online approach requires considerably more time. This advantage becomes even more pronounced in real-world scenarios where workloads are typically run multiple times. The offline approach's faster execution translates to a significant time saving for these repetitive tasks.

Second, our inference approach and the online inference approach are comparable in terms of the task execution time. From the runtime breakdown in Table III and the elapsed time comparison in Figure 3, we find out that the our method and the online approach run comparably in terms of task execution time. For example, when running `aes_core`, the offline approach needs 0.19 second and the online approach needs 0.15 second. Take `ac97_ctrl` graph as another example. The offline approach finishes in 0.146 second and the online approach finishes in 0.298 second. Our offline approach runs faster in three out of six graphs. Different from the online inference approach that interacts directly with the workers, our offline approach is less adaptive to the change of the computing environment than the online approach. It is difficult for our method to outperform the online approach in all graphs in view of the task execution time. Although the online approach is a little bit faster in some graphs, the difference is minimal. That means our offline inference approach is comparable to the online inference approach in terms of scheduling the tasks across the underlying execution units.

Overall, our offline inference task scheduling offers significant advantages for industrial workloads due to its faster overall runtime. While task execution time remains comparable, the reduced task inference overhead translates to a substantial performance improvement. This makes our approach a compelling choice for real-world STA applications and similar scenarios.

V. Conclusion

We have introduced a reinforcement learning-based offline inference task scheduling system to boost the performance of the online inference task scheduling system. We have evaluated our task scheduling system on an industrial static timing analysis workload. Compared to the existing RL-based online inference scheduling, our new RL-based offline inference scheduling achieves better runtime performance on all tested task graphs while using slightly more computing resources.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] "Inter-process communication," https://en.wikipedia.org/wiki/Inter-process_communication.

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

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

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

[10] 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.

[11] 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.

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

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