Scheduling to Ensure Performance and Cost Effectiveness in Power-Modulated Datacenters

A
Dissertation
Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

In partial fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Vanamala Venkataswamy

February 2023

APPROVAL SHEET

This Dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Author: Vanamala Venkataswamy

This Dissertation has been read and approved by the examining committee:

Advisor: Dr. Andrew Grimshaw

Committee Member: Dr. John A. Stankovic

Committee Member: Dr. Yanjun Qi

Committee Member: Dr. Haifeng Xu

Committee Member: Dr. Donald E. Brown

Accepted for the School of Engineering and Applied Science:

[illegible]

Jennifer L. West, School of Engineering and Applied Science

February 2023

Abstract

Datacenters are the critical infrastructure in today's information age. The sustained demand for digital services has led to record datacenter build-outs and increased energy consumption. Modern datacenters heavily rely on brown energy. Two significant problems with using brown energy are 1) brown energy is expensive and 2) harmful to the environment since brown energy generation releases greenhouse gases. Renewables are becoming increasingly accessible energy sources to power the datacenters, leading to dramatically lower energy costs and significant climate impact reductions. Green datacenters, also referred to as power-modulated datacenters, can utilize multiple energy sources (wind and solar) by intelligently adapting computing to energy generation. The difficulty with renewables is that power generation is intermittent and subject to frequent fluctuations, making job scheduling in such datacenters interesting from a research perspective. Green datacenters need intelligent systems and system software that adapt to the intermittent power supply from renewables.

Traditional heuristics-based job schedulers use hand-crafted scheduling policies. Hand-engineering domain-specific heuristics-based schedulers to meet specific objective functions in highly dynamic green datacenters is time-consuming, error-prone, expensive, and requires domain expertise. Reinforcement Learning (RL) has solved sequential decision making tasks of impressive difficulty by maximizing reward functions through trial and error. The growing body of research has shown that Reinforcement Learning schedulers can learn effective job scheduling policies in traditional datacenter environments with a constant power supply. Although the results demonstrated in the existing work are convincing, they do not examine the complexities presented in the dynamic green datacenter environments.

This dissertation delivers four fundamental contributions. First, we developed a unified green datacenter simulator driven by heuristic and RL scheduling policies and synthetic or real workloads and integrated multiple renewable energy sources to power the datacenter. The simulator allows resource scaling (small to medium scale), allowing the practitioners to experiment with datacenters of different capacities. Second, we systematically explore RL scheduler design features demonstrating the performance implications when adequately designed. Third, while many existing RL schedulers optimize for single objective effectively, they do not address multi-criteria optimization. Moreover, one or more of these objectives may be in opposition, e.g., maximizing the total value (revenue) while minimizing the overall job delay. We demonstrate that constrained RL schedulers learn to accomplish such opposing goals and satisfy multi-criteria optimization. Finally, classic online RL job schedulers can learn efficient scheduling strategies but often takes hundreds of thousands of timesteps to explore the environment and adapt from a randomly initialized DNN policy. Offline reinforcement learning, also known as batch RL, presents the prospect of policy optimization from large pre-recorded datasets without online environment interaction. Additionally, we show that incorporating prior datasets to pre-train the RL scheduler agent can short-circuit the random exploration phase and continuously improve with online data collection.

To deliver these contributions, we employed Offline, Online, and Constrained-Controlled RL methods. We evaluated the efficacy of these methods with diverse power supply and load conditions using synthetic and real workloads. This study provides several insights to design future RL schedulers that ensure performance and cost-effectiveness in power-modulated datacenters.

To my family Kiran, Ananya and Nimit.

In memory of my grandmother Achamma Konda Reddy.

Acknowledgements

My husband, Kiran Reddy, has always been an invaluable source of support and assistance. He encouraged me to return to school when he sensed that I could never feel fulfilled without achieving this goal. I am deeply grateful for his willingness always to pick up the slack, whether in caring for our children or managing the home, whenever I was unavailable.

There are more reasons than I can mention that I am grateful to my advisor, Dr. Andrew Grimshaw. I came to the University of Virginia as a project engineer on the XSEDE project. After working for five years, Andrew fully supported my decision when I expressed my interest in returning to graduate school. I am incredibly grateful, especially the last year after Andrew joined Lancium full-time; his continued support ensured that I made progress, guiding and nudging me toward the finish line. His advice has always been available when I needed it, whether on the small details about academic life or important matters about career and life.

I am ever grateful to Dr. Anita Jones for her generosity when we shared an office and for inspiring me to pursue my dream of finishing graduate school.

I thank my daughter, Ananya, for being a great source of joy throughout our ten years in Charlottesville. She has been a wonderful child. The fun times I spent with her - whether dropping her off at daycare, going to her soccer games on weekends, or watching family movies - have often kept me going. I also thank my son, Nimit, who always brings a smile to my face. From a newborn when I started graduate school to a kind-hearted six-year-old, his bright personality, thoughtfulness, and sense of humor make any problem fade into the background.

In addition to their financial contributions, my father, Venkataswamy, and my mother, Dhanalakshmi, have supported me in countless ways. Especially my mother, who was of tremendous help in taking care of my newborn son when I started graduate school. My brother, Madhu Chandra, always cheered me on in all my achievements, big or small. To my grandmother, I express sincere gratitude for instilling in me the value of kindness and equality. I will forever cherish her classical Indian stories, which always ended with valuable life lessons that helped shape me. I appreciate the constant support from my in-laws and extended family, especially my uncle Shivashankara Reddy, throughout this journey.

I thank my committee members, Professors John A. Stankovic, Yanjun Qi, Donald E.

Brown, and Haifeng Xu, for their insights and suggestions about my research. Also, I thank the faculty of the Computer Science Department, notably Professors Kevin Skadron, Jack Davidson, and Alf Weaver, for their support.

Many students and colleagues in the department have helped me work through my ideas and allowed me to bend their ears. In particular, I thank Swaroopa Dola, Anindya Prodhan, In Kee Kim, Ankur Sarker, Elaheh Soltanaghaei, Elahe Sadredini, Erin Griffiths, Muhammad Nur Yanhaona, Chris Koeritz, Michele Co for their time and patience. I am grateful to many collaborators, especially Dr. Allen Sill, Sandeep Nimmagadda, and Elham Hojati at Texas Tech University. I learned a great deal about operating green datacenters from this group. I appreciate the collaboration with Jake Grigsby, Dr. Qi's student, in the past year. I am truly fortunate to have received support and recommendation letters from Mohanram N, Dr. Ann Chervenak, Dr. Aiichiro Nakano, and Dr. Baishakhi Ray over the years. Thank you all for the encouragement and support.

I am thankful to Kim Gregg, Jai Maupin, Marion Hight, Jayne Weber, Deborah Rose, and Christine Palazzolo for their invaluable assistance with the bureaucratic aspects of graduate life. I thank the system administration staff Paul Henderson, Philip Apoian, Jed Keplinger, and Essex Scales for always providing timely systems support.

I offer special thanks to my friends simply for making life better. To Daphne Ingene and Jamie DeCoster, for badminton games, Pokemon walks and engaging discussions on current affairs. To Gladness Msumanje, for providing me assurance and peace of mind when I dropped off my kids with her at daycare. My friends, Prabu Kumar, Mehta, and Siddesh, and my best allies, Deepika and Anshu, for always cheering me on and lifting my spirits during the pandemic and numerous other occasions.

Finally, I thank those that funded my graduate education, including Michael McNamara at Lancium Inc and the University of Virginia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The sustained demand for digital services has led to record datacenter build-outs and increased energy consumption. Conservative estimates suggest that global datacenter energy consumption between 2010 and 2018 went up by 6%, totaling 205 TWh in 2018. Further research [1] implies that the datacenter energy consumption is an order of magnitude higher than the estimated 6%, considering numerous unaccounted small-to-medium scale datacenters and datacenters that cater to new technologies (e.g., blockchain, cryptocurrency mining). Datacenters in the U.S. consume 1.8% of the total electricity; electricity predominantly generated using non-renewable sources emitting an estimated $\sim$ 230 Million Metric tons of greenhouse gases every year.

Given high carbon emissions and growing societal awareness of climate change, government agencies, non-profits, and the general public demand cleaner (greener) goods and services. Consequently, cloud service providers are investing in green datacenters[1], i.e., datacenters partially or entirely powered by renewable energy. While some cloud service providers [2] [3] [4] buy carbon offsets, others [5] [6] are shifting towards datacenters entirely powered by renewables. These datacenters either generate their own renewable energy (self-generation) or draw from an existing carbon-free (e.g., wind, solar) power generation plant (co-location). Electricity generation from wind farms in the U.S. increased from $\sim$ 6 billion

[1]The terms green datacenter and power-modulated datacenter are used interchangeably throughout this dissertation.

kilowatt-hours (kWh) to 275 billion kWh between 2000 and 2018 [7].

Furthermore, electricity cost is one of the significant contributors to the total operational cost in datacenters [8] [9]. Cloud service providers need to factor energy costs into their final billing to make a profit. Higher datacenter operational costs translate to the end-users paying high prices to run their jobs. One way to reduce the costs is to reduce the operational expense in the datacenters. Since electricity is a significant piece of that equation [9] [10], using renewable energy sources (RES) to power datacenters is becoming a necessity and a reality. A handful of startups [5] [3] [11] are working in this direction, with national and international investors investing heavily in such datacenters. Similarly, corporations that previously relied on traditional cloud service providers for their computational needs are prospecting for green cloud service providers to reduce costs. Wind energy is one of the lowest-priced and cleanest energy sources available today [12]. By 2050, wind energy could avoid the emission of 12.3 gigatonnes of greenhouse gases [13].

The difficulty with renewables is that power generation is intermittent and subject to frequent fluctuations, making co-location and self-generation interesting from a research perspective. Solar energy generation has a diurnal pattern with maximum energy generation at midday, while wind energy generation is higher at night. By combining solar and wind sources, energy generation typically complements each other.

Traditional heuristics-based job schedulers [14] [15] [16] use hand-crafted scheduling policies suitable for datacenters with constant power supply. Hand-engineering domain-specific heuristics-based schedulers to meet specific objective functions of highly dynamic green datacenters is time-consuming, error-prone, expensive, and requires domain expertise.

Reinforcement Learning (RL) has solved sequential decision tasks of impressive difficulty by maximizing reward functions through trial and error. Recent examples using deep learning range from robotic locomotion [17] [18], sophisticated video games [19] [20], and congestion control [21]. The growing body of research [22] [23] [24] have shown deep reinforcement learning (DRL) schedulers can learn effective job scheduling policies in traditional datacenter environments with constant power supply.

Although the results demonstrated in the existing work are convincing, they do not examine the complexities presented in the dynamic green datacenter environments. The existing work propounds the RL schedulers as black boxes without exploring the design choices that may potentially improve their performance.

Additionally, while many existing RL schedulers optimize for single objective effectively, they do not address multi-criteria optimization where the goal is to optimize for more than one objective. Furthermore, one or more of these objectives may be in opposition, e.g., maximizing the total value (revenue) while minimizing the overall job delay. A trade-off exists between achieving high job value on the one hand and low expected delays on the other. Hence, the aims of achieving high rewards and low costs are in opposition. In practice, datacenter operators prioritize multiple objectives, including high system utilization and job completion. RL schedulers need to learn to accomplish such opposing goals to satisfy multi-criteria optimization.

Finally, classic online RL job schedulers can learn efficient scheduling strategies but often takes hundreds of thousands of timesteps to explore the environment and adapt from a randomly initialized DNN policy. Existing RL schedulers overlook the importance of learning and improving upon heuristic policies. For instance, designing reward functions that elicit desired behaviors in complex environments is challenging. Instead, the RL schedulers can leverage the behavior of custom heuristic schedulers' designed specifically for unique workloads or environments to learn and improve overall performance. Offline reinforcement learning, also known as batch RL, presents the prospect of policy optimization from large pre-recorded datasets without online environment interaction. Additionally, we can incorporate prior datasets to pre-train the RL scheduler agent short-circuiting the random exploration phase to learn a reasonable policy with online training. In essence, an effective RL scheduler framework for pre-training from off-policy datasets that continuously improves with online data collection can provide best-of-both-worlds.

1.1 Thesis Statement and Research Contributions

In this dissertation, **we hypothesize that Reinforcement Learning based schedulers perform better than heuristics schedulers for power-aware scheduling in datacenters. RL schedulers adapt to varying conditions and learn strategies that meet the specific objectives set forth by the datacenter operators**. These objectives may include single-criteria optimization, constrained optimization with opposing goals, or multi-criteria optimization. To evaluate this hypothesis, this dissertation 1) implemented a green datacenter simulator driven by various workloads, resource configurations, and operating conditions, including intermittent power supply from renewables; 2) uncovered several previously unexplored RL scheduler design features and tuning parameters that may lead to better-performing systems by investigating the limitations of existing RL schedulers, 3) optimizing for multiple objectives, and 4) utilizing historical datasets collected from expert demonstrations to reduce training time and demonstrate performance improvement over expert systems.

1.1.1 Power-modulated datacenter simulator

Due to many practical reasons, such as the cost of resources, time scale, presence of other loads on the clusters, or lack of access to the facilities, experimental evaluation cannot be adequately performed on real systems. To obtain reliable results, simulations must be repeated with different setups using the same controllable conditions that simulate different real-life scenarios. The RL agent learns policies by repeated trial and error. Training the RL scheduler on a real system requires dedicated access to resources to not interfere with other users' jobs. Furthermore, resource ramping based on intermittent power supply may cause significant physical wear and damage to the machines. While many ad-hoc simulators exist for clusters, they did not sufficiently meet the green datacenter requirements. For example, these simulators did not feature the capability to control resource availability based on the power supply. Additionally, we need a lightweight simulator suitable for RL scheduler training and evaluation. The existing RL schedulers use custom-built datacenter environments for specialized purposes. None of the existing environments simulate a green datacenter with resource ramping, a crucial feature for our research.

In Chapter-4, we present a unified green datacenter simulator driven by synthetic or real workloads and integrate multiple renewable energy sources and Energy Storage Devices (ESDs) to power the datacenter. The simulator allows resource scaling (small to medium scale), allowing the practitioners to experiment with datacenters of different capacities. The simulator supports the resource pool expanding and contracting in response to the intermittent power supply from renewables. Additionally, the simulator supports configuring various time horizons, job arrival rates, job size distribution, and job durations for synthetic workloads. Finally, we designed a green datacenter simulator controlled by either RL and heuristic scheduling policies (e.g., Quality of Service (QoS), Shortest Job First (SJF), Highest Value First (HVF), and First Come First Serve (FCFS)).

1.1.2 Designing RL scheduler for power-modulated datacenters

We surveyed existing research and determined that the current research does not adequately address challenges presented in the complex dynamic green datacenter environments. The current research presents the RL schedulers as black boxes without exploring the system design configurations. We identified four RL scheduler design features pertinent to green datacenters, namely 1) state and action space representation, 2) configuring for different workloads, 3) extended planning horizon, and 4) policy network configurations.

In Chapter-5, we present an RL scheduler framework called **R**enewable Energy **A**ware **Re**source management (**RARE**) and experimentally demonstrate the performance improvements when the RL scheduler is appropriately designed and configured. We show that our RL scheduler performs better than heuristics policies in the dynamic green datacenter environment for synthetic and HPC workloads for a small to medium-scale cluster with 10 to 1200 resources. The RL scheduler adapts exceptionally well to the intermittent power supply (synthetic and actual power prediction data). With synthetic workload, our RL scheduler performs 18% to 25% better for small-scale clusters and 2% to 20% better for medium-scale clusters than heuristic policies. With the HPC workload, the RL scheduler performs 7% to 14% better than the heuristic policies. With varying power supply (100%, 90% and 80% power), our RL scheduler performs 9% to 13% better in small scale cluster and 5% to 20% better compared to heuristic policies in medium scale cluster. We show that as the planning horizon extends (from 36 to 72-time units), our RL scheduler performs 4% to 14% and 6% to 10% better than heuristic policy for synthetic and HPC workloads, respectively.

1.1.3 Constraint controlled reinforcement learning scheduler

In green datacenters, intermittent power supply from renewables leads to intermittent resource availability, inducing job delays and associated costs. The scheduler's objective is to schedule jobs on a set of resources to maximize the total value (revenue) while minimizing the overall costs due to job delays. In addition, datacenter operators often prioritize multiple objectives, including job completion and system utilization.

In Chapter-6, we present a **C**onstraint **C**ontrolled **RL** (**CoCoRL**) scheduler that automatically learns conflicting reward and cost functions. We accomplish this by applying the Proportional-Integral-Derivative (PID) Lagrangian methods in Deep Reinforcement Learning to the job scheduling problem to achieve favorable learning dynamics. We demonstrate our scheduler's performance for both the primary objective (maximizing total job value) and the secondary objective (minimizing costs due to job delays). We demonstrated that CoCoRL simultaneously achieves a higher total job value, high system utilization, and a high job completion ratio while keeping the costs considerably lower compared to heuristic policies. For synthetic workload, our scheduler provides a significantly higher total job value ratio between 5%-25% for job arrival rates ranging from 20-60% (fewer jobs in the system). At a higher job arrival rate of 60-80% (more jobs in the system), our scheduler performs 5%-10% better than baseline heuristic policies. The CoCoRL scheduler performs comparably to the QoS policy and completes 5%-20% more jobs than the other heuristic policies. Our scheduler shows a 2%-6% higher system utilization than heuristic policies between 80% job arrival rate. For HPC workloads, our scheduler achieves similar superior performance while staying within the cost_limit, whereas the heuristic policies accrue 5x-

10x higher negative penalties. Finally, we demonstrate the significance of accurately tuning hyperparameters (e.g., cost limit) to satisfy various optimization goals set by datacenter operators.

1.1.4 Learning to schedule using offline and online RL methods

We investigate how learning from previously collected demonstrations can be applied in job scheduling using data-driven RL techniques. We explored two data-driven RL methods, namely 1) Behaviour Cloning (BC) and 2) Offline RL (historically known as batch RL), which aim to learn policies from logged data without further interaction with the environment. These methods address the challenges concerning the cost of data collection and safety, particularly pertinent to real-world applications of RL.

In Chapter-7, we show that the performance of BC methods is highly dependent on the quality of the training dataset. BC is likely to fail to learn good policy when the dataset does not contain enough transitions generated by a well-performing policy or the fraction of poor data is too large. Unlike BC, the performance of Offline RL is resilient to training datasets with mixed (both well-performing and poor) heuristic policies. When the dataset does not contain enough transitions generated by a well-performing policy or the fraction of poor data is too large, Offline RL methods can leverage the benefits of stitching parts of suboptimal trajectories. The challenge with Offline RL is that because the learning algorithm must entirely rely on the static dataset, there is no possibility of improving by exploration. If the dataset does not include transitions that reach high-reward regions of the state space, it may be impossible to learn such high-reward regions. Additionally, not all environments have historical or quality datasets needed in offline learning methods.

In Online Learning, the agent interacts with the environment and explores numerous state-action pairs to learn a generalizable policy. Model-free deep RL methods are notoriously expensive in terms of their sample complexity. Even relatively simple tasks can require millions of steps of data collection. What is considered an upside, *exploration process*, is also the downside because the exploration process is time-consuming. We utilize Offline RL as a **launchpad** to learn effective scheduling policies from prior experience collected using expert demonstrations or heuristic policies. Finally, we demonstrated that by effectively incorporating prior datasets to pre-train the agent, we short-circuit the random exploration phase to learn a reasonable policy with online training.

1.1.5 Summary

We present a unified green datacenter simulator driven by synthetic or real workloads and integrate multiple renewable energy sources and Energy Storage Devices (ESDs) to power the datacenter. We apply the proportional-integral-derivative (PID) Lagrangian methods in RL to accomplish constrained optimization where the opposing goals of maximizing total job value and minimizing job delays and multi-criteria optimization satisfy multiple objectives simultaneously. We address challenges concerning the cost of data collection and explore two data-driven RL methods which aim to learn policies from logged data without active interaction with the datacenter environment. We utilize Offline RL as a launchpad to learn effective scheduling policies from prior experience collected using Oracles or heuristic policies. Such a framework is effective for pre-training from off-policy datasets and well suited to continuous improvement with online data collection. This set of works demonstrates that Reinforcement Learning based schedulers perform better than heuristics schedulers for power-aware scheduling in green datacenters while effectively adapting to a complex dynamic environment, optimizing for different objectives, including constrained optimization or multi-criteria optimization and incorporating prior datasets to pre-train the RL agent to short-circuit the random exploration phase, which confirms our hypothesis.

1.2 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter-2: Job Scheduling - Background and Related Work discusses the scheduling problem and prior heuristic approaches, challenges, and limitations of existing work.

Chapter-3: Reinforcement Learning and Scheduling discusses RL basics and how we map the scheduling problem in green dataceters into an RL environment to learn effective scheduling under various conditions.

Chapter-4: Power-Modulated Datacenter Simulator presents a unified green datacenter simulator driven by synthetic or real workloads and integrates multiple renewable energy sources, and supports various operating conditions.

Chapter-5: Designing RL Scheduler for Power-Modulated Datacenters presents the limitations of current work, challenges presented in the complex dynamic green datacenter environments and explores RL scheduler design parameters and configurations for overall performance improvement.

Chapter-6: Constraint Controlled Reinforcement Learning Scheduler presents a constrained optimization problem in green datacenters and demonstrates the performance gains by learning rewards and penalties using PID Lagrangian methods.

Chapter-7: Learning to Schedule using Offline and Online RL Methods explores Offline and Online RL methods and demonstrates circumstances under which each method performs effectively. We will evaluate the various learning variations individually and combine some techniques.

Chapter-8: Conclusions summarizes the dissertation and discusses the implications of this work and potential future research directions.

Chapter 2

Job Scheduling - Background and Related Work

This chapter will discuss the Job Scheduling problem and existing work. We will introduce commonly used objective functions in datacenter scheduling. Then, we will briefly discuss green datacenters and why we chose Job Value as the RL scheduler's objective function and evaluation metric in the green datacenter.

2.1 Job Scheduling

Job Scheduling is deciding when and where to run a set of **jobs** on a set of **resources** to optimize an objective function. Information about available resources and jobs defines a scheduling problem. We need to know the resource type and the number of resources to determine when the jobs can feasibly be finished. By specifying the resources, we effectively define the boundary of the scheduling problem. Additionally, we describe each job in terms of resource requirements, duration, the earliest time to start, and the time it might take to complete. The job duration is generally uncertain, but we usually suppress that uncertainty (allowing buffer time) when stating the problem.

Objective function defines the objective of the optimization. Preferably, an objective function should consist of all costs that depend on scheduling decisions. However, costs are often difficult to identify entirely or measure. Typical objective functions for schedulers include: minimizing total execution time, minimizing cost to the user, minimizing the makespan, maximizing throughput, or maximizing revenue for the cloud provider. One of the most commonly used objective functions in datacenters is resource utilization.

Scheduling problems require a performance measure for a given set of jobs in a schedule. A solution to a scheduling problem amounts to answering two questions:

- Which resources should be allocated to perform each job? - **Space**
- When should each job be performed? - **Time**

If a given set of jobs available for scheduling does not change over time, the system is **static**; in contrast, when new jobs appear over time, the system is **dynamic**. The system is **deterministic** when conditions are assumed to be known with certainty. On the other hand, when we recognize uncertainty, the model is **stochastic**. Traditionally, static and deterministic models have proved more tractable than dynamic models and have been studied extensively. This research focuses on job scheduling in a dynamic and stochastic environment. Dynamic because the jobs arrive in an online manner and stochastic because resource availability is not constant and depends on the intermittent power supply from the renewables.

Schedulers are the resource management software that decides which jobs to run, when, and where to run them. Efficiently scheduling users' jobs on distributed computing resources with heterogeneous resources and job mix may need complex policies to meet agreed objectives. Traditionally, dynamic, online schedulers have used heuristics-based scheduling techniques where system engineers design algorithms to capture multiple and diverse problems. Reasoning about these heuristics' interactions is complicated and becomes intractable as the number of variables and heuristics increases.

The search for an exceptional job scheduler has existed for several decades [25]. Besides Computer Science, the optimal scheduling problem exists in many other fields, from industrial automation to NASA space shuttle payload processing [26]. The research is rich with ideas for schedulers that optimize for various objective functions. The optimal scheduling problem is **NP-hard** [27], and the proliferation of schedulers indicates that no one scheduler is suitable for arbitrary jobs to resource combinations. Ultimately, an enterprise's objective function drives the design of any bespoke job-shop scheduler.

2.2 Scheduler's Objective Function

An **Objective Function** maps an event or values of one or more variables onto a real number, intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize/maximize an objective function. An objective function is either a loss function or its negative (in specific domains, variously called a reward function, a profit function, a utility function, or a fitness function), in which case it is to be maximized.

The primary classifications for measuring the scheduling quality are 1) Application-centric scheduling and 2) Resource-centric scheduling. Application-centric scheduling aims at optimizing the objectives of the individual application. Resource-centric scheduling aims at optimizing the resource utilization of resources provider.

2.2.1 Power-modulated datacenters

Here we briefly discuss the power-modulated (a.k.a green datacenters) and specific challenges in green datacenters, e.g., power supply variations from renewables and challenges with scheduling jobs. Finally, we discuss reasons for selecting Job Value as a preferred objective function for our green datacenter model. A detailed explanation of the power-modulated datacenter model is discussed in Chapter-3.

The resource pool, R, ramps up and down based on the power available to the datacenter at any given time. Power availability decides when and how many resources are turned on or off. Therefore, power prediction data is part of the state space. As power availability changes, the corresponding resource availability is reflected in the state representation supplied to the scheduler agent.

2.2.2 Job Value as the objective function in power-modulated datacenters

In green datacenters, prioritizing the jobs that provide the highest value is essential. Given intermittent power supply (and machine availability), the resource management software must pick jobs most likely to generate higher value (and minimize SLO violations). Furthermore, the objective of picking jobs with the highest value serves as a proxy for utilization since the highest value jobs are packed on the machines first, and then the next tier of jobs (jobs with the lower value) are picked, and so on. This also ensures QoS for users willing to pay more by prioritizing their jobs over other low-priority jobs.

2.3 Related Work

Production planning and scheduling problems frequently arise in practice and have long been the focus of Operations Research, Control Theory, and Production Management. Although the modeling and learning approaches proposed in this dissertation are general and can be deployed for different applications, distributed job scheduling problems depict the target application domain in the context of which we test, analyze, and validate all approaches.

In the rest of this section, we will categorize every scheduler we discuss based on the taxonomy described in [25]. The taxonomy for classifying distributed schedulers includes local vs. global, static vs. dynamic, distributed vs. central, cooperative vs. non-cooperative, and optimal vs. suboptimal. Some schedulers may not strictly fit in just one of the sub-sections, but we included them in the corresponding sub-section because of the use case described in the literature.

2.3.1 Distributed and multiprocessor schedulers

Distributed scheduling has been a well-studied problem for many decades. The graph-matching approach described in [28] uses a minimax criterion representing the maximum time for a task to complete module execution and communication in all the processors.

Optimal task assignment is defined using graphs. The graphs are then used to represent the module relationship of a given task and the processor structure of a distributed computing system. The search for weak homomorphism corresponding to optimal task assignment is formulated as a state-space search problem and solved by the well-known A* algorithm [29].

The work in [30] describes greedy algorithms for communicating tasks with static task assignments in distributed computing systems. The goal is to minimize the total execution and communication costs incurred by an assignment. The model considers interference costs which reflect the degree of incompatibility between two tasks. Another solution that uses task clusters and distributed groups of processes that communicate heavily is described [31]. Task clusters are tasks with heavy inter-task communication that should be on the same host. Distributed groups also have inter-task communication but execute faster when spread across separate hosts. This work proposes a bidding strategy and uses system and task description messages.

In [32], the author proposes two scheduling methods. The first is adaptive with dynamic reassignment based on broadcast messages and stochastic learning automata. This method uses a system of rewards and penalties as a feedback mechanism to tune the policy. The second method uses bidding and one-time assignment in a real-time environment. The scheduler based on the bayesian decision described in [33] is a global, dynamic, distributed, cooperative, suboptimal, heuristic, and one-time assignment.

2.3.2 Batch and meta schedulers

Extensive research exists on resource management and scheduling on clusters - Torque [34], Moab [35], and Maui [36] that have had tremendous success in managing HPC clusters. These systems work on a single cluster under one administrative domain. They are not well suited to support a federation of geographically distributed clusters primarily because of non-shareable file systems. Many of these distributed schedulers were also created to support parallel programming application models and run coarse-grained workloads. These cluster schedulers allow clients to specify the types of processing environments, but unfortunately, not across multiple clusters or administrative domains.

Condor [37] [38] is a global, dynamic, distributed, cooperative, suboptimal scheduler that utilizes idle CPU cycles in workstations. The system has a central coordinator for keeping track of idle machines and placing queued jobs on the idle machines. The system also has a local scheduler on each participating workstation that constantly monitors local activity. When the local workstation is idle, it schedules a remote waiting job. When a local user returns to the workstation, the remote job is preempted, checkpointed, and moved to another idle machine to continue execution. The philosophy of Condor is to leave the owner in complete control of the workstation, no matter the cost of doing so. This system does not support the concept of QoS or job priorities.

Maui [36] is a global, dynamic, distributed, suboptimal (heuristics) based batch scheduler extensively used in the HPC community. Maui scheduler implements a backfill scheduler with job prioritization and emphasizes a fair share policy. Maui was initially designed to maximize cluster utilization but later evolved to maximize scheduler performance and flexible policy specifications. The concept of Job class (Job queue) is used to constrain the types, sizes, and resources that jobs can specify. The concept is carried forward in the other batch schedulers discussed next. Maui also supports QoS and Access Control Lists (ACLs) to determine the associated access privileges. Maui limits the users and resource consumption using throttling policies, e.g., limiting at most three active jobs per user at any given time. The newer systems achieve this using Cgroups, VMs, or containerization.

Moab [39] is a commercial cluster scheduler that supports moldable job requests in which the user provides several job size and wall time options. The scheduler will choose an option based on whichever option can be met first.

Simple Linux utility Resource Management (Slurm) [40] is a global, dynamic, distributed, suboptimal (heuristics) based batch scheduler widely used in HPC facilities. Slurm is an open-source native scheduler that operates on a single administrative domain (single cluster). Slurm has a plugin that enables systems like Maui or Moab to integrate with broader systems. Slurm allows exclusive and non-exclusive access to resources. Users can specify a job's resource requirements, submit a job, monitor job status, and terminate jobs. It arbitrates conflicting resource requests by putting jobs in the wait queue until sufficient resources become available. Slurm does not support work-preserving job preemption and migration.

Maui and Moab are meta-schedulers (manage one or more clusters within one administrative domain) in that they can integrate with Native-schedulers (schedulers that manage a single cluster) like Torque or Slurm.

2.3.3 Schedulers for grids and federated clouds

This subsection describes meta-schedulers or resource management systems that integrate multiple administrative domains to provide a single-system image to the end users. Along with scheduling jobs, the systems may provide additional features like security, accounting, and visualization data management tools, all packaged as one software bundle or toolkit. An extensive survey of other systems in this category is presented in [41].

Condor-G [42] combines software from Globus and Condor [38] to allow users to harness multi-domain resources that provide a single system image. The Condor-G system leverages significant security, resource discovery, and resource access in multi-domain environments supported by Globus Toolkit [43] and management of computation and harnessing resources within a single administrative domain supported by the Condor system. The combination of the inter-domain resource management protocols of the Globus Toolkit and Condor's intra-domain resource management methods allows the user to harness multi-domain resources as if they all belong to a single domain. The user defines the tasks to be executed then Condor-G handles all aspects of discovering and acquiring appropriate resources, regardless of their location; initiating, monitoring, and managing execution on those resources; detecting and responding to failure; and notifying the user of termination.

PBS Pro [44] is a resource management system for grid computing that includes security, computing, and data management features. Compute Grids built using PBS pro can support advance reservations, harvest idle desktop computer cycles, and peer schedule work (i.e., moving jobs across the room or across the globe). Data management in PBS Pro is handled via automatic stage-in and stage-out of files. The PBS Pro monitors daemon processes (called MOMs) to collect real-time data on the state of systems and executing jobs. This data, combined with information on queued jobs, accounting logs, and static configuration information, gives a complete view of the managed resources. These include advance reservation support, cycle harvesting, and peer scheduling. Job preemption is implemented, but job checkpointing and migration are not part of the scheduling system.

Legion [45] is a meta-computing system that acts as an Operating system with support for computing, storage, and other services. The scheduler supported by Legion is global, dynamic, distributed (hierarchical), and suboptimal (heuristics). The governing philosophy of scheduling in Legion is a negotiation of services between autonomous entities: the consumer of the service (application) and the service provider (system or resource). A similar approach is described in AppLeS [46]. The service provider has complete control of the system at all times and decides when and what resources are shared in this environment. The scheduler described in [45] is a basic heuristic scheduler. However, the more important feature is extending the resource management service and implementing a more sophisticated scheduler for specific application domains.

Globus [47] system enables modular deployment of grid systems by providing the required basic services and capabilities as a toolkit. The toolkit comprises components that implement basic services, such as security, resource location, resource management, data management, resource reservation, and communications. Globus is constructed as a layered architecture in which higher-level services can be developed using the lower-level core services. Its emphasis is on the hierarchical integration of Grid components and their services. The underlying scheduler can be Legion or AppLeS, which integrates with the Globus toolkit to support resource management.

GenesisII [48] is an open-source, standards-based (Open Grid Forum and Open Grid Services Architecture) grid middleware that supports remote computing and secure data sharing. GenesisII provides researchers with simple, easy-to-use, secure access to resources, particularly data and computes resources, regardless of location. The central feature of GenesisII and the GFFS is a shared, global, distributed path-based namespace where everything (compute resources, queues, files, directories, exports) is represented as files. The computer resources scattered under multiple administrative domains are linked in one or more queues providing location transparency to end users. Users submit their jobs (along with resource requirements) to the queue, and the jobs are scheduled to resources (could be under multiple administrative domains) matching the job requirements.

2.3.4 Schedulers for datacenter analytics workload

Datacenter analytics workload includes running periodic jobs that take large amounts of data (collected for a period of time, e.g., google search) and produce meaningful statistics on that data.

Mesos [49] is a global, dynamic, distributed, suboptimal resource management system. Mesos's main feature is enabling sharing (multi-tenancy) commodity clusters between multiple diverse cluster computing frameworks (Hadoop and MPI). Multi-tenancy aims to improve cluster utilization and avoid per-framework data replication. Mesos shares resources, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To support the sophisticated schedulers of today's frameworks, Mesos introduces a distributed two-level scheduling mechanism called resource offers. Mesos decides the resource count to offer each framework, while frameworks decide which resources to accept and which jobs to run on them.

YARN [50] is a global, dynamic, centralized, suboptimal resource management system. It was mainly developed to run MapReduce jobs and jobs with other frameworks (e.g., spark). YARN implements ClusterScheduler at its core, but other schedulers with varying heuristic scheduling policies can be implemented. YARN addresses multi-tenancy with Hadoop on Demand (HoD), where users submit their job with a description of an appropriately sized compute cluster to Torque. Torque enqueues the job until enough nodes become available. When nodes become available, Torque starts the leader process on the head node, which would then interact with Torque (or Maui) to start HoD's slave processes. The slave processes then spawn a JobTracker and TaskTrackers for that user, then accepting a sequence of jobs. Some of the work discussed below implement custom schedulers on top of YARN.

Morpheus [51] is a global, dynamic, centralized, suboptimal scheduling system for datacenter analytics workload. Morpheus codifies implicit user expectations as explicit Service Level Objectives (SLOs) inferred (using linear programming formulation) from historical data. Morpheus enforces SLOs using scheduling techniques that isolate jobs from sharing-induced performance variability, for instance, using a recurring reservation for repeated workloads. Dynamic reprovisioning is used to reduce performance variance due to failures. Cluster utilization is the key metric that Morpheus tries to optimize.

Graphene [52] is a global, static, distributed, suboptimal (heuristic) based scheduler for heterogeneous DAG jobs. Graphene focuses on scheduling long-running jobs and jobs that are hard to schedule first. The scheduler creates an offline schedule for the long and tough-to-scheduler jobs and then schedules the remaining tasks without violating dependencies. The scheduler aims to reduce the median job completion time for a given set of DAG jobs.

Carbyne [53] is a global, dynamic, distributed, suboptimal (heuristic) based scheduler that integrates with YARN. Carbyne allows jobs to altruistically give up their short-term fair share of cluster resources to improve Job Completion Time (JCT) across jobs while guaranteeing long-term fairness. However, the paper indicates that any workload that runs on YARN can use this scheduler, the primary use case that supports altruism for jobs with DAG and analytics workloads.

Tetrisched [54] is a global, dynamic, distributed, suboptimal scheduling system implemented in YARN for repetitive analytics jobs in datacenters. Tetrisched plans ahead by using a constraint solver to optimize job placement. It requires the user to supply explicit constraints with their jobs. It implements Space-Time Request Language (STRL) to specify space and time preferences. The is either specified or automatically generated based on SLOs and then used in Mixed Integer Linear Programming (MILP) solver to create a schedule. The schedule is revisited, and job placements are adjusted according to the system's current state.

Hydra [55] is a global, dynamic, distributed, sub-optimal (heuristic) based scheduler in YARN for data analytics jobs in Microsoft. This system is the successor to Apollo [56], which supported only one framework called Scope. Hydra extends YARN to support multiple frameworks with the main idea of a federation of resources. The cluster is subdivided into sub-clusters, each managed and scheduled by a separate entity. The higher-level scheduling entity allocates resources to coarse grain, and sub-cluster managers allocate specific machines to jobs.

2.3.5 Datacenter schedulers for user-facing services

Paragon [57] is a global, dynamic, centralized, suboptimal (greedy) scheduler that caters to resource heterogeneity and application interference. It uses information about the application's runtime. If the application is new, it profiles each application to extract the execution time estimates based on collaborative filtering techniques such as those employed in Netflix for filtering movies. Paragon compares the application similarities to estimate the runtime of a previously unseen application. The scheduler monitors each job's performance, and if the execution time deviates from the predicted time, the job is re-profiled. The job can be migrated using other mechanisms (VM migration).

Quasar [58] is a global, dynamic, centralized, suboptimal (greedy) scheduler mainly focusing on maximizing resource utilization and high application performance. Instead of relying on user-specified execution time, it takes a quality metric (throughput, latency) from the user. It converts that into a resource specification that satisfies the requested quality metric. It also profiles the jobs to collocate the applications so that they cause the least interference, thus satisfying the application performance. Quasar adjusts the resources by scaling up/down as the load increases/decreases, and if migration is supported, migrates jobs as needed.

Hawk [59] is a global, dynamic, distributed, suboptimal scheduler for frameworks like Spark [60]. Hawk combines centralized and distributed schedulers such that the centralized entity is responsible for scheduling long jobs, and short jobs are scheduled in a distributed fashion. Hawk uses randomized task stealing as part of scheduling data-parallel jobs on large clusters to ensure fairness for short tasks queued behind long ones. Hawk reserves a set of resources to run jobs obtained using a task-stealing mechanism. Hawk is specifically designed for data-parallel jobs and integrates with the Spark framework.

Firmament [61] is a global, dynamic, centralized, suboptimal (heuristic) scheduler with sub-second placement latency for large clusters by continuously rescheduling all tasks via a min-cost max-flow (MCMF) optimization. Firmament achieves this low latency by using multiple constraint-solving algorithms (MCFS) instances, solving the problem incrementally and via problem-specific optimizations. The main goal of Firmament is achieving high placement quality, thus increasing cluster utilization and decreasing placement latency. Firmament implements flow-based scheduling considering the entire workload, allowing rescheduling and priority preemption support. A major drawback is that Firmament assumes that the cluster state does not change while the algorithms run, which is not true in green datacenters.

Medea [62] is a global, dynamic, distributed (hierarchical), suboptimal (heuristic) based scheduler. Medea is designed for the placement of long and short-running containers. Medea introduces placement constraints to capture interactions among containers within and across applications. It follows a two-scheduler design: Medea applies an optimization-based approach that accounts for constraints and global objectives for long-running containers. Medea uses a traditional task-based scheduler for low placement latency for short-running containers. Medea gives the highest priority to long-running jobs and does not preempt or migrate jobs once placed.

Optimus [63] is a global, dynamic, distributed, suboptimal (ML approximation) based scheduler implemented on top of Kubernetes [64]. Optimus is a customized job scheduler for deep-learning clusters. Optimus uses online fitting to predict the training job's model convergence and, based on the performance, estimates training speed as a function of allocated resources for the job. Based on the online fitting, it dynamically allocates resources such that job completion time is minimized. Optimus is implemented on top of Kubernetes and works with scheduling containers within which the jobs run. Once scheduled, the jobs run to completion and are not preempted or migrated.

Kairos [65] is a global, dynamic, distributed (hierarchical), suboptimal (approximation) based scheduler. Kairos proposes using the LAS scheduling policy instead of directly estimating the task execution time. The scheduler has a central component for load balancing and a local component for handling scheduling and preemption. The scheduler preempts jobs only to resume them locally and does not migrate jobs from one cluster to another.

Gandiva [66] is a global, dynamic, central, suboptimal scheduler implemented on top of Kubernetes. Gandiva exploits intra-job predictability (time taken for each mini-batch iteration) to time-slice GPUs efficiently across multiple jobs leading to low job latency. This predictability w.r.t job performance is also used dynamically migrating jobs to better-fit GPUs leading to improved cluster efficiency. Gandiva is tailored for large machine learning training jobs, especially hyper-parameter search jobs, and exploits the repetitiveness of the jobs to predict the future resource requirements for that job.

Tiresias [67] is a global, dynamic, central, suboptimal scheduler tailored for Deep Learning training jobs. Tiresias uses two scheduling algorithms 1) Discretized Two-Dimensional Gittins index policy for a single server leading to minimizing JCT, and 2) Discretized Two-Dimensional LAS is information agnostic and aims to minimize the average JCT of all jobs in the system. When Tiresias's cluster manager has the distribution of previous job execution times, it chooses the discretized 2D-Gittins index and discretized 2D-LAS otherwise. Tiresias implements an RDMA network profiling library to intercept the network-level activities and determine the model structure of DDL jobs which aids in the job placement decision.

The other open-source and proprietary systems that are used in production include Apache Hadoop YARN, Apache Mesos [49], and Kubernetes [68], and proprietary schedulers Borg and Apollo [56] are also available for scheduling datacenter analytics jobs and user-facing services. These systems assume that most jobs are periodic and that completion times remain consistent with previous executions. This allows the schedulers to predict expected overall job completion times.

2.3.6 Schedulers for power-modulated datacenters

The schedulers discussed in the previous section do not demonstrate their suitability in green datacenter environments. To address the intermittent power supply from renewables, the existing heuristics schedulers [69] [70] [71] delay the deferrable jobs until the renewable power is adequate or the electricity price is low before the soft deadline of the jobs expires. Deferring the jobs may lead to poor QoS for the users. In [72], the scheduler employs a heuristic Mixed-Integer Linear Programming (MILP) formulation to minimize grid electricity when the electric grid is used and minimize the workload's performance degradation when using renewable energy sources.

Greenslot [73] tries to maximize green energy consumption while meeting job deadlines using the Least Slack Time First (LSTF) heuristic algorithm; the authors disclose that this leads to long delays for some jobs. Greenslot does not place a limit on how much cost is acceptable due to delays. Moreover, Greenslot does not suspend jobs once started, while our system has work-preserving suspend and restart capability. Recent work [74] presents a unified management approach for the thermal and workload distribution in datacenters. The objective is to minimize power consumption while satisfying thermal and Quality of Service (QoS) constraints. They implement a particle-based algorithm that requires adjusting some non-trivial number of parameters over multiple iterations.

2.4 Chapter Summary

This chapter introduced the job scheduling problem and discussed heuristic schedulers for diverse datacenter and workload configurations. Current heuristic cluster schedulers rely on handcrafted heuristics that are generic, easy to understand, and straightforward to implement. However, these schedulers may not always lead to achieving the best schedules foregoing potential performance optimizations by adhering to generic heuristics. Specifically, for power-modulated datacenters, the complexity increases due to intermittent power supply from renewables. Moreover, developing scheduling policies for power-modulated datacenters require expert domain knowledge and significant effort to devise, implement, and validate their efficacy requiring engineers to do the heavy lifting.

Chapter 3

Reinforcement Learning and Scheduling

This chapter introduces Reinforcement Learning (RL) basics and discuss how we map job scheduling in the power-modulated datacenter to the RL environment. Then we discuss reward engineering and types of reward functions. Finally, we present a brief related work on RL schedulers.

3.1 Reinforcement Learning Basics

Reinforcement Learning refers to a set of trial-and-error methods where an agent learns to make good decisions in a given environment via a sequence of interactions. The main components of RL are the **agent** and the **environment**. At each step of interaction, the agent sees an observation of the state of the world and then decides to take action. The environment changes based on the agent's action. The agent perceives a **reward** signal from the environment, a scalar that tells it how good or bad the current world state is. The agent's goal is to maximize its cumulative reward, called **return**. Figure 3.1 shows the RL agent observing the environment's current state, selecting an action from a set of allowable actions, and receiving a reward. The agent's action causes state transition by the environment (which may be deterministic or probabilistic). The reward depends on the current state and the action. This process of the agent observing the environment, taking action (after receiving some reward for the action), and transitioning to the new state is characterized as Markov Decision Process (MDP) [75].

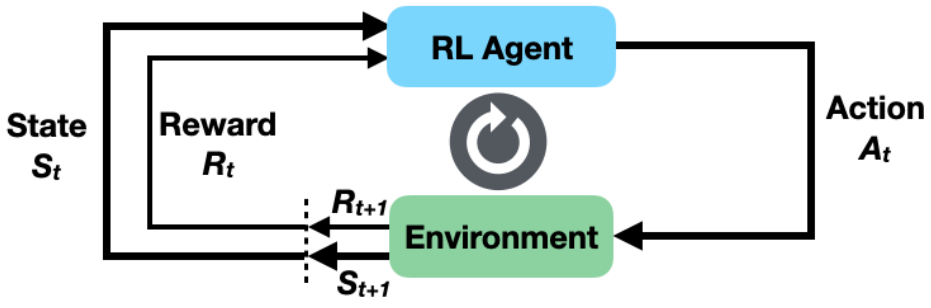


Figure 3.1: Typical RL agent observing the environment, taking action, and collecting reward/cost for the action.

A **state** is a complete description of the state of the world. There is no information about the world that is hidden from the state. An **observation** is a partial description of a state, which may omit information. In deep RL (DRL), the states and observations are represented by a matrix, real-valued vector, or higher-order tensor. For example, a visual observation may be represented using the RGB matrix of its pixel values; a robot's state may be represented by its joint angles and velocities.

The set of all possible valid actions in a given environment is called the **action space**. Environments can have **discrete action spaces** with only a finite number of moves available to the agent or **continuous action spaces**.

A **policy** is a rule that the agent uses to decide what actions to take. A policy can be deterministic, denoted by μ: $a_t = \mu(s_t)$, or it may be stochastic, denoted by π: $a_t = \pi(s_t)$. In deep RL, we have **parameterized policies**, i.e., policies whose outputs are computable functions that depend on a set of parameters (e.g., the weights and biases of a neural network) which can be adjusted to change the behavior using an optimization algorithm. The parameters of such a policy are denoted by θ and used as a subscript on the policy symbol to emphasize the connection: $a_t \sim \pi_\theta(s_t)$.

A **trajectory** is a sequence of states and actions in the world, $\tau = (s_0, a_o, s_1, a_1, \ldots)$. The very first state of the world, s_0, is randomly sampled from the start-state distribution.

State transitions (changes to the world between the state at time t, s_t, and the state at $t+1$, s_{t+1} are controlled by the natural laws of the environment and depend only on the most recent action, a_t.

3.1.1 Markov Decision Process

We consider Markov Decision Process (MDP, see [75]) consisting of a finite state space $\mathcal{S}$, a finite action space $\mathcal{A}$, a bounded reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbf{R}$ such that $\forall_s \in \mathcal{S}, a \in \mathcal{A}, |r(s,a)| \leq r_{max} < \infty$, a transition function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$, a policy $\pi : \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$, and an initial distribution $p_0 : \mathcal{S} \rightarrow [0,1]$.

At time step 0, an initial state S_0 is sampled according to $p_0(\cdot)$. Here $p_0(\cdot)$ represents a probability distribution on $\mathcal{S}$ whose probability mass function is p_0. At time step $t = 0, 1, \dots$, an action A_t is sampled according to $\pi(\cdot|S_t)$. Here $\pi(\cdot|S_t)$ represents a probability distribution on A, where the probability mass for $a \in A$ is $\pi(a|S_t)$. Then a reward $R_{t+1} = r(S_t, A_t)$ is emitted and a successor state S_{t+1} is sampled according to $p(\cdot|S_t, A_t)$. Here $p(\cdot|S_t, A_t)$ represents a probability distribution on $\mathcal{S}$, where the probability mass for $s \in \mathcal{S}$ is $p(s|S_t, A_t)$. In the discounted setting, we consider a discount factor $\gamma \in [0,1)$ to trade off the importance of long-term and short-term rewards. To summarize the possible future rewards starting from a state s following the policy π, we define the state value function v_π as $v_\pi(s) = \mathbb{E}_{\pi,p}[G_t|S_t = s]$. Here $G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1}]$ is the return at time step t.

Similarly, we use the action value function q_π to summarize the possible future rewards starting from a state-action pair (s,a) following the policy π as $q_\pi(s,a) = \mathbb{E}_{\pi,p}[G_t|S_t = s, A_t = a]$. The two value functions are related to each other as $v_\pi(s) = \sum \pi(a|s) q_\pi(s,a)$. Then the discounted total rewards J_π, which is the expectation of the summation of the discounted future rewards starting from time step 0, is $J_\pi = \mathbb{E}_{s \sim p_0(\cdot)}[v_\pi(s)]$.

3.1.2 Function approximation

When the state space $\mathcal{S}$ is too large, maintaining a look-up table (i.e., a vector $v \in \mathbb{R}^{|S|}$) as an estimate of v_π becomes intractable. Even worse, such a look-up table v cannot easily provide generalization across states. Function approximation is introduced to cope well with large state spaces and provide generalization across different states. Function approximation adopts a feature mapping $x : \mathcal{S} \rightarrow \mathbb{R}^K$, which maps each state s into a K-dimensional vector. As K is usually much smaller than $|\mathcal{S}|$, function approximation exhibits memory efficiency when the state space is large. As $x(s)$ is usually correlated for different $s \in \mathcal{S}$, function approximation naturally provides generalization across different states.

A Deep Neural Network (DNN) [76] is a function approximator that stores the policy for each (s, a) pair. Typically, DNNs have multiple (input, hidden, and output) layers, each containing several neurons. Interconnections between neurons are called edges, and their associated weights are called model parameters ($\pi_\theta(s, a)$). Then, learning consists of finding the suitable coefficients, or weights, by iteratively adjusting those weights along gradients that encourage less error. The input to policy and value networks is the state which includes jobs and available resources in the system. Based on the environment's feedback (reward), the neural net can use the difference between its expected reward and the obtained reward to adjust its weights and improve its interpretation of state-action pairs.

3.1.3 Reinforcement Learning algorithms

Two main approaches for policy learning are Q-learning and policy optimization. In Q-learning, the agent learns an estimate of the optimal action-value function (Q-function) and obtains the estimated optimal action by maximizing the Q-function. In policy optimization, the agent learns the optimal policy by directly optimizing the policy space. Specifically, if we denote θ as the parameters of the policy model that we are trying to learn, the policy optimization tries to update θ so that the policy model will generate better actions for a given set of observations $\tau =< s_0, a_o, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots >$. The policy optimization methods (e.g., actor-critic method) are generally applicable to a broader range of problems (including problems with continuous states) and tend to converge faster than the Q-learning method.

The **actor-critic** method leverages two networks 1) policy network and 2) value network. The policy network generates actions while the value network guides the training. The policy network (actor) takes the environmental state, at every step, as its input and outputs one or more actions determining which job to run. Value network (critic) helps to improve training efficiency. The value network's output (r_{est}) can be intuitively considered as the expected reward of a set of jobs based on the agent's current policy. After the policy network selects an action, the critic evaluates the new state to determine whether situations got better or worse than expected.

In an actor-critic method, the actor is trained to assign higher probabilities to actions that the critic determines will lead to higher monetary value. The critic then uses the improved actor to better estimate the expected return of selecting each action. Both networks rely on the state representation $\tilde{s}$ learned by the encoder network. Separating the encoder in this way lets us share parameters across the actor and critic training processes and reduces overall network size. However, for stability reasons, the encoder parameters are updated alongside the critic but not the actor [77].

3.2 Mapping Job Scheduling in Green Datacenters to RL Environment

To train the RL scheduler agent, we convert the green datacenter scheduling problem into a Markov Decision Process (MDP) with a state space $\mathcal{S}$ describing the current status of the datacenter resources, an action space $\mathcal{A}$ of new jobs, and a reward function $\mathbf{R}$ to be optimized. The operation of the datacenter - including receiving new jobs and placing scheduled jobs on available resources - becomes the MDP transition function, $\mathbf{T}$. Figure 3.2 illustrates the RL scheduler agent interacting with the green datacenter environment.

The scheduler's objective is realized with rewards that the agent receives. Reward, a single scalar function, is a combination of the positive reward or associated cost for action in a given state. The scheduler's reward function is configurable, allowing rewards to be either sparse (rewards only obtained on job completion) or dense (smaller rewards to guide the agent after each step).

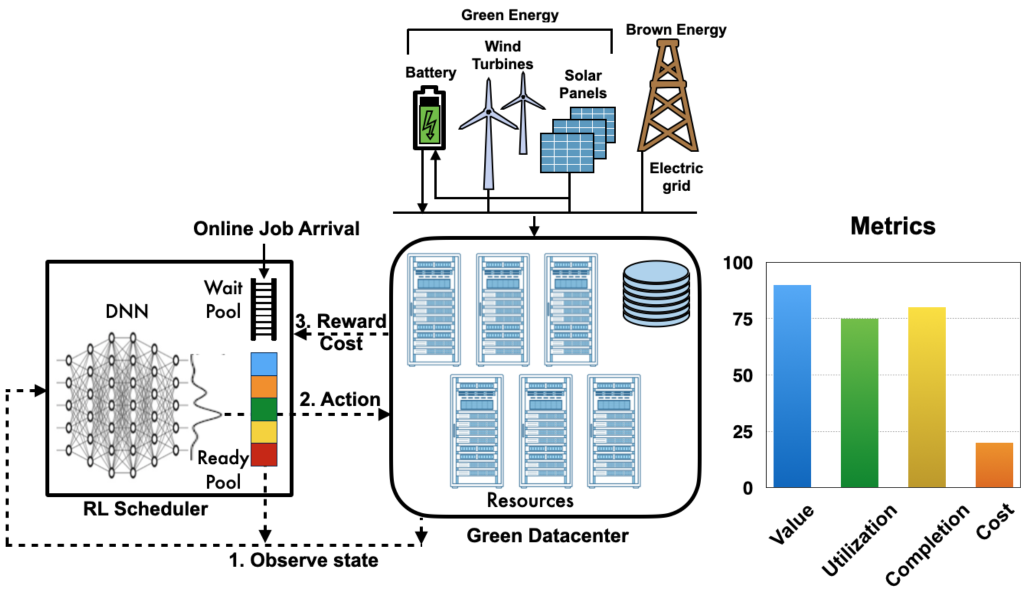


Figure 3.2: RL scheduler interacting with green datacenter (powered by renewable and brown energy sources), optimizing for various metrics.

3.2.1 State space

A state is a representation of the environment, and the state space is the set of all possible states. In the power-modulated datacenter environment, the state space $\mathcal{S}$ includes information about jobs, resources, and resource availability (based on power generation predictions).

3.2.1.1 Resources

Resources are represented in an image format of shape $(\text{time\_horizon}, \text{resource\_types}) \times \text{max\_resources}$, with grey pixels indicating available resources. As jobs are scheduled on the resource pool, segments of the image are occupied by colored rectangles representing jobs' resource requirements and duration. Figure 3.3 illustrates the cluster image (10 CPUs, 10 GPUs for 24-time units) and the allocation of each resource to jobs scheduled for service, starting from the current timestep and looking ahead 24-time units into the future.

The power availability feedback is not directly provided to the scheduler agent as part of the state information. Instead, the resource pool expands and contracts based on the datacenter's power supply at any given time. Power availability dictates when and how many resources are turned on or off. Therefore, power prediction data is an integral part of the state space, i.e., as power availability changes, the corresponding resource availability is reflected in the state supplied to the scheduler agent. Unavailable resources, due to power constraints, are marked black on the resource image. For instance, at timestep 22 (Figure 3.3), 70% of power is available, so 70% of the resources are on, and 30% are shut down. Similarly, at timestep 23, 80% of power is available, meaning 80% resources are on and 20% are shut down.

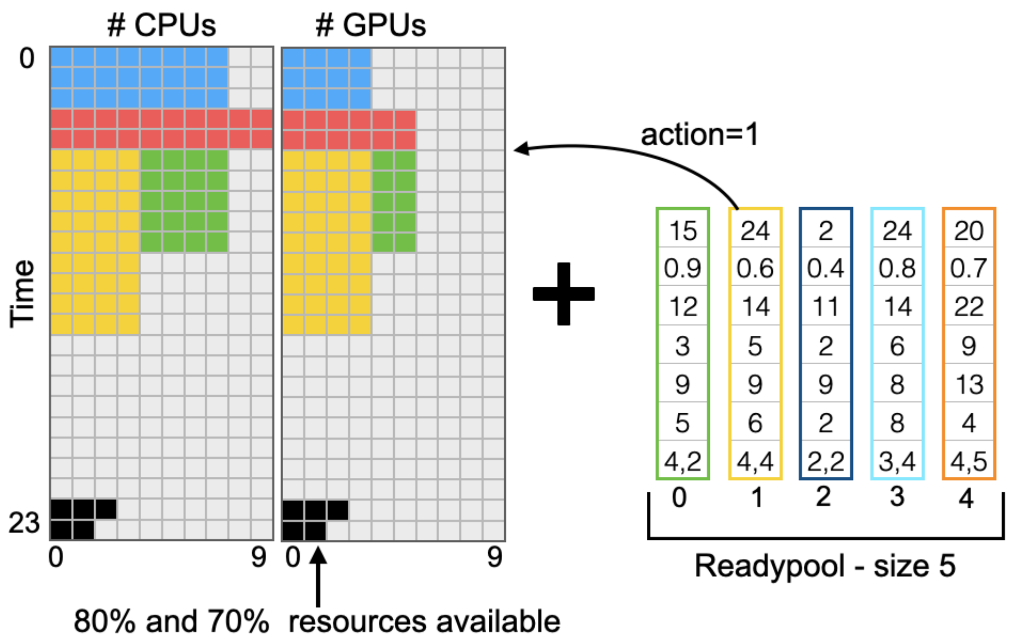


Figure 3.3: State information with Resources and Jobs: A 10 CPUs, 10 GPUs cluster with time-horizon=24, and ready_pool size=5. The unavailable resources are indicated in black squares (bottom).

3.2.1.2 Jobs

Each job, $j_i \in J$, has associated meta-data (described in Chapter-4, table 4.1). Jobs can have additional information, e.g., input/output files, that may affect completion time. We do not account for additional overhead in our environment. A workaround to accommodate such overheads is adding extra slack time to the job's expected finish time.

In our system, jobs can be in one of three locations: 1) wait_pool, 2) ready_pool, or 3) scheduled on the resources. The wait_pool is where jobs first arrive. The jobs from wait_pool are moved (e.g., FIFO or QoS order) to the ready_pool, where they can then be scheduled on the resources. The jobs in the ready_pool are represented in two formats 1) Vectors and 2) Images. In vector format, each job's vector consists of *job_value, qos, qos_violation_time, enter_time, expected_finish_time, duration* and *resource_requirement.* Additional meta-data for each job is calculated after the job is admitted, e.g., remianing_runtime (if the job gets suspended) and qos_violation_time. In image format (see §3.3), each job is represented as a rectangle with a horizontal side representing the number of resources requested and a vertical side representing the job length (time units). Both formats generate nearly identical performance results, but the vector format is compact compared to image only format.

In Figure 3.3, the ready_pool size is 5 (with job indices $0-4$) and has 5 jobs (represented in vector format). The yellow job (at ready_pool[1]) requires 4 CPU and 4 GPU units for the next six timesteps, and the job's value is 24. The jobs are processed over some fixed T timesteps. The time horizon shifts after processing jobs during that timestep, with the job metadata vectors updated and the resource image advancing by one row. As the time horizon shifts, the energy available from renewables (and battery) dictates the availability of resources for placing new jobs. To make scheduling decisions, the scheduler agent continuously observes the state - jobs, resources, and resource/power availability.

QoS value: Users may have different utility functions, i.e., users are willing to pay different amounts for different jobs based on their importance. The user picks the required QoS for that job based on the user's willingness to pay for the job (e.g., spot instances [78] [79] [80]). The QoS value is specified as a percentage of the time the user wants his job to run. The qos_violation_time specifies the upper bound by which time the job must finish executing. The higher the QoS percentage, the higher the job's value.

$$qos\_violation\_time = \frac{1}{QoS} \times run\_time \tag{3.1}$$

If a job remains in the system past qos_violation_time, it incurs negative rewards every time step after that. A higher QoS value means the closer the job's completion time to the expected_finish_time. If a user wants 0.95 (95%) QoS value and specifies expected_finish_time = 10 hours, then the job must be completed within 10.5 hours. Expressing QoS value in percentages gives an upper bound of when a user can expect his job to finish. The idea is similar to Least Attained Service (LAS) [81] in that a job receiving more service is suspended and later restarted if preempted.

3.2.2 Action space

The set of all possible actions the learning agent can take in the environment. The action space for a datacenter with ready_pool size n is a set of $n+2$ discrete options $\mathcal{A} = \{j_0, j_1, \ldots, j_n, \mathit{suspend}, \mathit{no\_op}\}$. The actions $\{a = j_i, \forall i \leq n\}$ schedule the ith ready_pool job j_i on available resources. The job's colored rectangle is added to the first available slot in the resource image with enough free space to schedule it. The action $a = \mathit{suspend}$ is used to suspend an incomplete job and replace it with a higher value. The suspend action is work-preserving, in that a suspended job resumes from the point it was stopped at and not from the beginning. The suspended jobs are re-queued after updating the remaining run time, along with the other ready jobs. Although our scheduler framework supports checkpoint-restart capability [82], the feature was turned off for the experiments discussed in this paper. Finally, the action $a = \mathit{no\_op}$ means that the scheduler agent does not want to schedule (e.g., resources requirements cannot be satisfied) or suspend any jobs in that timestep. In Figure 3.3, $\mathit{action} = 1$ (at ready_pool[1]) schedules the yellow job to run on the available resources. Additionally, the action space can extended to include new actions (e.g., migrate) as needed.

3.2.3 Rewards

The DRL scheduler's objective is realized with rewards that the agent receives. Rewards, which are scalars given by the reward function $\mathbf{R}(s_t, a_t, s_{t+1})$, are a combination of the positive reward or associated cost for action in a given state. Some actions collect positive rewards, while other actions accrue negative costs. For instance, if a job, j, running on a resource, collects a positive reward proportional to its value. A job's value, $j._{value}$, is calculated based on the type of resources requested, duration, and QoS value. If a job is delayed and QoS is violated, it collects a negative reward. Negative reward indirectly encourages fairness, ensuring low QoS value jobs are not delayed or starved. Other costs and rewards can be incorporated into the reward function. Our DRL scheduler's objective is to maximize the total job value from finished jobs, $|J_{finished}|$ expressed as,

$$Total\ Job\ Value = \sum_{i=1}^{|J_{finished}|} j_i.value \tag{3.2}$$

where $j_i.value = ((\#cpu \times cpu\_price\_per\_unit \times duration) + (\#gpu \times gpu\_price\_per\_unit \times duration)) * qos$.

A direct calculation of value is the price the user is willing to pay to run a job. Total Job Value is both an application-centric and resource-centric metric; the emphasis is on processing as many user jobs as possible, which may increase resource utilization. By processing as many jobs as possible, we essentially maximize the total value we gain from running those jobs. Even a small improvement in total job value can generate millions of dollars in savings for the service providers. Other common objective functions (utilization, makespan, and system throughput) are driven by system-centric parameters that enhance throughput and utilization rather than improving the utility of application processing. These systems treat resources as if they all cost the same price and the results of all applications have the same value, even though this may not be the case in reality.

3.2.3.1 Reward engineering

In RL, the reward function generates a value that the agent maximizes to achieve the desired goals. Every time we change the reward function, we may explicitly set a new and different goal for the agent. Designing a suitable reward function to elicit a specific behavior is known as reward engineering or reward shaping. Rewards can be simple (e.g., maximize utilization), compound (e.g., maximize utilization and total job value), or opposing (e.g., maximize total job value while minimizing job delays).

The RL scheduler's objective is realized with rewards that the agent receives. Typically, the reward is a single scalar function with a combination of the positive reward or associated cost for action in a given state. Some actions collect positive rewards, while other actions accrue negative costs. For instance, if a job runs on a resource, it collects a positive reward proportional to its value. If a job is delayed and QoS is violated, it collects a negative reward. Other costs and rewards can be incorporated into the reward function.

The reward functions are classified into three categories based on the number of objectives the reward tries to maximize or minimize. The objectives could be to maximize job value or resource utilization, minimize job delay or cost, or other metrics relevant to the datacenter operators.

The three reward categories are:

- Simple reward function: This type of function maximizes or minimizes a single objective.

- Compound unidirectional function: This type of function tries to combine multiple objectives, and the objectives are either maximized or minimized. The objectives are either increasing or decreasing in the same direction.

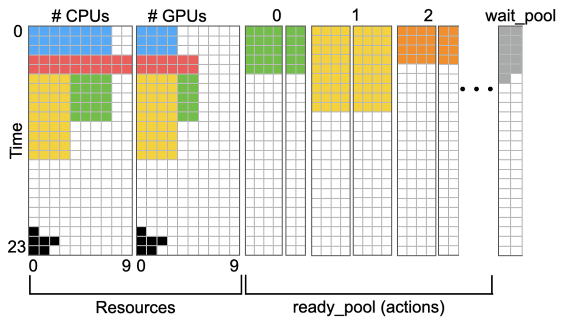
- Compound dual (opposing) function: This type of function tries to maximize one or more objectives while minimizing another.

The scheduler's reward function is configurable, allowing rewards to be either sparse (rewards only obtained at specific intervals or on job completion), dense (smaller rewards to guide the agent after each step), or mixed (sparse and dense) rewards.

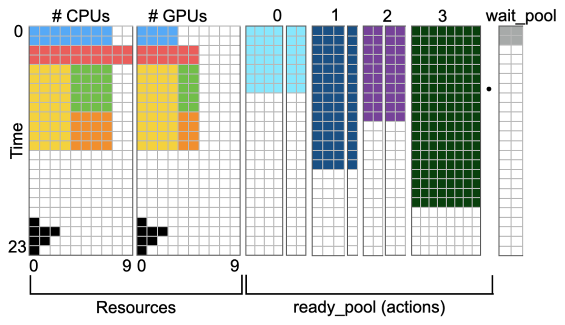
3.3 Training scheduler agent with the simulator

The basic idea in policy gradient methods is to estimate the gradient by observing the trajectories (samples) obtained by following the policy. The policy π is initialized randomly or using parameters from a previously trained model.

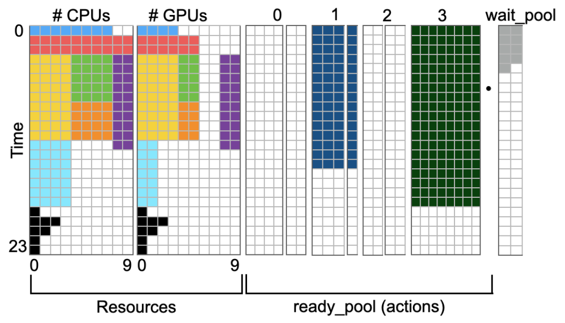
We train the agent with multiple examples (job sets) of job arrival sequences. We simulate $Max\_Episodes$ episodes for each job set to explore the probabilistic space of



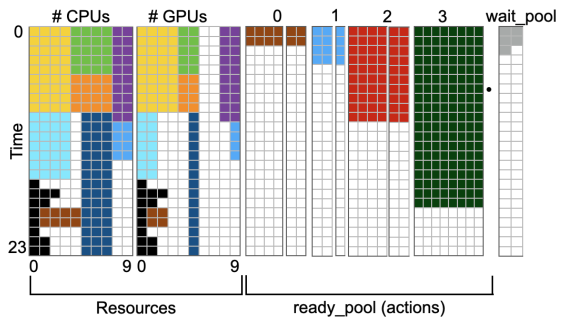
(a) Next action, a=2.



(b) Job at 2 gets scheduled (orange job in (a)). New jobs are in the ready queue.



(c) Next actions = 0 and 2.



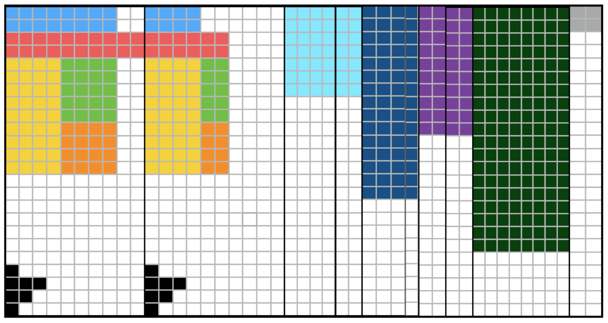
(d) Jobs at 0 and 2 are scheduled (light blue and purple jobs in (b)). After 3 time steps, actions = 0, 2, 1. Jobs at 0, 2, and 1 get scheduled.

Figure 3.4: RL scheduler agent training steps with jobs getting scheduled on the resources and new jobs arriving at the ready_pool.

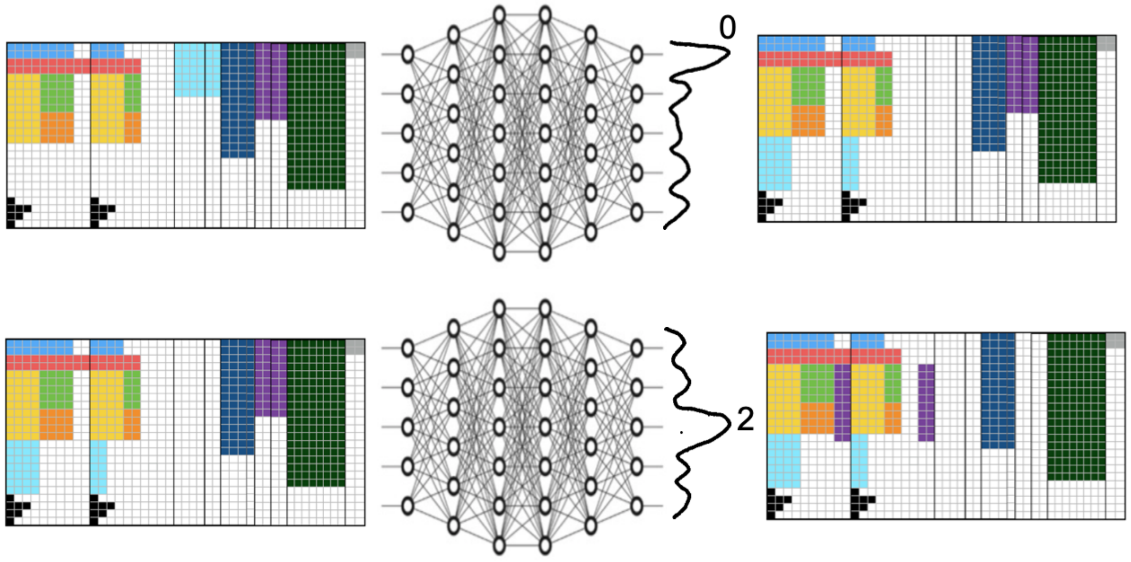
possible actions, $\pi(a_t, s_t)$, with the current policy.

At each step of an episode, the system obtains a state of the current job placements from the previous operations. We then select the next job by sampling a probability distribution over jobs in the *ready_pool* (Figure 3.4). After that job is scheduled on available resources, the system transitions to the next stage. Repeating these transitions means all the jobs in the job set are scheduled on the resources. This process of sampling jobs and state transitions is illustrated in Figure 3.4a through Figure 3.4d.

In Figure 3.4a, two jobs (represented in blue and red) from the previous iteration are running, and in the current time step, new jobs (yellow and green) are placed while the agent samples another job (orange). Finally, the state transitions to Figure 3.4b, where the time horizon shits (2 units of the blue job are remaining), the jobs (yellow, green, and



(a) Current state with 5 running jobs, 4 jobs in *ready_pool* and 4 jobs in the *wait_pool*.



(b) Policy network picking action = 0 and 2.

Figure 3.5: Image input to the policy network; action output from the network.

orange) are scheduled to run, and four new jobs (light-blue, dark-blue, purple and green) arrive in the ready queue from the *wait_pool*. The number of jobs in the wait queue (grey) is reduced by four. The time horizon shits again, and this forms the new state, and its corresponding image format is illustrated in Figure 3.4c.

At each time step, the cluster state represented in the image format (Figure 3.5a) becomes the input to the agent (policy network), illustrated in Figure 3.5b. The agent samples the possible actions (i.e., pick jobs in the *ready_pool*), the action with the highest probability represents the next job to be scheduled, and so on. In this case, the agent samples actions 0 and 2, i.e., pick two jobs (light-blue and purple) to be scheduled.

We record the state, action, and reward information ($\tau =< s_0, a_o, r_1, s_1, a_1, r_2, \dots >$) for all time steps up to $Max\_episode\_length$ for $Max\_Episodes$ episodes. We use reward values to compute the cumulative reward (discounted) at each time step t of each episode. The total value of all the jobs with the final placement state is the reward.

3.4 Scaling

A recurring challenge in RL has been solving problems with a huge state or action space, and most RL environments suffer from state space explosion problem as the state space increases. Given a state and action space, there are at most $|A|^{|S|}$ unique policies. This means that the size of the problem's solution space grows exponentially with each additional feature in our state [83]. This is commonly described as the "curse of dimensionality." With complex problems, we note that training time can become highly unrealistic and computational complexity intractable.

One solution that helps alleviate the state space explosion problem is splitting up large state spaces into smaller ones through state space decomposition. This allows us to distribute computation and accelerate training. The key idea is to use smaller neural networks to learn the dynamics of the decomposed state sub-spaces, with another neural network considering the relatively less frequent interactions between the different state sub-spaces. We proposed two possible solutions to this problem in chapter-8.

The flexible design of our datacenter simulator allows exploring various design options that would enable us to compare the different schedulers' performance for different system configurations, including datacenter size, system load, and workload distributions. For instance, we can model a small-scale datacenetr with 10 to 100 resources or a medium-scale datacenter with 100 to 1200 resources with a system load.

A comprehensive list of the various datacenter-specific parameters and RL-specific hyperparameters is provided in Appendix-A. These parameters can be adjusted to simulate various system configurations and to fine-tune the RL agent's performance.

3.5 Related Work - RL Schedulers

Reinforcement Learning has been used for a variety of tasks, including robotics [83] [84], manufacturing plants [85] and computer game playing [86]. RL is used for decentralized packet routing in a switch [87]. Recently, learning has been applied to designing congestion control protocols using a large number of offline [88] or online [89] experiments. Automatic traffic optimization in datacenters has shown promising results in [90]. This section briefly discusses reinforcement learning methods applied to job scheduling problems in diverse contexts. We will discuss topic-specific relevant related work in each chapter to better contextualize our work.

The first RL formulation of job-shop scheduling in the RL framework was proposed in [26]. The authors describe local, static, central, and suboptimal (approximation) scheduling in the RL setting. The system is a repair-based scheduler that first starts with a critical-path schedule and repairs constraint violations incrementally with the goal of finding a short conflict-free schedule. The authors propose temporal difference algorithms to train a neural network to learn a heuristic evaluation function over states. The TD evaluation function is a one-step look-ahead search procedure to find good solutions to scheduling problems. A small number of NASA space shuttle payload processing tasks were evaluated, and the results confirmed that RL could provide a new method for constructing high-performance scheduling systems.

DeepRM [91] is a global, dynamic, central, suboptimal (approximation) scheduler implemented as a deep RL framework. This work describes the design and demonstrates a simple multi-resource cluster scheduler. DeepRM operates in an online setting where jobs arrive dynamically and cannot be preempted once scheduled. DeepRM learns to optimize various objectives, such as minimizing average job slowdown or completion time. DeepRM employs a standard policy gradient reinforcement learning algorithm for learning from experience.

Minerva [92] is a global, dynamic, central, suboptimal scheduler for bottleneck detection in distributed factory settings. The work uses a similar technique as [26] in that it schedules jobs for a fixed interval, starting with scheduling bottleneck jobs and then rearranging the jobs that violate constraints until no more violations exist in the schedule. This work implements neural network-based Q-function approximation and Q-learning algorithm.

cuSH [93] is a cluster scheduler using RL to schedule jobs on heterogeneous clusters. The work is primarily influenced by [91] and implements similar DRL (CNN instead of DNN) techniques for finding near-optimal scheduling policies. This scheduler addresses scheduling for a single HPC cluster, not geographically distributed ones.

Decima [94] uses DRL and neural networks to learn workload-specific scheduling algorithms without human instruction beyond a high-level objective, such as minimizing average job completion time. In Decima, new representations for jobs' dependency graphs are implemented to support jobs running on the Spark framework.

Harmony [95] is a DRL cluster scheduler that places ML training jobs to minimize interference and maximize performance (training completion time). The DRL employs an actor-critic algorithm to stabilize training and improve convergence, job-aware action space exploration, and experience replay. Harmony employs an auxiliary reward prediction model, which is trained using historical samples to produce a reward for unseen placement.

All the systems above assume that power is constant in their datacenters, i.e., machine and network failures are the only failure modes. While server failures still exist in our environment, we are focusing on job scheduling problem in power-modulated datacenters (refer to Chapter-4).

Interest in exploring RL methods for job scheduling has exploded in recent years. While there are many implementations, comparing with State-of-the-art (i.e., other RL scheduler implementations) is not straightforward. RL scheduler's performance depends on 1) RL-specific factors and 2) the scheduling environment. The RL-specific factors include various learning algorithms (REINFORCE, A2C, PPO, CPPO, offline/online), neural configurations (differ based on problem size), optimization objectives expressed as reward functions (e.g., Job Value, Utilization, Makespan, Wait-time/Delays) and hyperparameters (including batch size, learning rates, and neural net activation functions). The scheduling environment factor includes state representation (e.g., image only, vectors only, image+vector), environment (e.g., traditional datacenters, power-modulated datacenters), workload types (e.g., HPC workloads, Cloud workloads, analytics, and Machine Learning workloads), and scale of the problem (e.g., small, medium, and large scale clusters) among others. Not making all the above changes results in unfair comparisons. Making all the above changes could result in a completely new or different RL Scheduler implementation than the original. Therefore our implementations, discussed in Chapters-5, 6 and 7, are compared with heuristic scheduling policies (refer to Chapter-5 for details on the heuristic scheduling policies).

3.6 Chapter Summary

This chapter introduced Reinforcement Learning formalized by the Markov Decision Process and discussed Actor-Critic methods. We described how we map job scheduling problem in the power-modulated datacenter to the reinforcement learning environment. We discussed reward engineering and types of reward functions and presented related work on RL schedulers.

Chapter 4

Power-Modulated Datacenter Simulator

The power-modulated (green) datacenter is a datacenter co-located at or near renewable energy sources. Co-locating the datacenters near green energy production sources reduces energy transmission loss and increases the energy available to the datacenters. Various renewable sources can power the datacenter with the provision to store (batteries) excess energy from renewables. Additionally, the datacenter is connected to the electric grid to support critical infrastructure when energy from renewables and batteries cannot sustain the load. In this chapter, we aim to design a green datacenter simulator controlled by heuristic and RL scheduling policies. We present a unified green datacenter simulator that allows experimenting with synthetic and real workloads, integrating multiple renewable energy sources and batteries to power the datacenter.

4.1 Introduction and Motivation

Due to many practical reasons, such as the cost of resources, time scale, presence of other loads on the clusters, or lack of access to the facilities, experimental evaluation cannot be adequately performed on real systems. To obtain reliable results, simulations must be repeated with different setups using the same controllable conditions that simulate different real-life scenarios. The Online RL agent learns policies by repeated trial and error. Training the RL agent on a real system requires dedicated access to resources to not interfere with other users' jobs. Also, resource ramping based on intermittent power supply may cause significant physical wear and damage to the machines.

Many simulators have been developed to mitigate the above problems [96]. If properly designed, simulators are very useful since different system configurations and workloads can be used to evaluate existing or proposed solutions. While many ad-hoc simulators exist for clusters, they do not sufficiently meet our requirements. For example, these simulators do not feature the capability to control resource availability based on the power supply. Also, the simulator must support various workloads, job arrival patterns, and scheduling policies (heuristic and RL policies). Furthermore, we needed a lightweight simulator suitable for RL training and evaluation. The amount of work needed to modify the existing simulators is quite extensive.

Many researchers have proposed a variety of simulated environments to train and evaluate RL schedulers. Unfortunately, these datacenter environments are custom-built for specialized workloads, including machine learning training jobs [22], DAG jobs [97], and HPC jobs [24]. Few other simulated datacenter environments focus on energy efficiency [23], application profiling and monitoring [98], and delayed scheduling [99] based on the availability of power. None of the proposed environments simulate a green datacenter where resource ramping is a crucial feature.

We need a unified green datacenter simulator driven by synthetic or real workloads that integrates multiple renewable energy sources and Energy Storage Devices (ESDs) to power the datacenter. The simulator must support resource scaling up and down, allowing the practitioners to experiment with datacenters of different capacities. Additionally, the simulator must allow the resource pool to expand and contract in response to the intermittent power supply from renewables. Finally, the simulator supports configuring various time horizons, job arrival rates, job size distribution, and job durations for synthetic workloads. We now present an overview of our green datacenter simulator.

4.2 Datacenter Simulator - Overview

This section describes the **discrete event simulator** that simulates the power-modulated datacenter environment. A discrete event simulation (DES) models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation time can directly jump to the occurrence time of the next event.

Our simulator aims to evaluate and compare the performance of various scheduling policies under different operating conditions, e.g., power availability levels, workloads, and system load. Figure 4.1 depicts a green datacenter simulator interacting with the scheduler. The scheduler component can be any scheduling mechanism, including heuristic (e.g., FCFS, SJF) or RL scheduler. In the green datacenter simulator, the state space represents the current state of the resources and jobs, actions represent the scheduler's next decision (schedule, suspend, or no-op), and a reward signal that the scheduler receives for an action. The state of all the resources (available, currently in use and unavailable resources due to power supply constraints) and jobs (running and waiting to be scheduled) is shared between the simulator and the scheduler component.

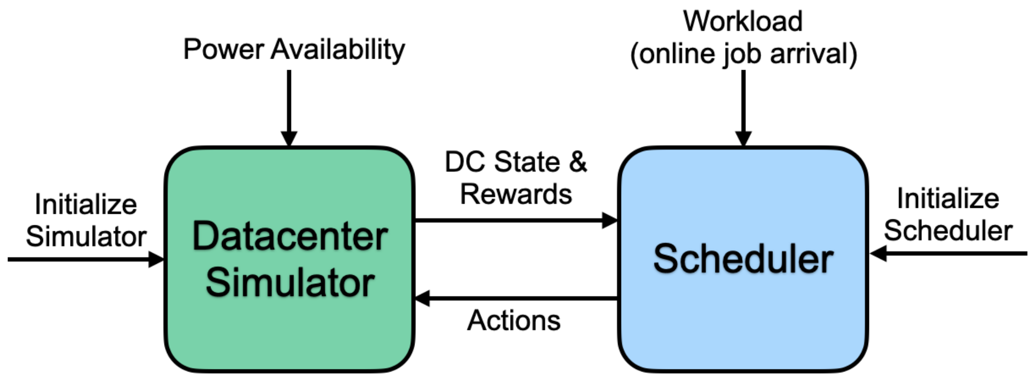


Figure 4.1: Scheduler interacting with the datacenter simulator, executing actions generated by the scheduler, and sharing the datacenter state.

First, the datacenter simulator is initialized with the required configuration (e.g., resource pool size, job arrival rate, and workload type). After initialization, the datacenter simulator continuously interacts with the scheduler component for the duration of the simulation, simulating the actions dictated by the scheduler. The datacenter state changes under two broad circumstances, 1) simulate actions from the scheduler component, and 2) simulate resource availability based on the power supply.

As the jobs arrive, the workload is fed to the scheduler component. The scheduler decides what actions to execute next based on the jobs and the current system state. The action sequence is sent to the simulator component, which simulates each action in the datacenter environment; after each action, the system's state changes. This new state change is visible to the scheduler component. Based on the new state, the scheduler generates new actions that the datacenter simulator executes, and the loop continues. The power availability (synthetic or real traces) dictates the number of resources available at any given time in the datacenter. The datacenter simulator turns resources on/off based on the power supply. The following sections describe the simulation components: resources, jobs, actions, and rewards.

4.2.1 Assumptions

When designing a simulator, it is critical to understand the scope of the system, user requirements, workloads, and underlying environment. This section discusses the assumptions made when designing our unified power-modulated datacenter simulator.

- Power generation prediction data, based on local weather predictions, is available to the simulator.
- Job meta-data is provided, and any missing information is handled in the admission control module beforehand.
- Checkpoint-restart capability behaves as expected and is work-preserving to suspend and resume jobs. The work done between two checkpoint events is lost when a node fails during that window. If the job's metadata includes a high-reliability guarantee, then the scheduler must restore the state of the job from its latest checkpoint. If

a job has specified the highest priority, it is immediately restarted from its latest checkpoint.

- Other intermittent failures, including machine and network failures, are not handled in the simulator module.
- Essential infrastructure, including networking, storage, and head nodes, are always operational.

4.3 Simulation Process

The main components of the simulation process include initializing the system with values provided by the user, executing the actions generated by the scheduler component, and generating a reward signal that indicates how good the action was. The main simulation loop is outlined in Algorithm 1.

Algorithm 1: Simulation main loop

```
1  while iter_count < max_iterations do
       // repeat, possibly with new seed
2      reset()
3      initialize_power_availability()
4      initialize_workload()
5      initialize_system()
6      Function Execute_gdc_simulation():
7          while simulation_steps_count < max_simulation_length do
8              if action == SCHEDULE then
9                  Function_schedule()
10             end
11             if action == SUSPEND then
12                 Function_suspend()
13             end
14             if action == NOOP then
15                 Function_noop()
16             end
17         end
18 end
```

4.3.1 Initialization

During the initialization step, we set up the simulator with initial parameters supplied by the user. The simulation parameters are supplied to the system via a file that the user modifies based on the simulation requirements. The initialization step includes 1) reset() routine and 2) init() routine. The reset() routine resets all the system parameters to default values. The init() routine has three sub-routines namely initialize_power_availability(), initialize_workload() and initialize_system(). The initialize_power_availability() subroutine, based on the user's input, selects between synthetic power availability or uses real power data from a CSV file. The initialize_workload() subroutine, based on the user's input, selects between synthetic workload or real workload traces (ANL or others) from a CSV file. The initialize_system() subroutine, based on the user's input, initializes various datacenter specific parameters, including resource size, number of resource types (CPU, GPU), time horizon size, resource price, etc. The system parameters are supplied from a python file (e.g., baseline_envs.py).

4.3.2 Jobs

In our system, jobs can be in one of three locations: 1) wait_pool, 2) ready_pool, or 3) scheduled on the resources. The wait_pool is where jobs first arrive. The jobs from wait_pool are moved to the ready_pool, where they get scheduled on the resources. The simulator simulates running jobs on the available resources. The jobs are dispensed to the simulator by the scheduler component. The scheduler decides on the action, and the simulator executes the corresponding action.

Our simulator supports various workloads, including synthetic and real traces. During initialization, the user can select the type of workload to simulate. Although the simulator supports any workload, we used synthetic and HPC workloads since those workloads closely match our use case. A detailed description of the workload model is presented in §4.4.

4.3.3 Resources

Resource availability is represented, of shape (time_horizon, resource_types $\times$ max_resources), in an image format. As jobs get scheduled on the resource pool, segments of the image are occupied by colored rectangles representing jobs' resource requirements and duration. Figure 3.3 illustrates the cluster image (10 CPUs, 10 GPUs for 24-time units) and the allocation of each resource to jobs scheduled for service, starting from the current timestep and looking ahead 24-time units into the future. A detailed description of the power availability model is presented in §4.5.1.

The power availability feedback is directly provided to the simulator, which updates the state information based on the power availability. The resource pool expands and contracts based on the datacenter's power supply at any given time. Power availability decides when and how many resources are turned on or off. Therefore, power prediction data is an integral part of the state space, i.e., as power availability changes, the corresponding resource availability is reflected in the state supplied to the scheduler agent. Resources unavailable due to power constraints are marked black on the resource image (refer to §4.5).

4.3.4 Actions

The action space for a datacenter with ready_pool size n is a set of $n + 2$ discrete actions $\mathcal{A} = \{j_0, j_1, \ldots, j_n, suspend, no\_op\}$. The action $a = suspend$ suspends an incomplete job and replaces it with a higher value. The scheduler can generate one or more actions for each simulation time step. The maximum number of actions for one simulation step is bounded by $n + 2$ (i.e., $ready\_pool\ jobs + suspend + noop$). Actions are generated either by heuristic or DRL schedulers. The following sections describe the actions and how the simulator simulates those actions in detail. Actions specific to the RL scheduler are presented in Chapter-3, section 3.2.2.

4.3.4.1 Schedule action

The actions $\{a = j_i, \forall i \leq n\}$ schedule the ith ready_pool job j_i on available resources. The job's colored rectangle is added to the first available slot in the resource image with enough free space for it to be scheduled. The simulation steps for schedule action are outlined in Algorithm 2.

Algorithm 2: Schedule Action

```
1  Function Execute_schedule():
2      if resources_available (CPU and GPU) then
3          move readypool[i] to available resources
4          update the state (image) after scheduling the new job
5          update resources (CPU and GPU) count
6      end
7      if suspend='yes' then
8          resume a job from the suspendpool
9      end
10     while available_slots > 0 do
11         new_job = get_job_from_waitpool()
12         readypool[available_slot] = new_job
13     end
14     update the qos_violation_time for all the jobs in readypool
15     increment simulation_time, simulation_steps_count
16     update the system state
17     collect rewards and costs
18     new_state = get_observation()
19     return new_state, reward
```

The get_observation() function converts datacenter's current state to an observation that can be fed to the neural network. The observation can be of two types 1) image-only format and 2) image+vector format.

4.3.4.2 Suspend action

The suspend action is work preserving, in that a suspended job resumes from the point it was stopped at and not from the beginning. The suspended jobs are re-queued after updating the remaining run time, along with the other ready jobs. The simulation steps for suspend action are outlined in Algorithm 3. Although our scheduler framework supports checkpoint and restart capability [82], we have not included experiments to demonstrate

Algorithm 3: Suspend Action

```
1  Function Execute_suspend():
2      if running_jobs then
3          while num_running_jobs > 0 do
4              select a candidate job to suspend from running_jobs list
5          end
6          suspend/remove candidate job from running_jobs list
7          add suspended job to suspended_jobs list
8      end
9      update the qos_violation_time for all the jobs in readypool and suspended_jobs
10     increment simulation_time, simulation_steps_count
11     update the system state
12     collect rewards and costs
13     new_state = get_observation()
14     return new_state, reward
```

checkpoint/restart capability since it is out of the scope of this dissertation.

4.3.4.3 No-Op action

Finally, the action $a = no\_op$ means that the scheduler agent does not schedule (e.g., resources requirements cannot be satisfied) or suspend any jobs in that timestep. In Figure 3.3, $action = 1$ (at ready_pool[1]) schedules the yellow job to run on the available resources. The simulation steps for no-op action are outlined in Algorithm 4.

Algorithm 4: No-Op Action

```
1  Function Execute_noop():
2      while available_resources > 0 do
3          new_job = get_job_from_waitpool()
4          readypool[available_slot] = new_job
5      end
6      update the qos_violation_time for all the jobs in readypool and suspended_jobs
7      increment simulation_time, simulation_steps_count
8      update the system state (e.g., job finishes)
9      new_state = get_observation()
10     return new_state
```

4.3.5 Simulator extensions

Our implementation assumes a “pool of resources” (CPUs and GPUs), which allows the scheduler to make fine-grain per-resource scheduling decisions. We do not account for data (and task) locality, but our simulator formulation can be extended to accommodate machine boundaries by allocating resources per machine model. In the future, we will incorporate machine boundaries in the state representation. The machine boundaries will provide a notion of nodes instead of a pool of resources. The reasoning behind this restriction is that the job’s performance often depends on whether the resources are local to a node or on different nodes. With this new state representation, a job can request resource type and the number of resources of each type, i.e., $(\#cpu, \#gpu)$. If a node does not have the requested number of $(\#cpu, \#gpu)$ resources, then the job cannot be scheduled on that resource. In such a case, the agent gets a higher reward if a job’s resource requirement is met on the same machine.

Typically, a datacenter will house heterogeneous resources, including but not limited to multiple generations of CPUs from various vendors, multiple generations of GPUs from different vendors, and multiple storage types (SSDs, Hard Disks, Archives) to cater to the diverse needs of the users. The current implementation simulates a homogeneous resource pool for all the experiments. The current implementation does not simulate unexpected machine failures, storage devices, or delays caused due to I/O and networking. We can extend the simulator to incorporate storage and network delays by adding buffer times to each job’s start and end times. Currently, we simulate two types of resources (CPUs and GPUs). Other resource types (e.g., memory, storage requirement) can be added easily by modifying the num_resource_types parameter and workload traces accordingly.

4.4 Workload Model and Workloads

This section describes various workloads and the job distributions for each workload type. The goal of evaluating a system is often to compare individual system designs or implementations. Evaluating different systems is expected to bring out performance differences that will allow for an educated decision regarding the system's design choices. However, performance differences could also be an artifact of the evaluation methodology. The performance of a system is not only a function of the system design and implementation. It may also be affected by the workload to which the system is subjected.

Many factors characterize the workloads, including job sizes, run times, and arrival patterns. The drawback of using trace workloads from real systems is that the traces reflects a specific workload, and there is always the possibility of the results that do not generalize to other systems or load conditions. In particular, there are cases where the workload depends on the system configuration; therefore, a given workload is not necessarily representative of workloads on systems with other configurations. This makes the comparison of different configurations problematic. The existing RL schedulers evaluate workloads without changing the system configurations for individual workload types.

The datacenter simulator consists of a cluster with different resource types. Jobs arrive at the cluster in an online manner in discrete timesteps. We assume that the resource demand of each job is known upon arrival; i.e., the resource requirements of each job j are given by the vector $r_j = (r_{j,1}, r_{j,2})$, and T_j is the duration of the job. We assume each job has a fixed allocation (no malleability), such that r_j must be allocated continuously from when the job starts execution until completion. If a job gets suspended, then the job's remaining run time is updated when the job resumes.

4.4.1 Synthetic workload

We used a synthetic workload where each job consists of meta-data, including job-id, resource requirement (#cpu, #gpu), and job duration. Jobs arrive online according to a *Poisson process*. The average job arrival rate, λ, determines the average load on the cluster. We chose the job duration and resource requests such that 70% of the jobs are short jobs with a duration between $1t$ and $10t$ chosen uniformly. The remaining are long-duration jobs chosen uniformly from $10t$ to $30t$ for a time horizon of 48. Each job can request a maximum of 50% of the total resources, picked randomly. The advantage of synthetic datasets is that we can control the job characteristics to study the sensitivity of various

RL methods for various job mixes.

4.4.2 Real workload

We train and evaluate the DRL scheduler using real High Performance Computing (HPC) workloads including Argonne National Laboratory (ANL) Intrepid [100], San Diego Supercomputer Center (SDSC) SP2 [101], San Diego Supercomputer Center (SDSC) Blue Horizon [102], Potsdam Institute for Climate Impact Research (PIK) IBM iDataPlex Cluster [103] and HPC2N Seth [104]. Additional details such as jobs' size, runtime, and system details can be found in the respective citations provided. These workloads were collected between 1998 and 2012. Although some of these logs are old, they have similar characteristics to modern workloads regarding job arrival rates, resource requirements, and duration. We made additional changes to the job logs to compensate for missing information. For example, we added GPU requirements to the job requests because Intrepid job logs did not have GPU jobs. Similarly, we augmented QoS data to each job where 60% of the jobs are of medium to high QoS value (0.6 to 1.0), and the rest are of low QoS value (0.1 to 0.6).

Each job, j_i in J, has associated meta-data described in Table 4.1. Jobs can have additional information, e.g., input/output data. The additional meta-data may affect a job's completion time, but we do not account for additional overhead in our environment. Additional overheads can be added as a slack time to the job's expected finish time.

Both synthetic and real workloads have multimodal distributions for job duration and resource requirements. For example, in a synthetic workload, a job can request different quantities of resources, 4 CPUs and 2 GPUs, for 10-time units. In real workloads, if a job requests more resources than the resource pool size currently simulated, the resource request is scaled down to the resource pool size. We generate a new job sequence for synthetic workloads for every training and evaluation run. For real workloads, we split the workload such that 80% is used during training and 20% during evaluation. The actual training/evaluation percentage split may vary slightly between workloads based on the length of the workloads.

Table 4.1: Per job meta-data

Field	Description
id	unique identification number
resource_req	requested number of units of each resource type
length	user-specified time duration for the job
value	monetary value when the job completes
QoS	user specified Quality of Service for the job
enter_time	job submission time
start_time	time when the job starts executing on the resource
finish_time	time when the job completes execution
qos_violation_time	upper limit when the job should finish w/o QoS violation

4.5 Power Infrastructure with Renewables

We define the power infrastructure to include the generation and distribution of power in a microgrid form. A microgrid grid integrates multiple power supplies, such as renewables, the electric grid, battery storage for renewables, and diesel generators. If the datacenter guarantees high availability (e.g., 99.99%), then relying on brown energy, including the electric grid and diesel generators, might be necessary. However, it is imperative to leverage renewable energy, if sufficiently available, rather than the electric grid or diesel generators to reduce brown energy consumption and carbon emissions.

In green datacenters, depending on the predicted power, the machines can be in any of the following power states: 1) Turned off (no power consumption, 2) Idle (low power consumption), and 3) Running full throttle (full power consumption). A basic linear power model to estimate servers' power consumption is given by,

$$P_{server(\mu)} = p_{idle} + (p_{full} - p_{idle}) \times \mu \tag{4.1}$$

where p_{idle} and p_{full} are the powers used by all machines at idle and fully utilized states, respectively. μ is the average power utilization of all machines. We denote the total renewable energy available at a datacenter at time t as E_t. This equation should hold at all times,

$$P_{server(\mu),t} \leq E_t \tag{4.2}$$

We assume that the critical infrastructure (head nodes, network, storage nodes) is always powered up to facilitate job checkpointing and migration [105]. Methods to maintain an optimal resource pool are outside the scope of this work.

4.5.1 Renewables and power availability

Forecasting is crucial for integrating variable renewable energy (VRE) resources such as wind and solar into datacenters. The difference between forecasted output and actual generation is forecast error. Factors affecting forecast performance include forecast time horizon, local weather conditions, and weather data availability. By integrating VRE forecasts into the scheduling system, datacenter operators can anticipate up- and down-ramps in VRE generation to balance load and generation in intra-day and day-ahead scheduling.

With shorter timescales, accurate VRE generation forecasting can help reduce the risk of incurring penalties. Over longer timescales, improved VRE generation forecasting based on accurate weather forecasting can help better plan long-running jobs (suspending and resuming the jobs appropriately). The forecasting accuracy decreases with the increase in the forecast time horizon. Thus, selecting a proper time horizon before designing a forecasting model is key to maintaining forecasting accuracy at an acceptable level [106].

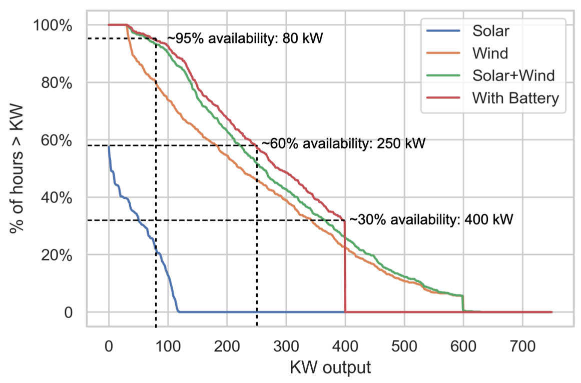
It may not be acceptable to reject or delay some jobs (e.g., jobs with high QoS requirements) if it is possible to execute them with a small additional amount of brown energy. In such cases, the datacenter faces a multi-criteria optimization problem comprising the selection of power sources and the scheduling of jobs. Similarly, batteries use also results in a multi-criteria optimization problem since the datacenter administrator can decide when to use the additional power from the battery. Multi-criteria optimization is ongoing, and we will cover this topic in future work.

We use synthetic power and real power prediction data traces in our experiments. When using synthetic power traces, the power availability level, e.g., 90%, means that 90% of the resources are turned on (10% resources turned off) during that period (see Figure 3.3). The real power prediction data (solar and wind) is from GLEAMM [107] datacenter. The

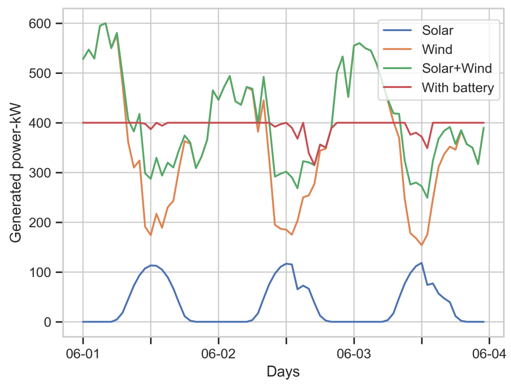
GLEAMM center is a microgrid equipped with 150 kW solar power and three wind turbines connected to the facility, each with 300 kVA of expected power generation.

4.5.2 Energy Storage Devices (ESDs)

ESDs or batteries store the excess energy from wind and solar, increasing the contribution from renewable resources and reducing the electric grid's need. This translates to reduced electricity costs, lower carbon emissions, and highly reliable services. ESDs act as a buffer to smooth out power from wind and solar farms, shifting energy from peak generation time of day (charging) to low generation periods (discharging). Various factors affect the ESD procurement decisions [108] for a given datacenter. For instance, the maximum load that the ESDs need to support, battery capacity, and charge/discharge rates of the battery. Storing MegaWatts (MW) of energy from wind and solar farms (multi MW peak) requires a large battery capacity and high battery charge/discharge rates. This can be cost and space prohibitive for small- and mid-size datacenters. Right-sizing the battery to meet a specified Quality of Service (QoS) is an important step.



(a) Available power exceeds a given value from renewables and battery.



(b) Power generation: solar (peak 120 kW), wind (peak 600 kW), June 2019.

Figure 4.2: Power generation and power availability from wind and solar at GLEAMM.

Figure 4.2a shows the probability that total expected power will exceed a given value for each renewable source and combinations from data calculated hourly across the 2019 calendar year at the GLEAMM facility. For example, a datacenter of 250 kW total electrical draw can expect to have its energy needs met at least 60% of the time entirely by the on-site solar array and wind farm. Approximately 80 kW or more will be available at least 95% of the time for critical on-site functions such as data storage, internal networking, and control functions of the computational cluster entirely from on-site renewable sources. During other times, power can be obtained from the battery or the electrical grid to ensure $\sim 95\%$ to $\sim 100\%$ access to the datacenter.

Figure 4.2b shows the power (over 3 days in June 2019) from renewables and power available with battery on site. The datacenter can expect a nearly constant power supply with the battery. We used a simple model (Algorithm 5) to right-size the battery to support a datacenter similar to GLEAMM facility with a 400 kW electrical draw. Assuming the total peak generation from renewables (solar and wind) is 720 kW, a battery with 1 MW capacity, discharge rate (E-rate) of 100, and charge rate (C-rate) of 100 can steadily supply energy to the datacenter (shown in red line, Figure 4.2b). The data shown here is for a small window (3 days in June) with ideal conditions, which drastically vary with seasons.

4.6 Simulator Validation

We validated our green datacenter simulator by running brief simulations with heuristic policies, e.g., FCFS. We collected the scheduled jobs' traces and verified that the jobs were scheduled in the specified order depending on the heuristic policy. Additionally, we repeated the experiment with different seeds (generating different job sequences) and verified that the scheduler behaved in an expected manner depending on the heuristic policy specified.

4.6.1 Validation with heuristic scheduling policies

To verify the correctness of the simulator, we simulated experiments with an easily verifiable scheduling policy, i.e., FCFS. With FCFS, the jobs that arrive first get scheduled first. We generated a set of synthetic jobs with various resource requirements, duration, and job arrival times. Once the simulation was complete, we verified that the jobs that arrived first were scheduled first. We ran similar experiments with SJF since verifying that the

Algorithm 5: Battery model: calculate total power available for datacenter use.

```
Input: renew_power - total power from renewables
Output: power_available - power available to the datacenter
Initialize: battery_capacity, max_load, c_rate, e_rate
1  for $t = 1, 2, \ldots, time\_horizon$ do
       // sufficient power from renewables, charge the battery
2      if $renew\_power \geq max\_load$ then
3          power_available = max_load
4          excess_power = renew_power - max_load
5          battery_level[t] = min(battery_level[t-1]+excess_power, battery_capacity)
6      else
           // not enough power from renewables, check the battery
7          needed_power = max_load - renew_available
8          battery_power = min(e_rate, battery_level[t-1])
           // battery has enough charge to satisfy the max load
9          if $battery\_power < needed\_power$ then
10             battery_level[t] = battery_level[t-1] - needed_power
11             power_available = max_load
12         else
               // battery cannot satisfy the full load, partial power supply
13             battery_level[t] = battery_level[t-1] - needed_power
14             power_available = renew_power + battery_power
```

shortest jobs are always scheduled first is easy. This indicates that the simulator executes the jobs in the desired order as the scheduler directs.

Next, we verified that the resources consumed by each job matched the given resource requirement of each job. We checked that when the resources were not over-subscribed. Only the jobs that could currently run on the available resources were scheduled, and the other jobs remained in the ready_pool until the resources became available. If the ready_pool was full, jobs wait in the wait_pool. Additionally, we verified that jobs from wait_pool are admitted to ready_pool only when slots are available in the ready_pool.

Also, we verified that the simulator rejects jobs that could not fit on the current cluster size. In other words, if the resource requirement of a job exceeds the max_job_resources, that job will be rejected by the simulator. Similarly, if a job's duration exceeds max_job_duration, such a job is rejected.

4.6.2 Validating action execution

To execute the schedule action, the simulator must remove a job specified by the scheduler component from ready_pool and place the job on the resources. The simulator first checks if the resources are available and then places the job on the resources. Then the simulator deducts the available resource count to reflect the new state.

The simulator first removes the candidate job from the resources to execute the suspend action. After this step, the resources previously occupied by the candidate are marked available. The suspended job is placed on the suspend_jobs list. We validated that the suspended job reflects the job_duration with the remaining time to finish (work preserving). If no candidate job was found, i.e., no running jobs, the suspend action will not incur any penalties/costs.

To validate noop action, we ran two experiments. First, we ran the simulator by controlling the job arrival rate (fewer jobs in the ready_pool). Second, we saturated the resources (no available resources) and issued noop command. In the first case, there were no jobs to execute in the ready_pool and therefore issuing a noop action resulted in no change to the system except the existing jobs (if any) progressed and time incremented. In the second case, since there were no available resources, issuing a noop action resulted in no change in the system because new jobs could not be placed on the resources. The existing jobs (if any) made progress and time increments.

Appendix-B lists the most relevant green datacenter simulator parameters that can be altered to simulate various configurations during training and evaluation. Additional parameters are excluded for brevity.

4.7 Future Work

In the future, we would like to extend our datacenter simulator to incorporate common problems faced in real-world settings. For example, our simulator does not model features like storage delays, interference, network delays, latency, or other failure rates. We can extend our datacenter model by adding extra time (e.g., to model latency) to the job's start and finish times. Similarly, storage delays can be modeled by adding additional delays based on the data size specified or requested by a job. Additionally, we will extend the datacenter model to enable additional resource selection, including memory, storage types, and network interfaces.

4.8 Chapter Summary

We proposed a unified green datacenter simulator that allows experimenting with synthetic and real workloads and integrates various renewable energy sources and Energy Storage Devices (batteries). We implemented a discrete event simulator to simulate the green datacenter environment. Our simulator aims to evaluate and compare the RL scheduler's performance under various operating conditions, i.e., power availability levels (synthetic, constant, and intermittent power supply), varying system loads, and different workloads (synthetic and real).

Chapter 5

Designing RL Scheduler for Power-Modulated Datacenters

Reinforcement Learning based job schedulers automatically learn scheduling policies from trial-and-error. For example, existing work [22] [23] [24] has shown that RL schedulers can learn effective job scheduling policies in traditional datacenter environments with constant power supply. Although the results presented in the current work are convincing, these implementations do not examine the complex dynamic green datacenter environments. Furthermore, the existing work treats the RL schedulers as black boxes without exploring the design choices that further improve performance. In this chapter, we identify limitations in the current work. We present RARE (**R**enewable energy **A**ware **RE**source management), a Deep Reinforcement Learning (DRL) job scheduler for power-modulated datacenters. We demonstrate RL scheduler features that significantly improve performance when correctly designed and configured.

5.1 Introduction and Motivation

Typically, RL researchers use standardized environments, such as OpenAI gym, to ensure fair performance comparison between various implementations. Even with standardized environments, hyperparameter tuning significantly impacts performance [109]. In the case of RL schedulers, the performance depends on two broad categories. First, RL-specific factors such as learning algorithms, DNN configurations based on problem size, optimization objectives (reward functions), and hyperparameters (batch size, learning rates, NN activation functions). Second, scheduling system-specific factors such as state representation (image only, image+vector), learning environment (traditional datacenters, power-modulated datacenters), workloads (HPC workloads, cloud workloads, machine learning workloads), and scale of the problem (small, medium, or large scale clusters). These two broach categories are interrelated, and system design decisions must consider both categories to extract expected performance. In this section, we will explore the limitations in existing work and demonstrate how we can adequately design the RL schedulers to achieve better performance in power-modulated datacenters.

5.1.1 Limitations in existing work

First, the "environment" plays a crucial role in reinforcement learning by providing suitable reinforcement (reward) and encouraging the agent to execute positive actions repetitively. The specially constrained environment rewards or penalizes the agent for correct or incorrect behavior (action). Although existing work [23] [24] [22] [98] has shown RL schedulers learn effective job scheduling policies in traditional datacenter environments (with constant power supply), they do not capture the complex dynamic green datacenters environments where the resource pool expands and contracts due to intermittent power supply from renewables. Moreover, dissimilarities in their environments (traditional vs. power-modulated datacenter environment) make it nearly impossible to compare these implementations one-to-one. Designing an RL scheduler environment with a power-modulated resource pool as part of the state space is crucial (shown in §5.4.3.2).

Second, the current work does not discuss the implications of system design choices, making it difficult to analyze why the RL schedulers perform better than heuristic policies. One such design choice is the size of the planning horizon. The RL scheduler seeks to maximize the future cumulative rewards over some predefined planning horizon. Typically, renewable energy predictions are generated for a 24-hour (day ahead) window. The RL schedulers can make better scheduling decisions with a longer planning horizon, whereas greedy heuristic policies cannot plan for future events. Therefore, studying the RL scheduler's performance over longer planning horizons is crucial for green datacenters (shown in §5.4.3.3).

Third, the current implementations, [22] [23] [24], treat the RL schedulers as a black box. Existing research does not explore RL-specific and green datacenter-specific configurations that may significantly improve performance. These configurations may include the neural network size (number of neurons in input, hidden, and output layers) and the state representation (jobs, resources, and power supply). Moreover, following a one-size-fits-all approach when evaluating the RL scheduler for different problem sizes and workloads (with different job properties and distributions) may not deliver the expected performance. Some of these design decisions have performance implications (shown in §5.4.3.4), while others may influence training time or system memory consumption (not explored in this dissertation).

5.2 Designing RL Schedulers for Green Datacenters

This section will discuss the RL scheduler design considerations that address the limitations identified in the previous section.

5.2.1 Manage resource pool based on power supply

In power-modulated datacenters, depending on the predicted power, the machines can be in any of the following power states: 1) turned off (no power consumption), 2) idle (low power consumption), or 3) running full throttle (full power consumption). The resource pool expands and contracts based on the power available to the datacenter at any given time. Power availability decides when and how many resources are turned on or off. Therefore, power prediction data is an integral part of the state space, i.e., as power availability changes, the corresponding resource availability is reflected in the state information supplied to the scheduler agent.

Given the power-modulated resource pool, the scheduler should learn scheduling policies such that jobs finish before resources are turned off, suspended, migrated, or scheduled after resources become available. The RL scheduler must be trained with resource pool expanding and contracting depending on the power supply to the datacenter (refer to Chapter-4, §4.5).

5.2.2 Extended planning horizon

Forecasting is crucial for integrating variable renewable energy (VRE) resources such as wind and solar into datacenters. By integrating VRE forecasts into the scheduling system, datacenter operators can anticipate up- and down-ramps in VRE generation to cost-effectively balance load and power generation in intra-day and day-ahead scheduling. Factors affecting forecast performance include forecast time horizon, local weather conditions, and weather data availability.

At short timescales, accurate VRE generation forecasting can help reduce the risk of incurring penalties (penalties for suspending and migrating jobs). Given the advancements in the quality and duration of weather prediction, the power generation prediction window could be 36, 48, or more hours into the future. Over longer timescales, improved VRE generation forecasting based on accurate weather forecasting [106] can help better plan long-running jobs without suspending and resuming the jobs frequently. Therefore, the system must be designed to incorporate longer planning horizons.

5.2.3 Workloads and scaling

The goal of evaluating a system is often to compare different system designs or implementations. Many factors characterize the workloads, including job sizes, resource requirements, runtime estimates, and arrival patterns. The drawback of using trace workloads from real systems is that the traces reflects a specific workload, and there is always the question of whether the results generalize to other systems or load conditions. In particular, there are cases where the workload depends on the system configuration; therefore, a given workload is not necessarily representative of workloads on systems with other configurations. This makes the comparison of different configurations problematic. The existing RL schedulers [24] analyze diverse workloads without changing the system configurations for different workloads.

5.3 RL Scheduler Agent Training

We train our scheduling agent with a custom variant of the model-free off-policy actor-critic framework with discrete actions [110] [111]. The agent interacts with the environment, sampling actions from π_θ in state s, transitioning to a new state s' and receiving a reward r. This experience is saved in a replay buffer $\mathcal{D}$ for later use. In addition to the "actor-network", we initialize a neural network ϕ to represent the Q-function, denoted Q_ϕ, which takes state and action vectors as input and outputs an estimate of the expected return when taking action a in state s and following π thereafter. We can use our critic network to train the actor network to output higher-value actions. The improved actor is then used to improve the critic network's value estimates, and this process is repeated until performance converges. This technique is "model-free". It does not attempt to directly model changes in the environment and "off-policy" because it recycles data collected from past decisions of the actor network.

The datacenter model (Chapter-3, section 3.2.2) with ready_pool size n is converted to an RL environment that takes an action index $< n + 2$ and returns a new state and reward. States are tuples containing both the resource image and array of job meta-data (section 3.2.1), while the reward function can be adjusted to reflect the goals of our scheduling system. This dissertation focuses on optimizing the total (monetary) value of completed jobs; the reward at timestep t is the total value of all completed jobs at that timestep. Our agent learns to select jobs from the ready_pool that maximize total job value with the help of three DNNs. The *encoder* combines the state information in the resource allocation image and job metadata array and produces a compact vector representation. The resource allocation image is processed by convolutional layers common in computer vision applications, while the job array is passed through standard feed-forward layers.

The two representations are then normalized for stability and concatenated together before a final sequence of layers condenses them to a vector $\tilde{s} \in \mathbb{R}^{128}$ that summarizes the current state of the scheduling environment. The *actor* network takes $\tilde{s}$ as input and outputs probabilities for selecting all $n+2$ scheduling actions. The *critic* also takes $\tilde{s}$ as input and outputs $q \in \mathbb{R}^{n+2}$, where $q[i]$ is an estimate of the total monetary value that we expect to achieve in the future when beginning in the current state and taking the ith action.

Algorithm 6: RARE: DRL Scheduler Training Process

```
Initialize: Encoder Net $g_\psi$, Actor Net $\pi_\theta$, Critic Net $Q_\phi$
Input: Advantage Samples $k$, Replay Buffer with pre-provided transitions
        $\mathcal{D} \leftarrow \{(s_i, a_i, r_i, s'_i), \dots\}$
1 for training step $t \in \{0, \dots, T\}$ do
2     Randomly Sample Batch of $B$ transitions $\{(s_i, a_i, r_i, s'_i)\}_{i=0}^{i=B} \sim \mathcal{D}$
      // the encoder embeds resource image + job metadata into a single array
3     Let $\tilde{s_j} := g_\psi(s_j)$
      // critic loss (where $\not\nabla$ cancels gradient contributions)
4     $\mathcal{L}_{critic} \leftarrow \frac{1}{B}\sum_{i=0}^{i=B}\left(\left(Q_\phi(\tilde{s}_i, a_i) - \mathop{\mathbb{E}}_{a' \sim \pi_\theta(\tilde{s}'_i)}\left[(r_i + \not\nabla\gamma(Q_\phi(\tilde{s}'_i, a')\right]\right)^2\right)$
      // estimate the advantage function, $A(s, a)$, by comparing the value of $a$
         to the average value of actions sampled from the policy in a given
         state.
5     Let $\hat{A}(\tilde{s}_i, a_i) := Q_\phi(\tilde{s}_i, a_i) - \frac{1}{k}\Sigma_0^k Q_\phi(\tilde{s}_i, a' \sim \pi_\theta(\tilde{s}_i))$
      // offline actor loss [112]. supervised regression copies actions with
         positive advantage (where $\mathbb{1}_{\{x\}}$ is 1 if $x$ is $True$ else 0)
6     $\mathcal{L}_{actor} \leftarrow \frac{1}{B}\sum_{i=0}^{i=B}\left(-\mathbb{1}_{\{\hat{A}(\tilde{s}_i, a_i)>0\}}\log\pi_\theta(a_i|\tilde{s}_i))\right)$
      // update neural nets by gradient descent
7     $\psi \leftarrow \psi - \alpha\nabla_\psi\mathcal{L}_{critic}$ , $\phi \leftarrow \phi - \alpha\nabla\phi\mathcal{L}_{critic}$ , $\theta \leftarrow \theta - \alpha\nabla_\theta\mathcal{L}_{actor}$
8 end
Output: Trained Scheduling Policy $\pi_\theta(g_\psi(s))$
```

In an actor-critic method, the actor is trained to assign higher probabilities to actions that the critic determines will lead to higher monetary value. The critic then uses the improved actor to better estimate the expected return of selecting each action. Both networks rely on the state representation $\tilde{s}$ learned by the encoder network. Separating the encoder in this way lets us share parameters across the actor and critic training processes and reduces overall network size. However, the encoder parameters are updated alongside the critic but not the actor for stability reasons.

The training process is outlined in Algorithm 6. Our specific implementation includes several additional details that have been shown to improve stability and performance; the offline version of our scheduler is closest to Critic Regularized Regression (CRR) [112].

5.4 Evaluation

This section evaluates the RL scheduler's performance with different workloads and power availability at the green datacenter and explores the effects of design choices on the performance. Before presenting the results, we briefly discuss the workload and experimental setup.

5.4.1 Experimentation conditions

Our green datacenter simulator compares different resource allocation and scheduling policies using various workloads and power availability settings. Our datacenter model (section 3.2.1) integrates resources, jobs, and power supply from renewables (section 4.5.1) and ESDs (section 4.5.2). The flexible design of our datacenter simulator allows for exploring various design options that can improve the RL scheduler's performance. We modeled a small-scale (10 to 50 resources) and a medium-scale datacenter (100 to 1200 resources) for the following experiments. Scheduler agent training and evaluation experiments were conducted on servers with Intel(R) Xeon(R) CPU E5-2620 v4 2.10GHz and AMD EPYC 7252 8-Core processors, and NVIDIA GTX1080Ti (12GB VRAM), NVIDIA TESLA P100 (12GB VRAM), NVIDIA A100 (40GB VRAM) GPUs, NVIDIA RTX6000 (24GB VRAM) GPUs, NVIDIA A16 (16GB VRAM) GPUs, and NVIDIA A40 (48GB VRAM) GPUs. See Computing Resources, Dept. of Computer Science [113] for more information about the compute resources.

5.4.2 Evaluation metrics

The metric used for evaluating the RL scheduler's performance is the *Total Job Value* (section 3.2.3) from running the jobs. The *Total Job Value* accumulated during evaluation includes a total value for all the jobs completed on time. The higher the *Total Job Value*, the better. We repeated each experiment 10 times with random seeds and found the error margin between runs was insignificant. In the following graphs, the Total Job Value (y-axis), the total value of all the completed jobs, is expressed in scientific notation (e notation).

For comparison, we evaluate heuristic scheduling policies, including Shortest Job First (SJF), Quality of Service (QoS), Highest Value First (HVF), and First Come First Serve (FCFS) for comparison. The SJF heuristic policy picks the job with the shortest runtime first. With the QoS scheduling policy, the job with the highest QoS value (refer to section 3.2.1.2) is scheduled first. The highest value job is scheduled first with the HVF policy, and the job with the earliest enter_time is scheduled using FCFS. We selected the representative heuristics (SJF, FCFS, QoS, and HVF) because each heuristic optimizes one or more metrics (job value, utilization, job completion). The SJF heuristic minimizes job delays (and maximizes utilization), the FCFS heuristic (commonly used policy in SLURM [114]) minimizes wait times, QoS greedy heuristic minimizes job delays and QoS violations, and HVF greedy heuristic maximizes value. We will use these heuristic scheduling policies for comparison in Chapter-6 and Chapter-7 as well.

We explored a random scheduling policy that performed similarly (or better) for 10 and 20 resources. For larger problem sizes, the random job scheduling policy performed significantly lower (sometimes negative value due to QoS violations) than all the other heuristic policies. Therefore, the random policy is disregarded from further comparisons. Our framework does not support backfilling; we will incorporate this feature in our future work.

For additional details on synthetic and real workloads, refer to Chapter-3.

5.4.2.1 Power availability

We use synthetic power and real power prediction data traces in our experiments. When using synthetic power traces, the power availability level, e.g., 90%, means that 90% of the resources are turned on (10% resources turned off) for that time step (see Figure 3.3).

The real power prediction data (solar and wind) is from GLEAMM [107] datacenter (see Chapter-4, §4.5.1).

5.4.3 Results

First, we evaluate our RL scheduler with synthetic and ANL HPC workloads. Second, we demonstrate the RL scheduler's adaptability to the intermittent power supply. Third, we evaluate design choices, namely extended planning horizon and increasing ready_pool size, that significantly increase the performance of the RL scheduler compared to heuristics. Finally, we show that the RL scheduler can learn to imitate the existing heuristic policies and improve performance over those heuristic policies.

5.4.3.1 Performance with synthetic and HPC workloads

Performance with synthetic workload and power data: This section demonstrates the RL scheduler's performance with the increasing number of resources. We modeled a small and medium scale datacenter and maintained the workload and power availability at 100%. The *Total Job Value* obtained compared to heuristic scheduling policies is plotted in Figure 5.1.

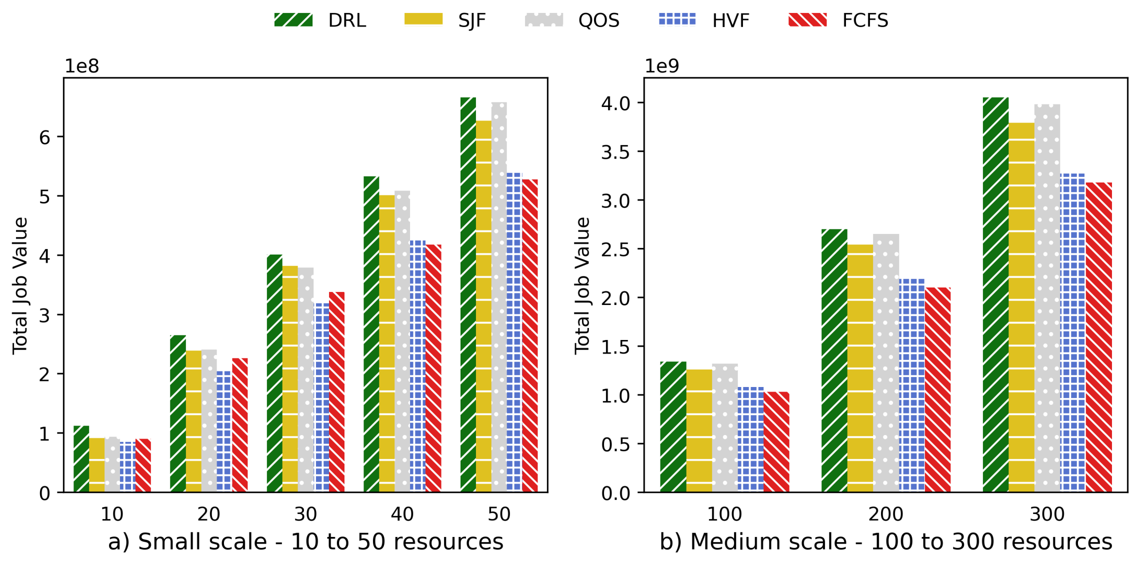


Figure 5.1: RL scheduler's performance vs. heuristic scheduling policies with varying datacenter capacity (small to medium scale datacenter).

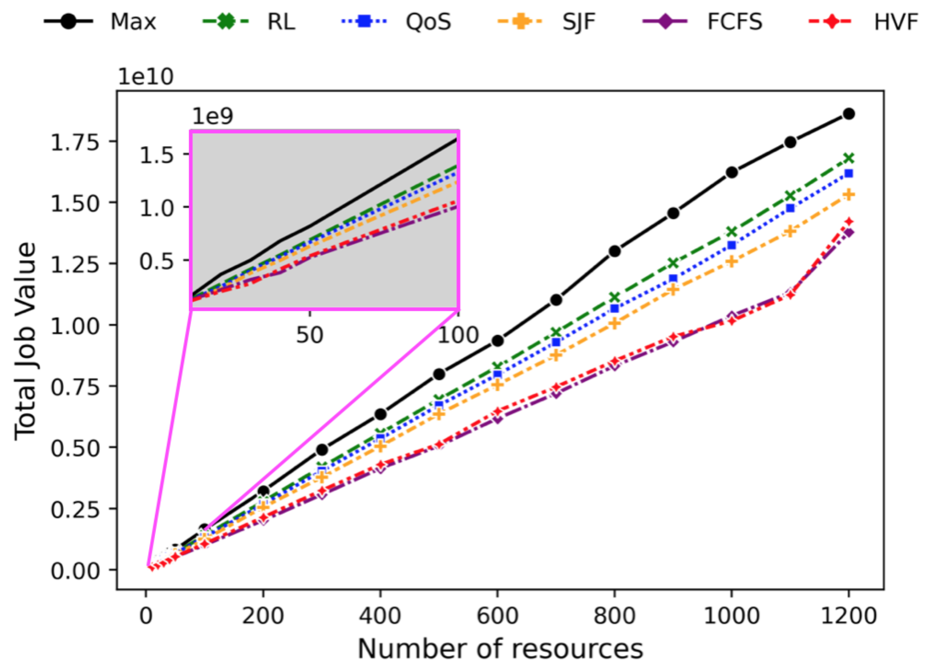


Figure 5.2: Performance - RL and heuristic policies upto 1200 resources.

Analysis: From Figure 5.1, the RL scheduler performs 18% to 25% better for small-scale datacenter and 2% to 20% better for medium-scale datacenters compared to heuristic policies. As the number of resources increases (≥ 50), the RL scheduler's performance closely matches (2% to 6% better) the performance of QoS and SJF policies. Figure 5.2 shows that the RL scheduler performs consistently better than heuristic policies as the problem size increases up to 1200 resources. We note that as the state space increases with bigger problem size, the RL scheduler must explore more states to decide on the best action in any given step. Given the vast state space, the agent cannot explore all possible state-action pairs within the fixed episodic limits. Therefore, the performance gap between the RL scheduler and maximum value becomes wider as we scale to a higher number of resources (shown in Figure 5.2). This vast state space problem can be alleviated by splitting the state space into smaller sizes. We will investigate this approach in the future.

Performance with real HPC workloads and real power prediction data: We modeled a small-scale datacenter (10 to 30 resources) with ANL HPC job workload and maintained the job arrival rate at 100%. We also used the power prediction data from the GLEAMM datacenter to simulate a real-world green datacenter powered by renewables and batteries (no brown energy).

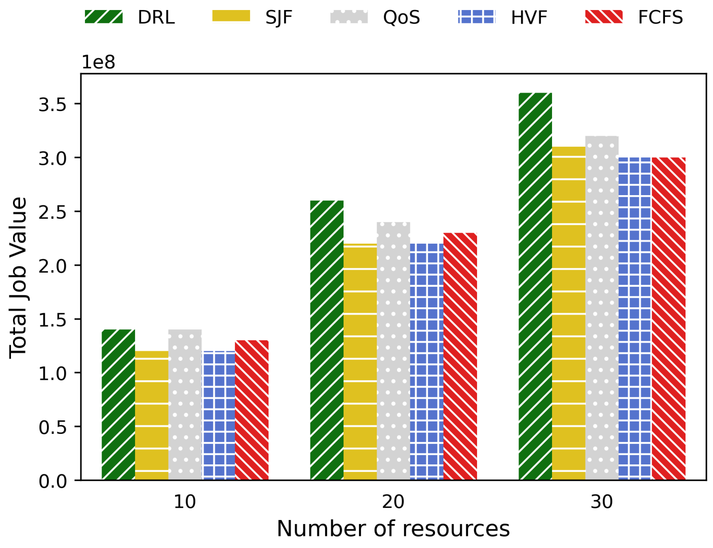


Figure 5.3: RL scheduler's performance vs. heuristic scheduling policies with ANL workload and GLEAMM power data.

Analysis: Figure 5.3 shows the performance of the RL scheduler compared to heuristic policies on the ANL workloads and real power prediction data from GLEAMM. The RL scheduler's performance for ten resources matches the QoS and is 5% to 10% better than other scheduling policies. For 20 and 30 resources, the RL scheduler performs 7% to 14% better than the heuristic policies.

Different workloads have diverse job mixes and distributions; therefore, their performance varies [24]. Although the one-size-fits-all approach works, we plan to investigate the diverse workload properties further to gain deeper insights into designing RL schedulers (e.g., DNN shape, size, and state representation).

5.4.3.2 Scheduler's adaptability to intermittent power supply

This section presents the RL scheduler's adaptability to the varying power supply. The intermittent power generation by renewables necessitates the datacenter resources to switch between power states (off, idle, full throttle). Our experiments simulate intermittent power supply to the datacenter at each time step, not fixed reduced power supply. We modeled small and medium-scale datacenter with different power availability levels and measured the total job value obtained at each level. Based on power availability, the resource pool size expands and contracts at every timestep (Chapter-3, Figure 3.3). The job arrival rate is constant at 100% for all the power availability levels.

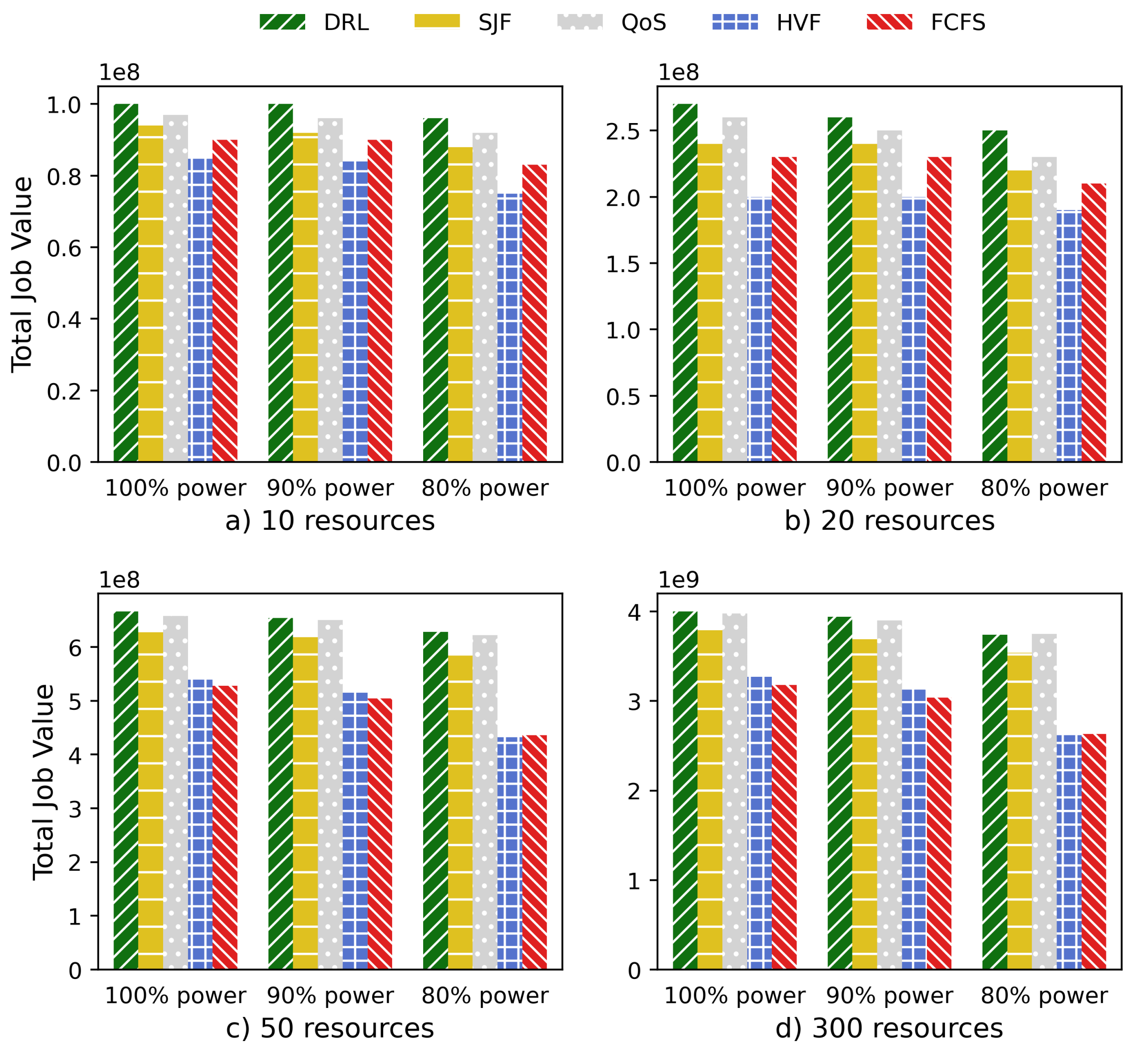


Figure 5.4: RL Scheduler's performance with varying power supply - small and medium scale datacenter.

Analysis: In Figure 5.4, we plotted the total job value with varying power supply $(100\%, 90\%$ and $80\%)$ for a small and medium-scale cluster. For small-scale cluster (Figure 5.4a, b and c), the RL scheduler performs 9% to 13% better (10 and 20 resources) and 8% to 12% better (50 resources) than heuristic policies. For medium-scale cluster (Figure 5.4d), the RL scheduler performs 1% better than QoS policy and 5% to 20% better than other heuristic policies. The greedy heuristic policies, like SJF, do not plan for the future by design. On the contrary, the RL scheduler observes the resource availability changes in the future and intelligently schedules suitable jobs maximizing total job value. As observed in the previous section, the performance difference between the RL scheduler and heuristic policies narrows (300 resources) with a larger state space.

5.4.3.3 Extended planning horizon

The renewable energy predictions are typically generated for a 24−*hour* (day ahead) window. More recently, researchers have developed better prediction models that can predict (with relative accuracy) power generation for extended time windows (2-3 days) [115]. This subsection investigates the RL scheduler's performance with various planning horizons, namely 36, 48, 60, and 72-time units. For this experiment, we used synthetic workload and 100% power to isolate the performance implications of the extended planning horizon.

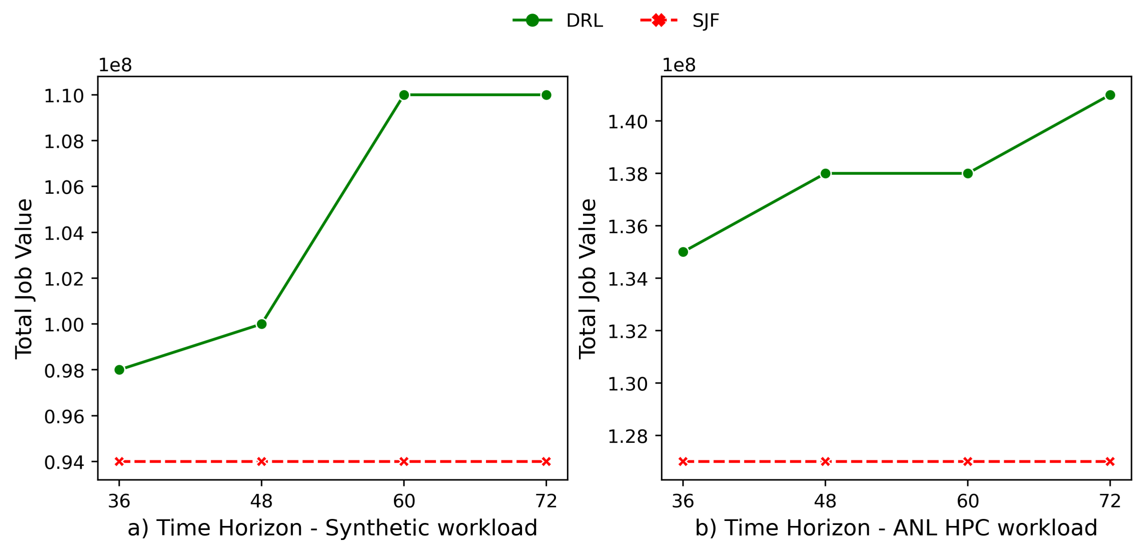


Figure 5.5: RL scheduler's performance vs. SJF scheduling policy with increasing time horizon - 10 resources.

Analysis: From Figure 5.5, as the planning horizon increases from 36 to 72, the RL scheduler performs 4% to 14% and 6% to 10% better than SJF heuristic policy for synthetic and ANL HPC workloads, respectively. The RL scheduler seeks to maximize the future cumulative rewards over some predefined planning horizon (a.k.a, time horizon). With a shorter planning horizon, TH=36, the RL scheduler might be limited to *myopic* decisions yielding immediate gains. The greedy heuristic policies lack the ability to plan for future events; specifically, the performance of SJF policy cannot improve as long as the jobs' runtimes are strictly less than the planning horizon.

Our experiments assume that the quality of predictive information does not decay with an extended time horizon. In reality, as the time horizon increases, uncertainty increases due to weather prediction inaccuracy (described in section 4.5.1). This uncertainty can be captured by changing the $discount\ factor, \gamma$. The discount factor determines how much the RL agent cares about rewards in the distant future relative to those in the immediate future. If $\gamma = 0$, the agent will be completely myopic and only learn about actions that produce an immediate reward. For our experiments above, we set $\gamma = 0.99$. We note that optimization problems become computationally intensive (due to state-space explosion) with longer time horizons. In the future, we will identify the limits beyond which extending the time horizon will result in diminishing returns.

5.4.3.4 Varying readypool size

This section evaluates the performance of different scheduling policies as the ready_pool size varies. The size of the ready_pool (Chapter-3, Figure 3.3) is fixed for any given problem size because the DNN's shape cannot change dynamically during training or evaluation. The RL scheduler can only select one or more jobs in the ready_pool at each step. On the other hand, typical heuristic schedulers can select one or more jobs from all of the waiting jobs in the system. This experiment demonstrates that capping the list of jobs to a reasonable number (ready_pool size) does not affect the performance of the RL scheduler. For this experiment, we used ten resources, synthetic and ANL workloads with 100% power supply, to study the effect of ready_pool size on the quality of the results produced by the RL scheduler.

Analysis: Figure 5.6a shows the RL scheduler's performance (synthetic workload) compared to the SJF scheduling policy with varying ready_pool sizes. The RL scheduler performs best with a ready_pool size of 15, an 18% improvement over SJF. The RL scheduler's

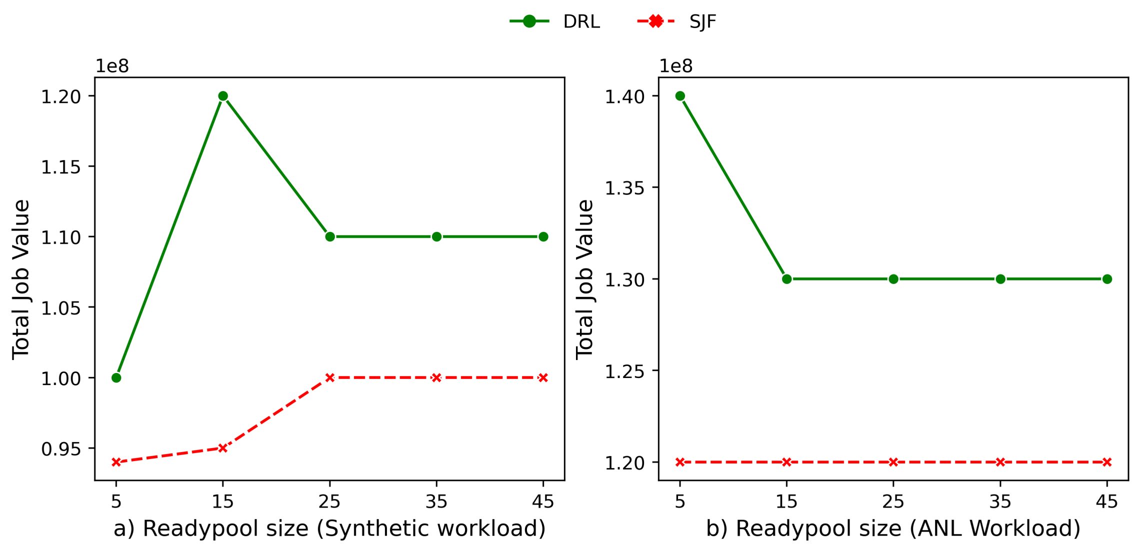


Figure 5.6: RL scheduler's performance vs. SJF scheduling policy with varying *ready_pool* sizes - 10 resources.

performance decreases for ready_pool sizes of 25 and higher but still performs 4% to 7% better than SJF. On the other hand, the SJF policy always picks the smallest job, and the performance stays constant even when we increase the ready_pool size because the jobs' lengths are within a certain distribution (described in section 4.4.1). Even if more jobs are visible (in the ready_pool) to the SJF scheduler, the job lengths are likely to be similar.

Figure 5.6b shows the RL scheduler's performance with ANL HPC workload. The RL scheduler performs 10% better than SJF when the ready_pool size is 5 and 7% better for ready_pool size 15 and above. Whereas, with the synthetic workload, the RL scheduler's performance increases, with the ANL HPC workload, the performance decreases as ready_pool size increases. We showed that having a smaller set of ready_pool jobs does not affect the RL scheduler's overall performance. Further, we believe that the graph trends for the two workloads are different due to the differences in the job distributions. We plan to investigate further with other workloads.

5.5 Future Work

We showed that the RL scheduler's performance linearly scales up to 1200 resources. This was the largest problem size we could train on our existing hardware. In the future, on better hardware, we would like to identify the largest problem size that can be accommodated without compromising on the RL scheduler's performance. The RL scheduler's performance will diminish as the state space increases. At that point, we will explore other solutions, including a multi-agent approach that partitions the state space among multiple scheduling agents might provide better results.

We explored two representations, 1) image-only representation where resources and jobs were represented as an image, and 2) image with job vector where resources were represented as an image and jobs were represented as vectors. We want to explore another representation where resources and jobs are represented as vectors only.

Finally, comparing one state-of-the-art RL scheduler implementation with other is extremely difficult due to the reasons discussed in the introduction section of this chapter. We want to open source and extend our work so that the practitioners have a basic set of design features but also be able to customize the datacenter environment (types of workloads, cluster configurations, and other features) to implement their unique solutions and provide a fair comparison with other state-of-the-art schedulers implemented using our basic environment.

5.6 Chapter Summary

In this chapter, we surveyed existing research and determined that the current research does not adequately address challenges presented in the complex dynamic green datacenter environments. The current research presents the RL schedulers as black boxes without exploring the system design configurations. We identified four RL scheduler design features pertinent to green datacenters, namely 1) state and action space representation, 2) configuring for different workloads, 3) extended planning horizon, and 4) policy network configurations.

We presented our DRL scheduler framework, RARE (**R**enewable energy **A**ware **RE**source management). We experimentally demonstrated performance improvements when the RL scheduler is appropriately designed and configured. We show that our RL scheduler performs better than heuristics policies in the dynamic green datacenter environment for synthetic and real HPC workloads for a small to medium-scale cluster with 10 to 1200 resources. The RL scheduler adapts exceptionally well to the intermittent power supply (synthetic and actual power prediction data). With synthetic workload, our RL scheduler performs 18% to 25% better for small-scale clusters and 2% to 20% better for medium-scale clusters than heuristic policies. With the HPC workload, the RL scheduler performs 7% to 14% better than the heuristic policies. With varying power supply (100%, 90% and 80% power), our RL scheduler performs 9% to 13% better in small scale cluster and 5% to 20% better compared to heuristic policies in medium scale cluster. We show that as the planning horizon extends (from 36 to 72-time units), our RL scheduler performs 4% to 14% and 6% to 10% better than heuristic policy for synthetic and HPC workloads, respectively.

Chapter 6

Constraint Controlled Reinforcement Learning Scheduler

In power-modulated datacenters, resource availability depends on the power supply from renewables. Intermittent power supply from renewables leads to intermittent resource availability, inducing job delays (and associated costs). Green datacenter operators must intelligently manage their workloads and available power supply to extract maximum benefits while constraining the costs. Then, the scheduler's objective is to schedule jobs on a set of resources to maximize the total value (revenue) while minimizing the overall costs (e.g., due to delayed jobs or QoS violations). Hence, the aims of achieving high rewards and low costs are in opposition. A trade-off exists between achieving high job value on the one hand and low expected delays on the other. In addition, datacenter operators often prioritize multiple objectives, including high system utilization and job completion. To accomplish the opposing goals of maximizing total job value, minimizing job delays, and optimizing for multiple objectives, we apply the Proportional-Integral-Derivative (PID) Lagrangian methods in Deep Reinforcement Learning to job scheduling problem in the green datacenter environment. Lagrangian methods are widely used algorithms for constrained optimization problems. We adopt a controls perspective to learn the Lagrange multiplier with proportional, integral, and derivative control, achieving favorable learning dynamics. Feedback control defines cost terms for the learning agent, monitors the cost limits during training, and continuously adjusts the learning parameters to achieve stable performance. Our experiments demonstrate improved performance compared to scheduling policies without the PID Lagrangian methods. Experimental results illustrate the effectiveness of the **Co**nstraint **Co**ntrolled **R**einforcement **L**earning (**CoCoRL**) scheduler that simultaneously satisfies multiple objectives.

6.1 Introduction and Motivation

Existing RL schedulers presented in [22] [116] [23] solely focus on optimizing a single reward. While these implementations demonstrate good performance for a specific metric, they do not consider costs associated with hazardous actions. In practice, these RL policies face skepticism about their robustness when encountering unusual situations. For example, an RL model may "misbehave" on an unanticipated workload change or intermittent power supply making bad scheduling decisions that lead to applications' service level objective (SLO) violations.

While errors during training in these domains (robotic locomotion [17] [18], sophisticated video games [19] [20], congestion control [21], and cluster job scheduling [24] [117] [118]) come without a cost, limiting the rates of hazardous outcomes in some learning scenarios is crucial. One example is wear and tear on a robot's components or surroundings. While it is possible to impose such limits directly by prescribing constraints in the action or state space, hazard-avoiding behavior must be learned.

Additionally, focusing only on generating "good" or "balanced" schedules is not enough. The agent must not only satisfy the constraints but also optimize for other metrics. The two most common objectives that datacenter operators prioritize are order-based and resource-based [119]. For example, satisfying due dates (reducing job delays) is an order-based objective, while efficient resource utilization is a resource-based objective. This can be achieved by appropriately co-designing the reward and cost functions to satisfy multiple objectives simultaneously.

Furthermore, any effective scheduling system must adapt to unforeseen events. In a power-modulated datacenter environment, the resource availability is based on the power supply from renewables. Even with the best weather predictions, there can be unforeseen circumstances when the power supply changes drastically. The scheduler should handle such situations gracefully by maintaining high rewards and costs within limits.

One solution is to embed all conflicting requirements in a constrained RL problem and use a primal-dual algorithm that automatically chooses the agent's parameters. The main advantage of this approach is that constraints ensure satisfying behavior without manually selecting the penalty coefficients. Instead of applying constraints in the action or state space directly, hazard-avoiding behavior is learned. We utilize the well-known framework of the Constrained Markov Decision Process (CMDP) [120], limiting the accumulation of the cost signal, which is similar to the reward signal. Lagrangian methods are a classic technique for solving constrained optimization problems. The desired scheduling policy is one that maximizes the usual reward while satisfying the cost constraint.

This chapter presents a constraint-controlled RL scheduler that uses a primal-dual algorithm that automatically learns conflicting reward and cost functions. We demonstrate that a Constraint-controlled RL scheduler learns policies that satisfy the constraints and optimize for other objectives, such as resource utilization and job completion. Our results illustrate that our CoCoRL scheduler efficiently adapts to real HPC workloads while using real power supply data (solar and wind) from an existing Green datacenter. We illustrate the importance of accurately tuning hyperparameters to satisfy various optimization goals set by datacenter operators.

6.2 Background

In many situations in the optimization of dynamic systems, a single utility for the optimizer may not be sufficient to describe all the objectives involved in sequential decision-making. A special situation is where one controller has multiple objectives. Instead of introducing a single utility to maximize (or a cost minimize), that may be some function (e.g., weighted sum) of the multiple objectives, a natural approach for handling such cases is optimizing one objective with constraints on others. In particular, this allows us to understand the trade-off between the various objectives.

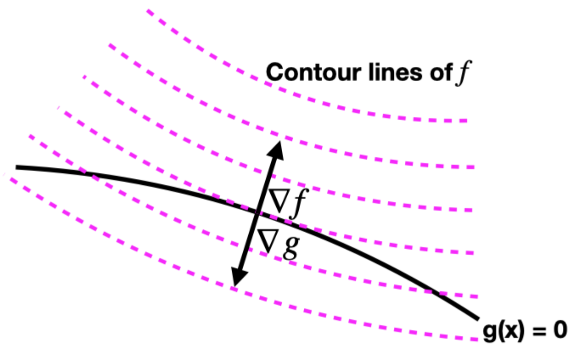


Figure 6.1: The constrained minimum, ∇f = -$\lambda \nabla g$.

One solution is to embed all conflicting requirements in a constrained RL problem and use a primal-dual algorithm that automatically chooses the agent's parameters. This approach's main advantage is that constraints ensure satisfying behavior without manually selecting the penalty coefficients. The Constrained Markov Decision Process (CMDP) [120] framework facilitates limiting the accumulation of the cost signal, which is similar to the reward signal (Figure 6.1) where the optimal scheduling policy is one that maximizes the usual return while satisfying the cost constraint.

6.2.1 Lagrangian methods for constrained optimization

Lagrangian methods are a classic family of approaches to solving constrained optimization problems. For example, the equality-constrained problem over the real vector x:

$$\min_{\mathrm{x}} f(\mathrm{x}) \quad \text{s.t. } g(f) = 0 \tag{6.1}$$

is transformed into an unconstrained one by the introduction of a dual variable–the Lagrange multiplier, λ–to form the Lagrangian: $\mathcal{L}(\mathrm{x}, \lambda) = f(\mathrm{x}) + \lambda g(\mathrm{x})$, which is used to find the solution as:

$$(\mathrm{x}^*, \lambda^*) = \arg\max_{\lambda} \min_{x} \mathcal{L}(\mathrm{x}, \lambda) \tag{6.2}$$

Gradient-based algorithms iteratively update the primal and dual variables where λ acts as a learned penalty coefficient in the objective, leading to a constraint-satisfying solution [121].

6.2.2 Dynamical systems and feedback control

Dynamical systems are processes that are subject to external control. A generic formulation for discrete-time systems with feedback control is:

$$\begin{aligned} \mathrm{x}_{k+1} &= F(\mathrm{x}_k, u_k) \\ \mathrm{y}_k &= Z(\mathrm{x}_k) \\ u_k &= h(\mathrm{y}_0, \mathrm{y}_1, \ldots, \mathrm{y}_k) \end{aligned} \tag{6.3}$$

Where x is the state vector, F is the dynamics function, u applied control, y is the measurement outputs, and the subscript denotes the time step. The feedback rule, h, can access past and present measurements. The optimal control problem is to design a control rule, h, that results in a sequence $\mathrm{y}_{0:T} = \{\mathrm{y}_0, \mathrm{y}_1, \ldots, \mathrm{y}_T\}$ (or states $\mathrm{x}_{0:T}$) that scores well for some cost function C. For instance, reaching a goal condition, $C = |\mathrm{y}_T - \bar{\mathrm{y}}|$, or closely following a desired trajectory, $\bar{\mathrm{y}}_{0:T}$.

A typical feedback control system, illustrated in Figure 6.2, comprises a controller, a system to be controlled, actuators, and sensors. The setpoint represents the exact value of the controlled variable. The error is the difference between the setpoint and the current value of the controlled variable, i.e., $e(t)$ = *setpoint* – *current value* of the controlled variable. The manipulated variable is the quantity that the controller varies to influence the value of the controlled variable. The feedback loop of the system is as follows: 1) The system, at regular intervals, monitors and compares the controlled variable to the setpoint to determine the error, 2) The controller computes the required control signal based on the error; and 3) The actuators change the value of the manipulated variable to control the system.

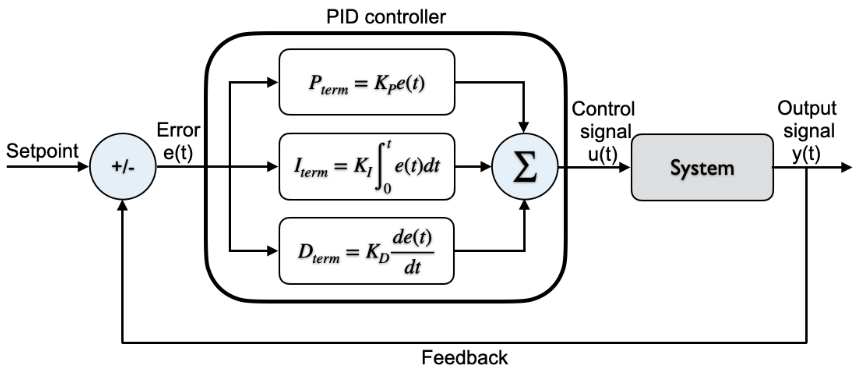


Figure 6.2: Typical PID feedback loop to control the system.

6.2.3 Constrained-controlled reinforcement learning

The Constrained Markov Decision Processes (CMDP) [120] extend MDPs [122] to include constraints into RL. A CMDP is the extended tuple $(S, A, R, T, \mu, C_0, C_1, \ldots, d_0, d_1, \ldots)$. The cost functions $C_i : S \times A \times S \to \mathbb{R}$ are defined with the same form as the reward functions, and $d_i : \mathbb{R}$ represents cost limits. For this work, we consider a single, all-encompassing cost.

In RL, the expected sum of discounted rewards computed over $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots)$ trajectories, using the policy $\pi(a|s)$ is a common performance objective, defined as $J(\pi) = E_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})]$. The corresponding value function for the cost is defined as $J_C(\pi) = E_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t C(s_t, a_t, s_{t+1})]$. Then the constrained-controlled RL problem is solved for the best possible policy:

$$\pi^* = \arg\max_{\pi} J(\pi) \quad \text{s.t. } J_C(\pi) \le d \tag{6.4}$$

DRL uses a Deep Neural Network (DNN) for the policy, $\pi_\theta = \pi(\cdot|s;\theta)$ with θ as a parameter vector. The policy gradient algorithms improve the policy over time by gathering experience in the task of estimating the reward objective gradient, $\nabla_\theta J(\pi_\theta)$ iteratively. Therefore our constrained optimization problem is expressed as maximizing score at some iterate, π_k, ideally obeying constraints at each iteration:

$$\begin{gathered}\max_{\pi} J(\pi_k) \quad \text{s.t. } J_C(\pi_m) \leq d \\ \text{where } m \in \{0, 1, \ldots, k\}\end{gathered} \tag{6.5}$$

We cast constrained RL as a dynamical system with the Lagrange multiplier as a control input, to which we apply PID control in the learning algorithm. The PID multiplier method proposed in [123] is a recent result where a PID update rule is considered for a learned Lagrange multiplier.

6.3 Related Work

This section briefly discusses relevant related work applicable to this chapter. For a complete review of related work on heuristic and RL schedulers, refer to Chapter-3 and Chapter-4.

Feedback control-based schedulers: A wide variety of contributions using control theory based resource management in various settings are proposed in [124] [125] [126]. The scheduler in [127] applies the PID controller to load balance tasks in real-time systems. These systems use heuristic or formal methods and manually tune the control parameters to achieve the desired optimization objectives.

Single objective RL schedulers: Much of the prior work focuses on unconstrained optimization problems i.e., without safety constraint. These systems optimize a single objective such as latency, throughput, power, and energy consumption. The Spotlight [22] partitions the agent's neural network training operations onto different devices (CPUs and GPUs) to minimize ML model training time. The RL scheduler in [97] is designed to minimize the makespan of DAG jobs considering both task dependencies and heterogeneous resource demands. The scheduler in [98] implements a co-scheduling algorithm based on an adaptive RL by combining application profiling and cluster monitoring. The optimization objective in [98] is to maximize resource utilization. These implementations incorporate a single reward function that maximizes or minimizes a single objective. Although effective, this naive application of RL to optimize for a single objective can lead to poor performance on the secondary objectives. Also, none of these schedulers are designed for green datacenter environments.

Multi-objective RL schedulers: Many prior work has studied multi-objective optimization in scheduling [128]. Recent work [74] presents a unified management approach for the thermal and workload distribution in datacenters. The objective is to minimize power consumption while satisfying thermal and Quality of Service (QoS) constraints. They implement a particle-based algorithm that requires adjusting some non-trivial number of parameters over multiple iterations. DeepEE [23] proposes improving datacenters' energy efficiency by concurrently considering the job scheduling and cooling systems. The goal, in [23], is to reduce cooling costs in a datacenter rather than optimize job scheduling. The joint optimization problem aims to minimize the power usage effectiveness while preventing overheating in the rack server and keeping the load balance. The downside of these systems is that designing a good reward function that balances different, often conflicting, objectives is challenging. A different optimal solution exists for each set of penalty coefficients, also known as Pareto optimality. Manually tuning the exact coefficients of different objectives is a time-consuming process. Our work focuses on automatically learning the policy parameters based on the cost signal instead of manually tuning for conflicting objectives.

RL schedulers with constraints: A good number of prior work has studied safety constraints in RL training [129]. Two main categories of the work include limiting the action space or state space to satisfy constraints. The work in [130] blocks specific actions when a safety condition is violated. However, shielded RL only applies to simple problems with tabular states and treats the shield as part of the environment without modifying the RL optimization objective. Therefore, the shield is only a protection mechanism to constrain the action set, not a technique for adapting the agent's policy. The proposed work in [131] uses the RL framework for robotics to inject experience data from the expert's control into the replay buffer for off-policy RL methods. This framework uses a classical PID controller as an alternative to speed up the training of RL for robot planning and navigation problems. As the actor-network becomes more advanced, it can then take over to perform more complex actions. Eventually, the PID controller is discarded entirely.

Another body of prior work focuses on adjusting RL algorithms to avoid exploring unsafe states [132], modeling the risk in state transitions, and conservative exploration [133]. These methods incorporate the fallback policy within neural networks but cannot guarantee that the system will eventually recover to a safe state. Training wheels [134] prioritize meeting the safety condition and rely on deterministic fallback policies. While it may be possible to impose limits directly by prescribing constraints in the action or state space, hazard-avoiding behavior must be learned. Our work focuses on embedding all conflicting requirements in a constrained RL problem and learning the parameters automatically instead of restricting unsafe states and actions.

6.4 Approach

This section presents the mapping of RL as a dynamical system and the PID Lagrangian method for constrained-controlled RL agents. In section 6.5, we present scheduler design as an instance of this mapping and the RL training algorithm.

6.4.1 Mapping RL as a dynamical system

Similar to the system of equations in eq 6.3, the first-order dynamical system in constrained RL [123] form is defined as:

$$\begin{aligned} \theta_{k+1} &= F(\theta_k, \lambda_k) \\ y_k &= J_C(\pi_{\theta_k}) \\ \lambda_k &= h(y_0, y_1, \ldots, y_k, d) \end{aligned} \tag{6.6}$$

Here F is a nonlinear function corresponding to the policy update on the RL agent's parameter vector, θ. The penalty or cost-objective, y, is the measured output of the system. This measured output (y) is fed to the feedback control rule, h, along with the cost limit, d. With this starting point, the learning algorithm given by F, and the penalty coefficient update rule, h, can be tailored to solve the constrained optimization in (eq 6.5).

The policy gradients for reward and cost of the first-order Lagrangian method, $\nabla_\theta \mathcal{L}(\theta, \lambda) = \nabla_\theta J(\pi_\theta) - \lambda \nabla_\theta J_C(\pi_\theta)$, is organized in the form of eq 6.6 as:

$$F(\theta_k, \lambda_k) = f(\theta_k) + g(\theta_k)\lambda_k \tag{6.7}$$

$$f(\theta_k) = \theta_k + \eta \nabla \theta J(\pi_{\theta_k}) \tag{6.8}$$

$$g(\theta_k) = -\eta \nabla \theta J_C(\pi_{\theta_k}) \tag{6.9}$$

where η is the Stochastic Gradient Descent (SGD) learning rate. The controller's role is to push inequality constraint violations $(J_C - d)_+$ to zero.

6.4.2 Lagrangian update with PID controller

A constrained optimization problem is a problem of the form maximizing (or minimizing) the function $F(x, y)$ subject to the condition $g(x,\ y) = 0$. The Constrained MDPs have two criteria; 1) the usual reward and 2) the cost as a second value function. The reward must be optimized while the cost must remain below some specified threshold.

The PID update rule, is shown in Algorithm 7. During training, the proportional term fastens the response to constraint violations, the integral term eliminates steady-state violations at convergence, and the derivative control acts in anticipation of violations. It prevents cost overshoot and limits the rate of cost increases within the feasible region. The integral term eliminates steady-state violations at convergence. For detailed preliminary work, refer to Appendix-A [123].

6.5 Constrained Controlled RL Scheduler Design

This section presents the constraint-controlled RL scheduler overview, workload, and resource model. Next, we will discuss the Constraint-controlled RL scheduler and training algorithm overview.

In order to train the constraint-controlled RL scheduler, we convert the datacenter scheduling problem into a CMDP(§6.2.3, [120]) with a state space $\mathcal{S}$ describing the current

Algorithm 7: Update Lagrange Multiplier using PID control parameters.

```
   Initialize: $J_{C,prev} \leftarrow 0$//Previous Cost
   $I \leftarrow 0$//Integral
   $K_p, K_i, K_d \geq 0$ //Tuning parameters
1  for $i, ..., i+k$ do
2      Collect current cost, $J_C$
3      $e(t) = J_C - d$
       // proportional term
4      $P = K_p e(t)$
       // integral term
5      $I = (I + e(t))_+$
       // differential term
6      $D = K_d(J_C - J_{C,prev})_+$
       // Control output
7      $\lambda = (K_p P + K_i I + K_d D)_+$
8      $J_{C,prev} = J_C$
9      return $\lambda$
10 end
```

status of the cluster resources, an action space $\mathcal{A}$ of new jobs, and a reward function **R** to be optimized and a cost function **C** to be minimized.

6.5.1 State space, reward and cost functions

The state space (section 3.2.1), $\mathcal{S}$, includes information about jobs (section 3.2.1.2), resources (section 3.2.1.1), and resource availability (based on power generation predictions).

The CoCoRL scheduler's objective is realized with rewards and costs that the agent receives. In addition to the usual reward signal (maximize the total job value from finished jobs, section 3.2.3), the environment also provides a separate cost signal for each delayed job at every timestep. These costs are separate from the task-based reward signal.

$$Total\ Job\ Value = \sum_{i=1}^{|J_{finished}|} j_i.value \tag{6.10}$$

$$Cost = \sum_{i=1}^{n} \mathbf{1}_{j_i \in J} \tag{6.11}$$

We can customize the costs to be an aggregate cost signal reflecting more than one constraint, e.g., job delays and QoS violations. By default, cost functions are simple indicators of whether a job is delayed. The cost signal, waiting or delayed jobs, is expressed in eq 6.11. We chose a simple cost function, but practitioners can opt for a complex cost function that embodies different penalties rolled up into a single cost function.

Figure 6.3 illustrates the CoCoRL scheduler overview. We applied the Constrained-controlled Proximal Policy Optimization (CPPO) [123] policy gradient method, a constraint-controlled variant of PPO [135].

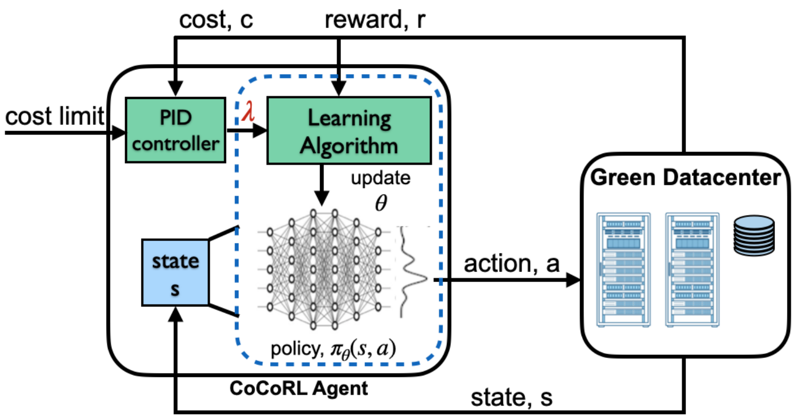


Figure 6.3: RL scheduler with PID controller (highlighted in green) interacting with the green datacenter. A Constraint-controlled policy ensures that the system actions generated by the policy neural network do not violate constraints during training and deployment.

6.5.2 Constraint-controlled RL scheduler training

The training (Algorithm 8) follows the typical minibatch-RL scheme. The agent senses the current state takes action, and records the reward information for a fixed number of time steps in each episode. The trajectory taken during each episode is recorded as $\tau =< s_0, a_o, r_1, s_1, a_1, r_2, \ldots >$. Rewards are computed from recorded values for each time step t of every episode. The sampled estimates of the cost criterion, $\hat{J}_C$, are fed back to control the Lagrange multiplier (refer to Algorithm 7). As the agent learns for rewards, the upward pressure on costs from reward learning can change, requiring a dynamic response. The costs are fed to the PID controller to control the Lagrange multiplier, λ. The $\theta-$learning loop in the CPPO method updates the parameters, θ, accordingly.

Algorithm 8: CoCoRL training with Constrained PPO policy gradient method.

```
**Input:** Batch Size $B$, Learning Rate $\alpha$, Discount $\gamma$, Cost limit
Datacenter Simulator Env with Dynamics $\mathbf{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$
Reward Function $\mathbf{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$
Cost Function $\mathbf{C} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{C}$
**Initialize:** Actor Net $\pi_\theta$, Critic Nets $V_\phi$, $V_{C,\psi}$,
Control rule, $J_C \leftarrow [\,]$ //cost history
1  for training step $t \in \{0, \ldots, T\}$ do
2      Sample the environment: mini-batch
3          // sample action
4          $a \sim \pi(\cdot|s;\theta)$
5          // transition to new state
6          $s' \sim T(s, a)$
7          // collect reward
8          $r \sim R(s, a, s')$
9          // collect cost
10         $c \sim C(s, a, s')$
11     Apply feedback control:
12         Store sample estimate $\hat{J}_C$ into $J_C$
13         $\lambda \leftarrow h(J_C, d), \lambda \geq 0$ //see algo 7
14     Update $\pi$ using Lagrangian objective
15         Update critics, $V\phi(s), V_{C,\psi}(s)$
16         // reward and cost policy gradients
17         $\nabla_\theta L = \nabla_\theta \hat{J}(\pi_\theta) - \lambda \nabla_\theta \hat{J}_C(\pi_\theta)$
18 end
**Output:** Trained Scheduling Policy $\pi_\theta$
```

6.6 Evaluation

We investigated the performance of the CoCoRL scheduling algorithm, the CPPO [123] policy gradient method described in Algorithm 8, on a small green datacenter setting with ten resources. The policy network is a two-layer Multilayer Perceptron (MLP) and a final Long Short-Term Memory (LSTM) [136]. The reward function (the primary objective) maximizes the total job value, while the cost function (the secondary objective) limits the overall job delay. We show the effectiveness of PID control in maximizing the total value while simultaneously reducing constraint violations. By design, the baseline heuristic scheduling policies do not seek to reduce constraint violations.

6.6.1 Experimentation conditions

For experiment sections 6.6.3.1, 6.6.3.2, and 6.6.3.3, we used a synthetic workload (refer to Chapter-4, section 4.4.1). For experiment sections 6.6.4 and 6.6.5, we trained and evaluated the CoCoRL scheduler using ANL [100], SDSC-SP2 [101], SDSC-Blue [102], and PIK-IPLEX [103] workloads. All the experiments in the following sections were conducted with the real power prediction data (solar and wind) from GLEAMM datacenter [107]. Jobs arrive in an online fashion, meaning that the scheduler does not know the job information *a priori*. The job arrival rates vary between 20% to 120%. Refer to Chapter-4, 4.4.2 for additional details on these datasets.

6.6.2 Performance metrics

The performance metrics used for the following experiments are Total Job Value (Chapter-3, section 3.2.3), Job Completion ratio, and System Utilization ratio. The job Completion ratio is the ratio of the jobs finished and the total jobs submitted during the simulation. System Utilization is the ratio of resources used and the total number of resources.

The primary objective of the CoCoRL scheduler is to maximize the total job value, and the secondary objective is to minimize the overall job delays. The primary objective, total job value, is directly measured by calculating the value of the completed jobs. The secondary objective, cost due to job delays, is measured indirectly through job completion and system utilization ratios. The rationale is that fewer jobs are delayed if more jobs complete on time. Similarly, high system utilization may indicate more running jobs, thus reducing overall job delays.

The baseline heuristic scheduling policies, namely, QoS (Quality of Service), Shortest-Job-First (SJF), First-Come-First-Serve (FCFS), and Highest Value First (HVF), are used for comparison. For a detailed explanation of these heuristic policies, refer to Chapter-5, section 5.4.2.

6.6.3 Performance with synthetic workload

6.6.3.1 Performance - total job value

This section evaluates the performance of different scheduling policies as the job arrival rate increases. Figure 6.4 shows performance in terms of the Total Job Value ratio, the total value of finished jobs, and the total value of all the jobs during the entire simulation and varying job arrival rates (between 20% and 120%). The 95% confidence interval for each point of the Job Value ratio is within ±0.025.

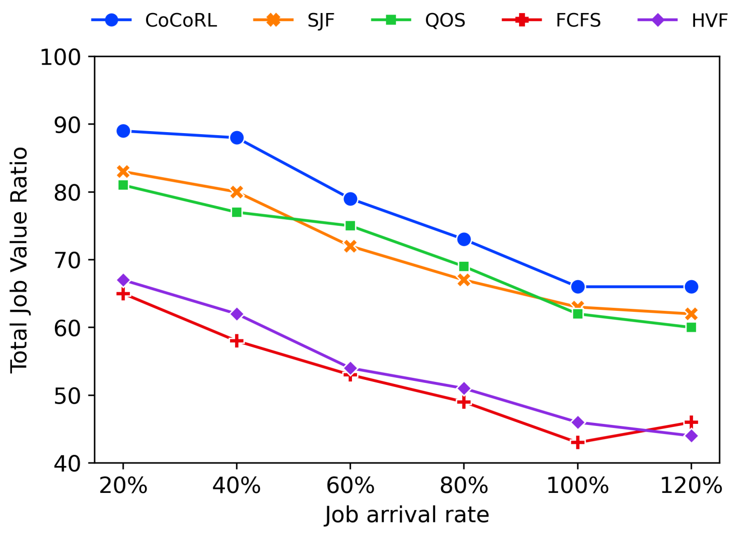


Figure 6.4: Total job value ratio with varying system load.

Analysis: The primary objective of the CoCoRL scheduler is maximizing the total job value, which the CoCoRL scheduler accomplishes, illustrated in Figure 6.4. The performance of the CoCoRL scheduler is significantly (7-24%, 8-30% and 4-26%) better than heuristic policies for arrival rates of 20%, 40%, and 60%, respectively, since there are fewer jobs (lower job arrival rate) than available resources in the datacenter. The total job value ratio gradually decreases as the job arrival rate increases (60-80% arrival rate) due to more jobs than available resources. At 80% job arrival rate, the CoCoRL performs 4-21% better than baseline heuristic policies. As the system load increases (100-120% arrival rate), CoCoRL still maintains a 4-20% higher job value ratio than heuristic policies. We see the total job value ratio dropping due to a significantly higher number of jobs than available resources, i.e., more jobs are not completed (thus losing their value) as the arrival rate increases.

6.6.3.2 Performance - job completion

This section evaluates the performance of different scheduling policies and their job completion ratio as the job arrival rate increases (between 20% and 120%). Figure 6.5 shows performance in terms of job completion ratio, i.e., the number of jobs completed and the total number of jobs as the job arrival rate varies. The 95% confidence interval for each point of the Job Completion ratio is within ±0.015.

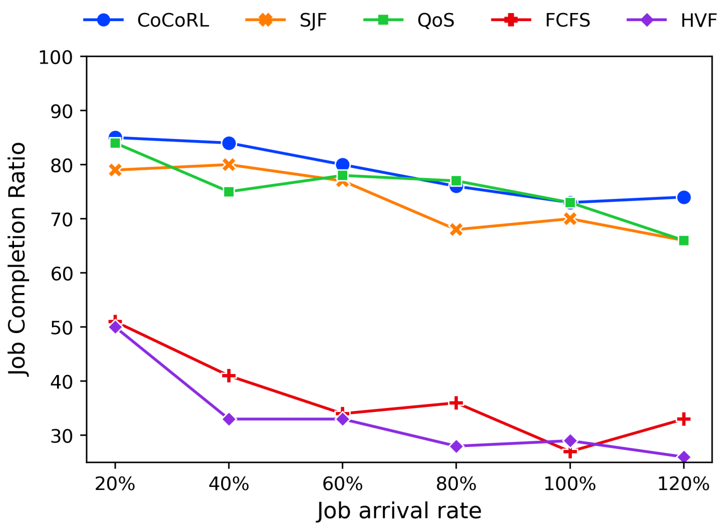


Figure 6.5: Job completion ratio with varying system load.

Analysis: From Figure 6.5, the CoCoRL and QoS heuristic policy demonstrate similar job completion ratio low-medium system load (20-80% arrival rate) with CoCoRL performing slightly better than QoS policy. As the system load increases (80-100% arrival rate), the CoCoRL and QoS scheduling policies complete significantly more jobs (77% and 73%) than the other heuristic policies. When the system load is 120%, the CoCoRL scheduler performs significantly better with 8% higher job completion ratio than the QoS and SJF and 40% better than other policies.

6.6.3.3 Performance - system utilization

This section evaluates the system utilization using different scheduling policies as the job arrival rate increases. Figure 6.6 shows performance in terms of system utilization as the job arrival rate increases (between 20% and 120%). The 95% confidence interval for each point of the Utilization ratio is within ±0.01.

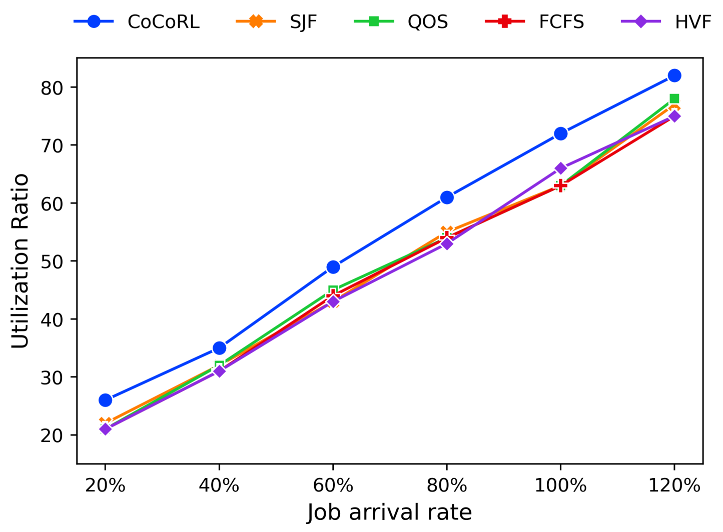


Figure 6.6: System utilization ratio with varying system load.

Analysis: From Figure 6.6, the system utilization is low when the load on the system is low (20-60% load), with CoCoRL showing 4% better utilization. As the system load increases, the system utilization increases for all the scheduling policies, but CoCoRL shows a consistently higher system utilization of 4%-6% than heuristic policies. Since CoCoRL schedules more jobs (from the previous experiment), more resources are used, leading to higher system utilization. Even though more jobs are available as the system load increases, it is possible that some of the jobs' resource requirements cannot be satisfied, fragmenting the resources. The fragmented resources lead to the underutilization of the system. Therefore, we do not see 100% utilization even when the system load is 120%.

Performance - Cost: This section evaluates the total cost using different scheduling policies as the job arrival rate increases. Figure 6.7 shows the performance of the CoCoRL in terms of the total cost accrued during the entire simulation with varying job arrival rates (between 20% and 120%).

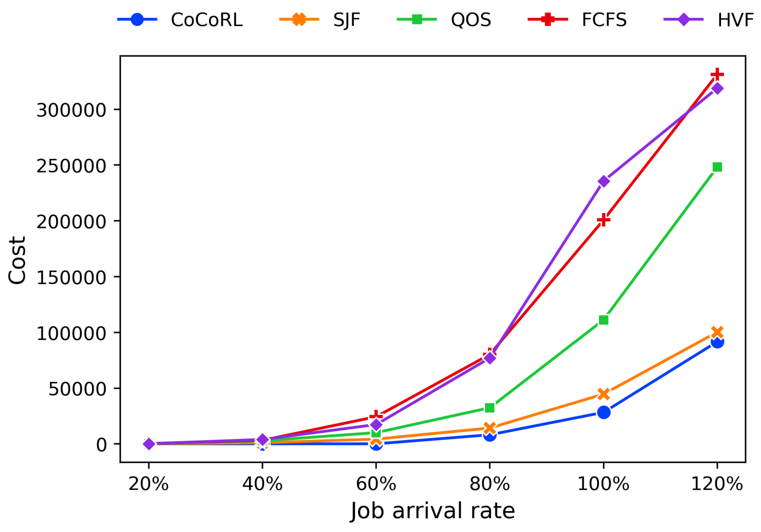


Figure 6.7: Total cost with varying system load.

Analysis: From Figure 6.7, CoCoRL accrues significantly lower cost, 1x, 10x, and 20x lower, than SJF policy for low job arrival rates of 20%, 40%, and 60% respectively. As the job arrival rate increases, more jobs are delayed leading to an increase in the cost for all the policies. We note that CoCoRL cost grows much slower and significantly lower compared to other heuristic policies, with 2x lower than the SJF policy.

Finally, we compare the total job value, completion, and system utilization ratio for different scheduling policies at 80% and 100% job arrival rates plotted in Figure 6.8.

In summary, Figure 6.8 shows that CoCoRL can simultaneously achieve (1) a higher total job value, (2) high system utilization, and (3) a high job completion ratio. At 100% job arrival rate (Figure 6.8b), CoCoRL performs at par or better than the QOS heuristic scheduling policy. None of the other heuristic policies under comparison meet these goals simultaneously.

6.6.4 Performance with HPC workloads

This section evaluates the performance of different scheduling policies with the real HPC workloads. Figure 6.9 shows performance in terms of the Total Job Value ratio, total job completion ratio, system utilization ration, and the total cost during the entire simulation

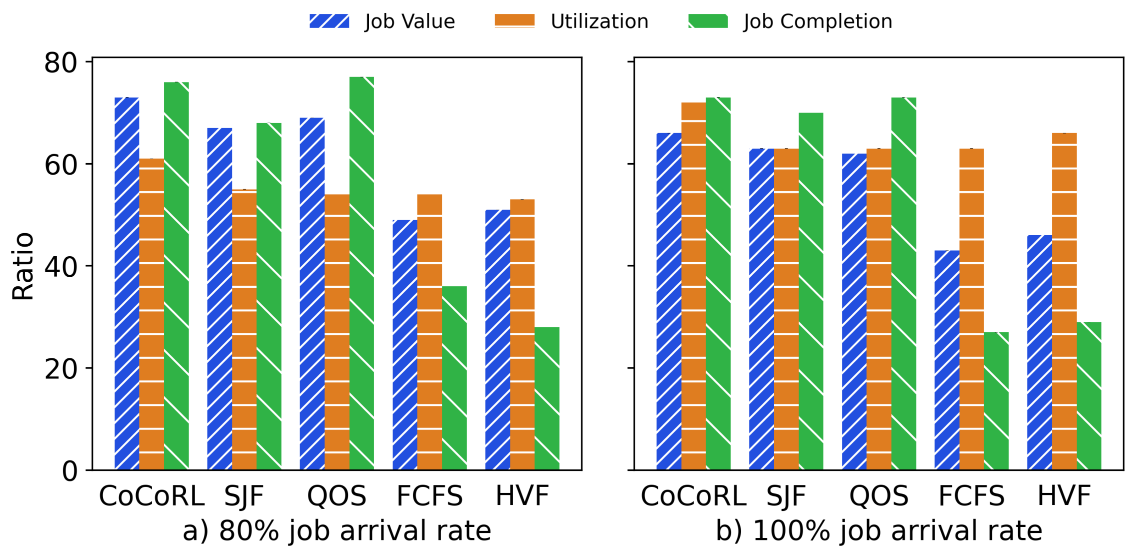
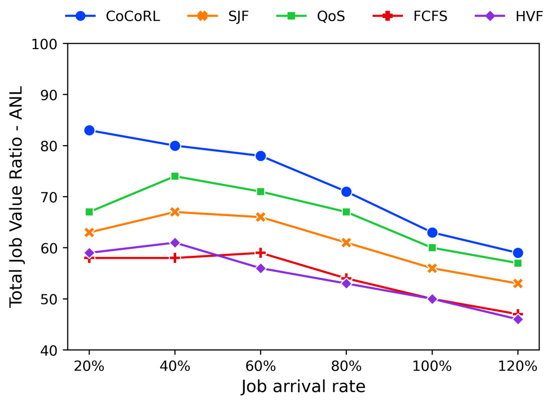


Figure 6.8: CoCoRL's performance compared to heuristic policies - Job Value, Jobs Completed, and System Utilization ratios at 80% and 100% job arrival rate.

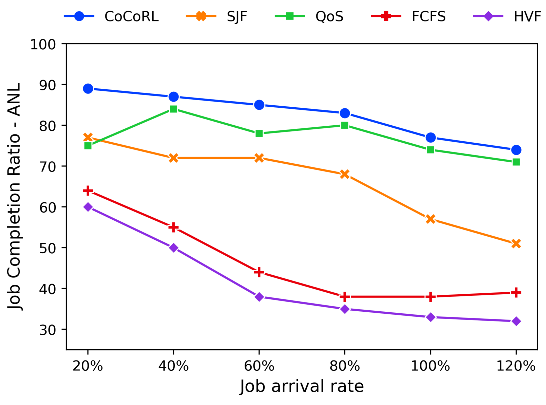
and varying job arrival rates (between 20% and 120%). The 95% confidence interval for each point of the Job Value ratio is within ±0.02.

Analysis: In Figure 6.9a, CoCoRL performs significantly (15-23%, 6-22%, and 7-22%) better for arrival rates of 20%, 40% and 60% respectively with ANL workload. At 80% job arrival rate, the CoCoRL performs 4-17% better than baseline heuristic policies. As the system load increases (100-120% arrival rate), CoCoRL's performance maintains a moderately higher (3-10%) total job value than heuristic policies. All scheduling policies perform consistently similarly as the job arrival rate increases, the artifact of ANL job characteristics as most jobs in this workload are big jobs (higher resource requirement and longer job duration). Similarly, CoCoRL outperforms the heuristics policies with HPC workloads shown in Figure 6.10a, Figure 6.10b, Figure 6.10c, and Figure 6.10d.

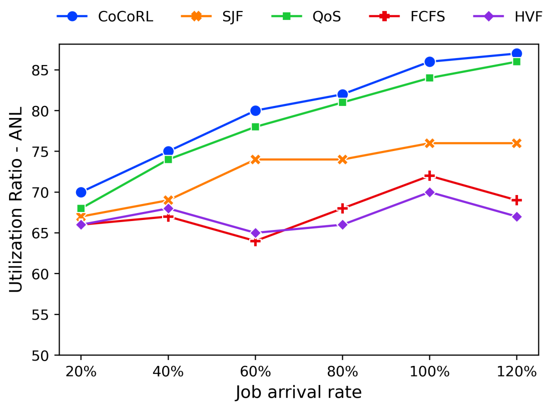
The job completion ratio of different scheduling policies using ANL workload is plotted in Figure 6.9b. At 20% and 40% job arrival rates, CoCoRL completes 13%-21% and 8%-10% more jobs, respectively, compared to heuristics policies. At 60% higher job arrival rates, CoCoRL performs at par to slightly better than the heuristic policies. The system utilization ratio is plotted in Figure 6.9c illustrating CoCoRL's system utilization which is 2%-20% better compared to best (QoS) and least (HVF) heuristic policies.



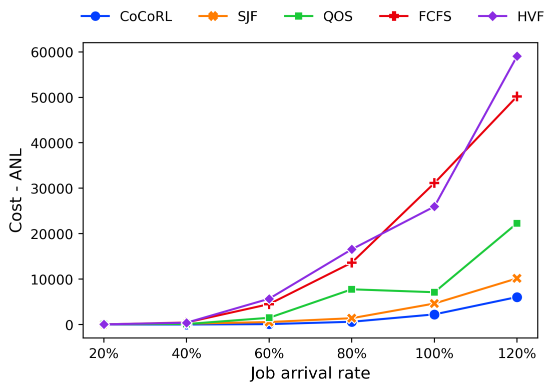
(a) Total Job Value



(b) Job Completion



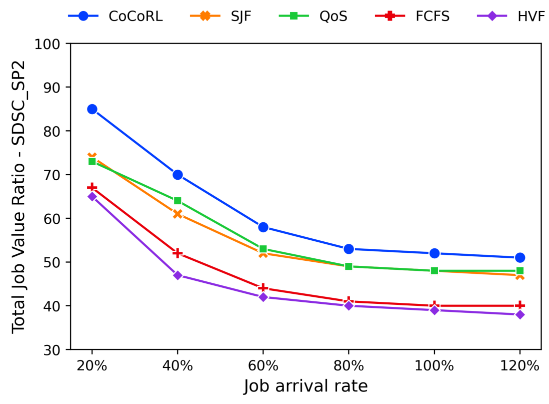
(c) System Utilization



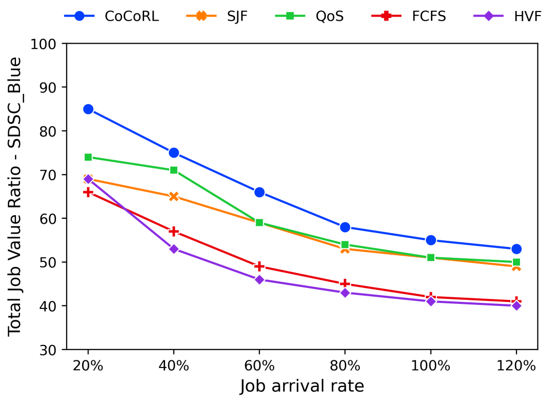
(d) Cost

Figure 6.9: Total job value, job completion, system utilization ratio and cost with varying system load - ANL workload.

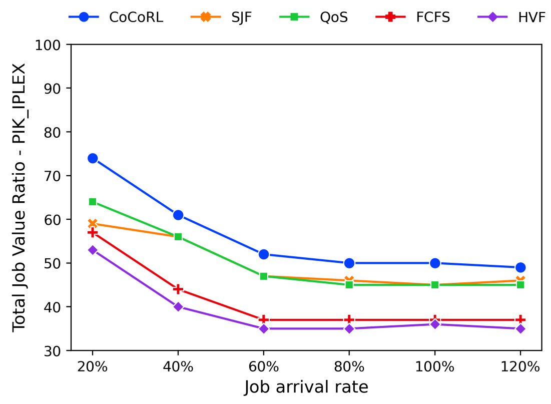
From 6.5.1, the environment provides a separate cost signal for each delayed job at every timestep. The cost is separate from the reward signal. To select the cost_limit, we ran the unconstrained RL scheduler in the datacenter environment. For each episode, we collected the cost for each delayed job during that episode. The total cost in the unconstrained environment gives us the baseline. The goal, then, is to limit these accrued costs. Therefore, the cost_limit supplied to the CoCoRL scheduler is much less than the total cost collected in the unconstrained environment. Figure 6.11 shows the performance of the CoCoRL in terms of the total cost accrued during the entire simulation with varying job arrival rates (between 20% and 120%). The 95% confidence interval for each point of



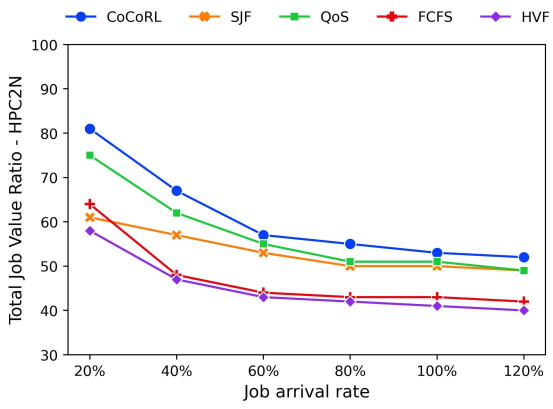
(a) SDSC SP2 workload



(b) SDSC Blue workload



(c) PIK-IPLEX workload

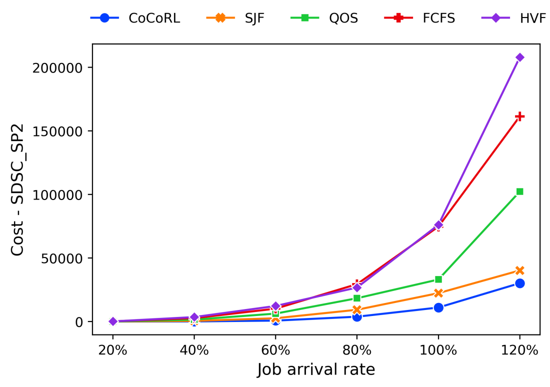


(d) HPC2N workload

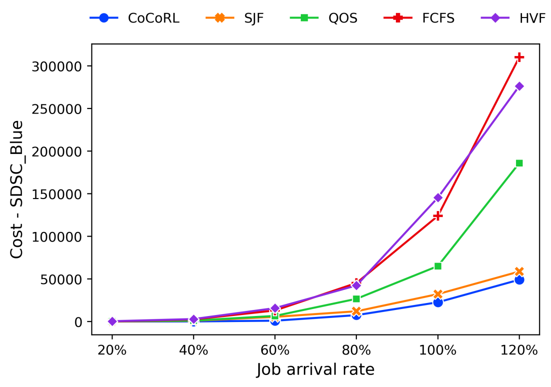
Figure 6.10: Total job value ratio with varying system load and real workloads.

the Job Value ratio is within ±0.25. For this experiment, the cost_limit was fixed at 2000.

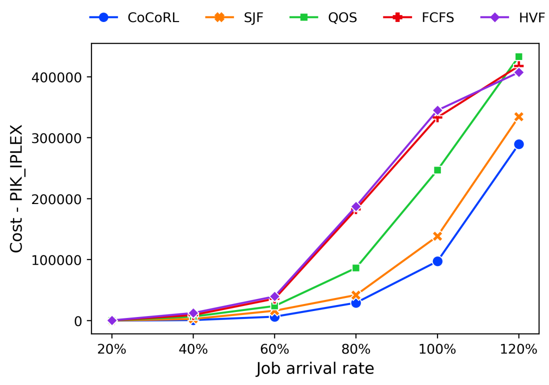
From Figure 6.9d (ANL workload), at 20% job arrival rate, CoCoRL, SJF, and QoS all accrue similar costs, whereas FCFS and HVF accrue higher costs as the job arrival rate increases. We note that CoCoRL accrues significantly lower costs for job arrival rates 0f 40-80%. CoCoRL and SJF maintain a meager accrued cost as the job arrival rate increases, while the other heuristic policies accrue significantly higher costs. Similarly, we note CoCoRL has considerably lower costs compared to heuristics policies with different workloads shown in Figure 6.11a, Figure 6.11b, Figure 6.11c, and Figure 6.11d.



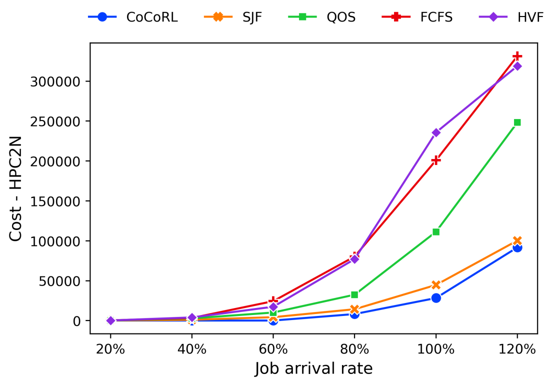
(a) SDSC SP2 workload



(b) SDSC Blue workload



(c) PIK-IPLEX workload



(d) HPC2N workload

Figure 6.11: Accrued total cost with varying job arrival rates and real workloads.

6.6.5 Hyperparamaters - PID values and cost limits

In this subsection, we will highlight the significance of tunning parameters, namely gain values for $K_p,\ K_i,\ and\ K_d$ shown in the training Algorithm 8. We note that the values of K_p, K_i, and K_d are configured during training to satisfy the corresponding optimization objective. The K_p, K_i, and K_d gain values can be identified by trail-and-error or using the Ziegler-Nichols method [137]. In addition to gain values, setting the appropriate cost_limit ensures that the scheduler learns the appropriate Lagrange multiplier, which steers the agent toward desired behavior and stays within specified cost limits. The cost limits for various workloads were identified by running the unconstrained RL and accumulating cost signals for delayed jobs. We further reduced the cost limit collected in the previous step, such that the scheduler can learn to make a reasonable trade-off between rewards and costs. For example, if the unconstrained RL scheduling agent recorded a cumulative cost of 11000 (due to delayed jobs), then the cost limit was reduced to 7000 while training the CoCoRL scheduler agent. Then, the CoCoRL agent is configured to learn scheduling actions so that it does not exceed the cost limit of 7000.

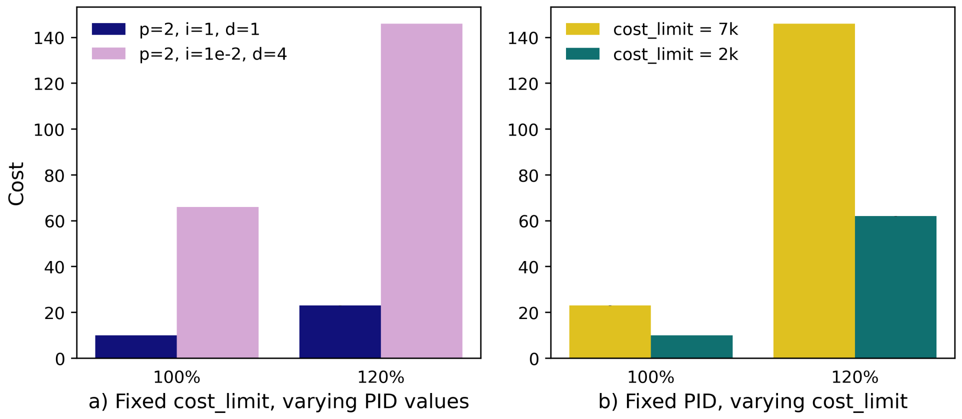


Figure 6.12: Accrued cost with different PID values and cost_limits - ANL workload.

Analysis: Figure 6.12a shows the performance of the two models trained with different PID values ($K_p = 2, K_i = 1, K_d = 1$ and $K_p = 2, K_i = 1e - 2, K_d = 1$). The model with a higher integral value, $K_i = 1$, maintains significantly lower costs than the lower integral value, $K_i = 1e-2$, for job arrival rates of 100-120%. The high K_i setting generally achieves better cost performance due to efficient cost control.

Figure 6.12b shows the performance of the two models trained with different cost_limit values (7000 and 2000). For example, the datacenter operator can set conservative cost_limit to ensure QoS violations are minimized. On the other hand, selecting a lenient cost_limit might generate a higher total job value where the scheduler picks higher-value jobs at the risk of violating the QoS agreements of low-value jobs. In essence, if the cost limit is too small, the scheduler agent will learn unsafe scheduling behavior leading to QoS violations. If the cost is too severe, the agent may fail to learn anything useful. Appropriately modeling the average cost in an unconstrained setting and training the RL scheduler to stay within the required cost limit will balance the trade-off between multiple objectives.

6.7 Future work

We tuned hyperparameters to attain our optimization goals. However, our hyperparameter settings do not indicate the best possible performance of each of our optimization goals. We plan to explore the tuning hyperparameters for other multi-criteria optimization problems. For example, selecting from various power sources to minimize brown energy and battery usage. Multi-criteria optimization is ongoing, and we will cover this topic in future work.

We can generally combine multiple cost types into a single cost function and define a combined cost limit. However, to have fine-grain control over different cost types (e.g., job delays vs. SLO violations), we may need to define separate cost functions corresponding to each type. In the future, we would like to study how different cost functions can be incorporated during learning to control individual cost types in the desired manner.

6.8 Chapter Summary

In green datacenters, intermittent power supply from renewables leads to intermittent resource availability, inducing job delays and associated costs. The scheduler's objective is to schedule jobs on a set of resources to maximize the total value (revenue) while minimizing the overall costs due to job delays. In addition, datacenter operators often prioritize multiple objectives, including job completion and system utilization.

In this chapter, we presented a Constraint Controlled RL (CoCoRL) scheduler that automatically learns conflicting reward and cost functions. We accomplish this by applying the Proportional-Integral-Derivative (PID) Lagrangian methods in Deep Reinforcement Learning to the job scheduling problem to achieve favorable learning dynamics. We demonstrate our scheduler's performance for both the primary objective (maximizing total job value) and the secondary objective (minimizing costs due to job delays). We demonstrated that CoCoRL simultaneously achieves a higher total job value, high system utilization, and a high job completion ratio while keeping the costs considerably lower compared to heuristic policies. For synthetic workload, our scheduler provides a significantly higher total job value ratio between 5%-25% for job arrival rates ranging from 20-60% (fewer jobs in the system). At a higher job arrival rate of 60-80% (more jobs in the system), our scheduler performs 5%-10% better than baseline heuristic policies. The CoCoRL scheduler performs comparably to the QoS policy and completes 5%-20% more jobs than the other heuristic policies. Our scheduler shows a 2%-6% higher system utilization than heuristic policies between 80% job arrival rate. For HPC workloads, our scheduler achieves similar superior performance while staying within the cost_limit, whereas the heuristic policies accrue 5x-10x higher negative penalties. Finally, we demonstrate the significance of accurately tuning hyperparameters to satisfy various optimization goals set by datacenter operators.

Chapter 7

Learning to Schedule using Offline and Online RL Methods

Deep reinforcement learning algorithms have succeeded in several challenging domains. Classic Online RL job schedulers can learn efficient scheduling strategies but often take thousands of timesteps to explore the environment and adapt from a randomly initialized DNN policy. Existing RL schedulers overlook the importance of learning from historical data and improving upon custom heuristic policies. Data-driven reinforcement learning presents the prospect of policy optimization from pre-recorded datasets without online environment interaction. Following the recent success of data-driven learning, we explore two data-driven RL methods: 1) Behaviour Cloning and 2) Offline RL, which aims to learn policies from logged data without interacting with the environment. These methods address the challenges concerning the cost of data collection and safety, particularly pertinent to real-world applications of RL. Although the data-driven RL methods generate good results, we show that the performance is highly dependent on the quality of the historical datasets provided during training. Finally, by effectively incorporating prior expert demonstrations to pre-train the agent, we can short-circuit the random exploration phase to learn a reasonable policy with online training. We utilize offline RL as a **launchpad** to learn effective scheduling policies from prior experience collected using Oracle or heuristic policies. Such a framework is effective for pre-training from historical datasets and well suited to continuous improvement with online interaction.

7.1 Introduction and Motivation

The process of online reinforcement learning involves iteratively interacting with the environment and collecting experience, typically with the most recently learned policy, and then using the experience to improve the policy [138]. In many settings, this online interaction is impractical because data collection is expensive (e.g., robotics, healthcare) or dangerous (e.g., autonomous driving). Moreover, even in domains where online interaction is possible (e.g., job scheduling), practitioners may prefer utilizing previously collected data, especially if the domain is complex and requires large datasets for effective generalization.

Learning a task from scratch can require a prohibitively time-consuming amount of exploration of the state-action space to find a good policy, especially in sparse reward environments. Moreover, learning without prior knowledge is an approach rarely taken in the natural world. Knowledge of how to approach a new task can be transferred from previously learned tasks or extracted from the performance of an expert. The existing RL schedulers overlook the importance of learning and improving upon existing heuristic policies. The RL schedulers can leverage the behavior of custom heuristic policies explicitly designed for unique environments to learn and improve overall performance. The heuristic policies generate expert demonstrations, and the RL agents learn from these demonstrations to improve upon the heuristic policies.

For many of these domains, including job scheduling, large amounts of historical data are readily available. Effective data-driven methods for deep reinforcement learning can utilize this data to pre-train offline while improving with online fine-tuning. This has led to a resurgence of interest in data-driven RL methods, namely 1) Behaviour Cloning (BC) and 2) Offline RL (historically known as batch RL) [139], which aim to learn policies from logged data without further interaction with the real system.

Behavior cloning is an approach for imitation learning [140], where the policy is trained with supervised learning to imitate the actions of a provided dataset directly. This process is highly dependent on the performance of the data-collecting process. In many cases, these come from an existing rule-based, heuristic, or myopic policy that we are trying to replace with an RL approach. The effective use of such datasets would make real-world RL more practical and enable better generalization by incorporating diverse prior experiences. Due to the efficient use of collected data and the stability of the learning process, this research area has attracted much attention recently.

Depending on the quality of the prior demonstrations, useful knowledge can be extracted about the task being solved, the dynamics of the environment, or both. Pure BC methods incorporate prior data with the aim of directly mimicking demonstrations. This is desirable, assuming demonstrations are known to be optimal. However, it enforces strict requirements on offline data quality, which can cause undesirable bias when the demonstration data is not optimal. Often, the collected data can be mixed with sub-optimal transitions.

Offline reinforcement learning algorithms also aim to leverage large existing datasets of previously collected data to produce effective policies that generalize across a wide range of scenarios without needing costly active data collection. Empirical studies comparing offline RL to imitation learning have come to mixed conclusions. Some studies show that offline RL methods outperform imitation learning significantly, specifically in environments that require "stitching" parts of suboptimal trajectories [141]. In contrast, many recent articles have argued that BC performs better than offline RL on both expert and suboptimal demonstration data over a variety of tasks [142] [143] [144]. This makes it confusing for practitioners to understand whether to use offline RL or merely run BC on collected demos.

While offline learning methods provide a mechanism for utilizing prior data, such methods are generally ineffective for fine-tuning online data as they are often too conservative. In effect, these methods require us to question: Do we assume the prior dataset is optimal? Do we use strictly offline data or only online data? We need algorithms that learn successfully in either of these cases to make it feasible to learn policies for real-world settings. Therefore, we study a simple actor-critic algorithm (in §7.4) that bridges pre-training from prior data and improvement with online data collection. This RL technique is effective for pre-training from off-policy datasets but is also well suited to continuously improve with online data collection.

Our **contribution** is the empirical characterization of BC, offline RL, and online RL pre-trained with offline datasets for job scheduling in the green datacenter environment. Existing RL scheduler research exclusively focuses on Online RL methods. We could not find any published research exploring recent developments in data-driven RL methods applied to the job scheduling problem. We demonstrate when the different data-driven methods can be applied (based on the quality of the prior datasets) and under what conditions each method is effective. Our results emphasize the importance of leveraging decades worth of research on heuristic schedulers, using transitions from heuristic methods to learn a policy and improve it further without spending excessive time or computational effort with only Online learning. Specifically, combining the Offline+Online method speeds up learning while leveraging existing heuristic/expert demonstrators. We demonstrate how to generate datasets of varying qualities (by combining transitions from good and not-so-good heuristic policies) applicable to the job scheduling problem. The contribution of this study is not just another RL scheduler but a systematic study of what makes sense - standard offline RL, pure online RL methods, or offline pre-training with online fine-tuning. We evaluate these RL techniques in the power-modulated datacenter environment.

7.2 Background

This section briefly compares offline Reinforcement Learning, off-policy, online Learning, and offline learning with online fine-tuning methods. Figure 7.1 illustrates offline reinforcement learning (a), classic off-policy reinforcement learning (b), classic online reinforcement learning (c), and offline pre-training with online fine-tuning (d). We re-introduce the basics of RL to differentiate between different methods and to aid in readability and continuity.

Reinforcement Learning (RL). RL is a framework aimed at dealing with tasks of sequential nature. Typically, the problem is defined as a Markov decision process (MDP) (S, A, R, p, γ), with state space $\mathcal{S}$, action space $\mathcal{A}$, reward function $\mathbf{R}$ (scalar), transition

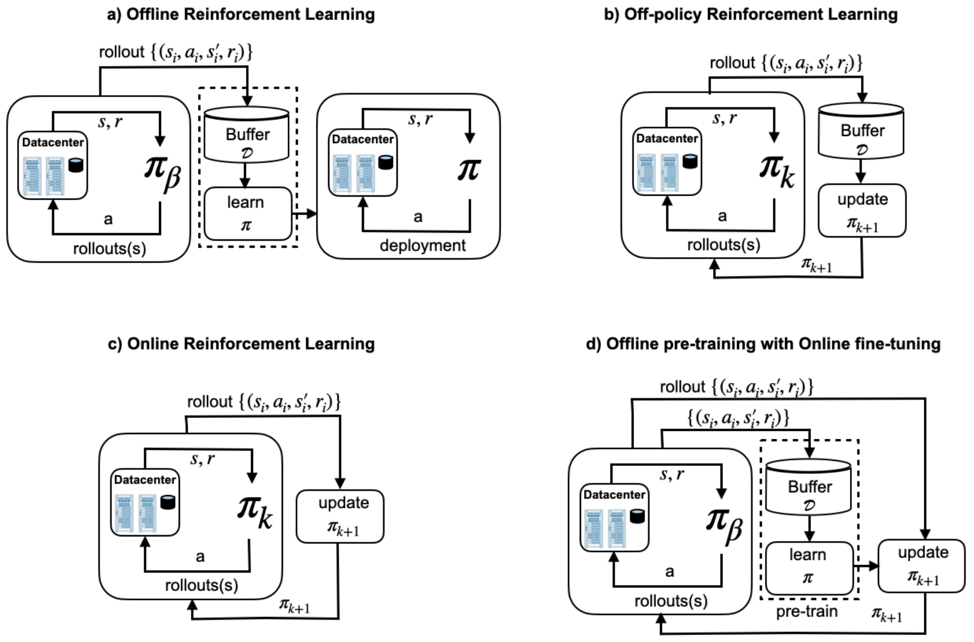

a) Offline Reinforcement Learning
rollout $\{(s_i, a_i, s_i', r_i)\}$
s, r
Datacenter
π_β
a
rollouts(s)
Buffer $\mathcal{D}$
learn π
π
deployment
b) Off-policy Reinforcement Learning
π_k
update π_{k+1}
c) Online Reinforcement Learning
d) Offline pre-training with Online fine-tuning
pre-train

Figure 7.1: Illustration of offline reinforcement learning (a), classic off-policy reinforcement learning (b), classic online reinforcement learning (c), and offline training with online fine-tuning (d).

dynamics p, and discount factor γ. The behavior of an RL agent is determined by a policy π. The RL agent's objective is to maximize the long-term expected discounted return $E_\pi[\sum_{t=0}^{\infty}\gamma^t r_{t+1}]$, i.e., the expected cumulative sum of rewards when following the policy in the MDP. This objective is evaluated by a value function, which measures the expected discounted return after taking action a in state s: $Q^\pi(s,a) = E_\pi[\sum_{t=0}^{\infty}\gamma^t r_{t+1}|s_0 = s, a_0 = a]$.

Behavior Cloning (BC). Another approach for training policies is imitating an expert or behavior policy. Behavior cloning (BC) is an approach for imitation learning [140], where the policy is trained with supervised learning to imitate the actions of a provided dataset directly. Unlike online RL, the success of BC is highly dependent on the quality of the dataset. BC is likely to fail when the prior dataset does not contain enough transitions generated by a policy performing well on the task or the signal-to-noise ratio is too large.

Off-policy. Off-policy learning consists of a behavior policy that generates the data and a target policy that learns from this data [138]. The behavior policy continuously collects data for the agent in the environment. For example, data is collected using previous policies up to time k during training $\pi_0, \pi_1, \ldots, \pi_k$ and stored in a replay buffer. This data is used to train the policy π_{k+1}. An example of off-policy RL is actor-critic variants in [145].

Offline RL. Offline RL (a.k.a batch RL), similar to BC, breaks the presumption that the agent can interact with the environment. Instead, we provide the agent with a fixed dataset collected by some unknown data-generating process (experts or heuristic policies). This setting may become challenging since the agent loses the opportunity to explore the environment according to its current views and must infer good behavior from only the provided transitions. More generally, offline RL is a counterfactual inference problem: given data generated from a given set of decisions infer the consequence of an independent set of decisions.

Offline training with online fine-tuning. This method is concerned with accelerating online fine-tuning by pre-training on smaller offline datasets. We follow the same process as offline RL to pre-train with datasets collected from experts or heuristic policies to learn policy π. Then, by initializing the policy network with π, instead of random initialization, we fine-tune the policy by interacting with the environment similar to classic online RL.

7.3 Related Work

Behavior Cloning and Offline RL. The authors in [146] propose scheduling on Domain-specific systems-on-chip as a classification problem and propose a hierarchical imitation learning-based scheduler that learns from an Oracle to maximize the performance of multiple domain-specific applications. A similar framework is suggested in [147] for mobile platforms. HiLITE [148] employs a hierarchical imitation learning framework to maximize energy efficiency while satisfying soft real-time constraints on embedded systems on chip (SoC).

Existing RL scheduler research exclusively focuses on Online RL methods. Most of the published research demonstrating data-driven methods are in the field of robotics using openAI gym environments but these methods are not demonstrated in the job scheduling context. To the best of our knowledge, there is minimal prior work, if any, using BC and offline RL methods applied to the job scheduling problem. They do not demonstrate the various data-driven methods and under what conditions each method is effective (depending on the quality of the prior datasets). We could not find any prior research applying a combination of offline pre-trianing and online learning to the job scheduling problem. In this chapter, we aimed to understand if, when, and why offline RL is a better approach for tackling sequential decision-making problems, specifically job scheduling in green datacenters.

7.4 Job Scheduling using Offline and Online Learning

Figure 7.2 illustrates an overview of an offline agent learning from prior datasets and an online agent - initialized with pre-trained policy using offline data - fine-tuning the policy by actively interacting with the green datacenter environment. The state space (section 3.2.1), $\mathcal{S}$, includes information about jobs (section 3.2.1.2), resources (section 3.2.1.1), and resource availability (based on power generation predictions).

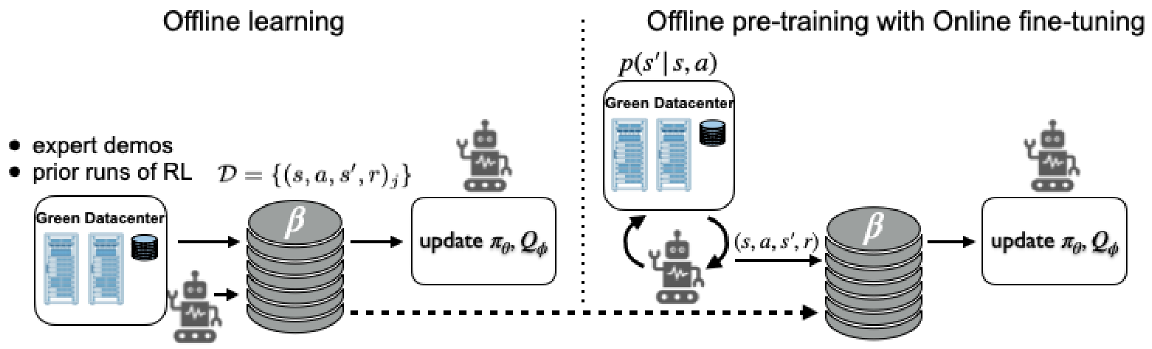

Offline learning
Offline pre-training with Online fine-tuning
• expert demos
• prior runs of RL
$\mathcal{D} = \{(s, a, s', r)_j\}$
Green Datacenter
β
update π_θ, Q_ϕ
$p(s'|s,a)$
Green Datacenter
(s, a, s', r)
β
update π_θ, Q_ϕ

Figure 7.2: RL scheduling agent: offline learning (left) and offline pre-training with online fine-tuning (right).

The core RL algorithm used to train the encoder, actor, and critic varies depending on whether we use online or offline data. The offline RL algorithm is described in Algorithm 9, and the online RL algorithm is described in Algorithm 10.

Both methods train a Q-function critic with the standard mean-squared Bellman error update [138], where our critic network(s), Q_ϕ, learn to predict bootstrapped estimates of

Algorithm 9: Offline training process

```
   Input: Advantage Samples $k$, Replay Buffer with pre-provided transitions
          $\mathcal{D} \leftarrow \{(s_i, a_i, r_i, s_i'), \dots\}$, function $f$
   Initialize: Actor net $\pi_\theta$, Critic net $Q_\phi$
1  for training step $t \in \{0, \dots, T\}$ do
2      Sample Batch of $B$ transitions $\{(s_i, a_i, r_i, s_i')\}_{i=0}^{i=B}$ from $\mathcal{D}$
       // actor loss (see [112])
3      $\mathcal{L}_a \leftarrow \frac{1}{B} \sum_{i=0}^{i=B} f(Q_\phi, s_i, a_i) \log \pi_\theta(a_i \mid s_i)$
       // critic loss
4      $\mathcal{L}_c \leftarrow \frac{1}{B} \sum_{i=0}^{i=B} \Big( \big(Q_\phi(s_i, a_i) - \mathbb{E}\left[(r_i + \gamma Q_\phi(s_i', a')\right]\big)^2 \Big)$
       // update nets by gradient descent
5      $\phi \leftarrow \phi - \alpha \nabla \phi \mathcal{L}_c$
6      $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_a$
7  end
   Output: Trained Scheduling Policy $\pi_\theta(s)$
```

future value based on immediate rewards and a "target" network. We implement an ensemble method that trains multiple critics to reduce over-estimation error, as in SAC [110] and TD3 [149]. We use PopArt [150] normalization to standardize the magnitude of our critic outputs; this simplifies tuning hyperparameters across datacenter environments with different resources and job values.

The primary difference between online and offline variants is the gradient update of the actor. The online algorithm trains the actor to maximize the value predictions of the critic, as in the discrete-action version of SAC [111]. Like SAC, we use a max-entropy RL formulation that encourages robust policies and prevents our agents' actions from collapsing to a local optima during training. As an additional exploration measure, we use an epsilon-greedy strategy [151] to add random noise to action selection during the early stages of learning. Directly maximizing the outputs of the critic exploits overestimation error in unfamiliar state-action pairs [152]. This "uninformed optimism" can be a useful exploration strategy for online learning but becomes a significant issue during offline learning when we are unable to evaluate actions' true value. The offline actor update needs to constrain the policy to the distribution of actions covered by the existing training data. This is achieved

Algorithm 10: Online training process with Offline data

```
Input: Datacenter Simulator Env with Dynamics $\mathbf{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, Reward
       Function $\mathbf{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, Replay Buffer $\mathcal{D} \leftarrow \{(s_i, a_i, r_i, s'_i), \dots\}$
Initialize: Actor net $\pi_\theta$, Critic net $Q_\phi$
1  for training step $t \in \{0, \dots, T\}$ do
       // sample action from the policy
2      $a_t \sim \pi_\theta(s_t)$
       // advance datacenter sim and receive next state and reward
3      $s'_t \leftarrow \mathbf{T}(s_t, a_t)$,
4      $r_t \leftarrow \mathbf{R}(s_t, a_t, s'_t)$
       // add transition to the replay buffer
5      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r_t, s'_t)\}$
6      Randomly Sample Batch of $B$ transitions $\{(s_i, a_i, r_i, s'_i)\}_{i=0}^{i=B} \sim \mathcal{D}$
       // critic loss
7      $\mathcal{L}_c \leftarrow \frac{1}{B} \sum_{i=0}^{i=B} \left( \left(Q_\phi(s_i, a_i) - \mathbb{E}\left[(r_i + \gamma Q_\phi(s'_i, a')\right]\right)^2 \right)$
       // online actor loss (see [110])
8      $\mathcal{L}_a \leftarrow \frac{1}{B} \sum_{i=0}^{i=B} \left( \mathop{\mathbb{E}}_{a' \sim \pi_\theta(s_i)} \left[ -Q_\phi(s_i, a') \right] \right)$
       // update nets by gradient descent
9      $\phi \leftarrow \phi - \alpha \nabla \phi \mathcal{L}_c$
10     $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_a$
11 end
Output: Trained Scheduling Policy $\pi_\theta(s)$
```

by performing supervised learning on the state-action mapping contained in the training data, filtered by a function f (Algorithm 9, line 3).

Intuitively, the actor network is trying to copy the decisions in the dataset, but the filter f prioritizes some actions over others. When f outputs the same value for all samples, we recover standard Behavioral Cloning as a special case. A popular alternative is to up-weight actions that lead to a higher return than the current policy and down-weight those that lead to a lower return. This concept is formalized by the advantage function $A^\pi(s, a) = Q^\pi(s, a) - \mathbb{E}_{a' \sim \pi(s)}[Q^\pi(s, a')]$. In practice, the advantage is estimated using the critic network and samples from the actor: $A^{\pi_\theta}(s, a) \approx Q_\phi(s, a) - \frac{1}{k} \sum_{j=1}^{j=k} Q_\phi(s, a_j \sim \pi_\theta(s))$. The filter function f can then down-weight samples with low advantage, such as $f(Q_\phi, s, a) = \exp(A^{\pi_\theta}(s, a))$ [153]. Our work follows [112] and uses a more intuitive binary filter that simply ignores data that the critic network does not think will improve the policy

$f(Q_\phi, s, a) = \mathbb{1}\{A^{\pi_\theta}(s, a) > 0\}$.

7.5 Evaluation

Empirically, we found that QoS and SJF heuristic policies perform better than other heuristic policies for small and medium cluster sizes (10 to 100 resources). Therefore, we use QoS policy as a baseline and combined transitions from other heuristic policies to generate datasets with varying quality of noise-to-signal ratio. We used a synthetic workload, and a job arrival rate of 120% was used for all the experiments. The performance metric used is Total Job Value.

We ran five training runs for each learning method with randomly initialized scheduler agents. To evaluate, we averaged ten rollouts (for each of 5 training runs) with 100K steps and random seed. The 95% confidence interval for each point in the following graphs is within the ± 0.015 and ± 0.023 range.

7.5.1 BC and Offline learning

Behavior Cloning (BC): We start by demonstrating the performance of the first data-driven method called Behavior Cloning. The performance of BC methods is highly dependent on the quality of the training dataset. BC is likely to fail to learn good policy when the dataset does not contain enough transitions generated by a well-performing policy or the fraction of poor data is too large. For this experiment, we collected transitions from heuristic policies, namely SJF, QoS, FCFS, and HVF policies. We then load a combination of these transitions into the replay buffer BC_combo scheduler agent. Each policy contributes 25% of the total transitions loaded into the replay buffer. We repeat the training by loading only QoS and SJF transitions into the replay buffer to train the BC_qos_sjf agent. In this case, each heuristic policy contributes 50% of the total transitions loaded into the replay buffer. Finally, we loaded only QoS transitions into the replay buffer, i.e., 100% of the replay buffer consists of transitions from QoS policy to train the BC_qos agent.

Analysis: Figure 7.3 shows the performance of BC agents when trained with different

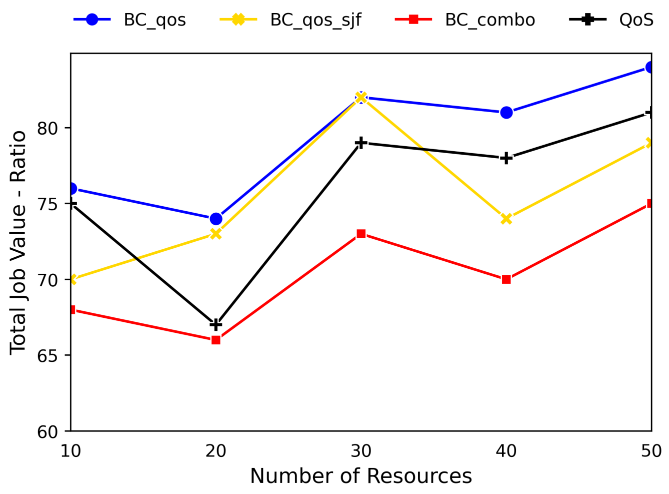
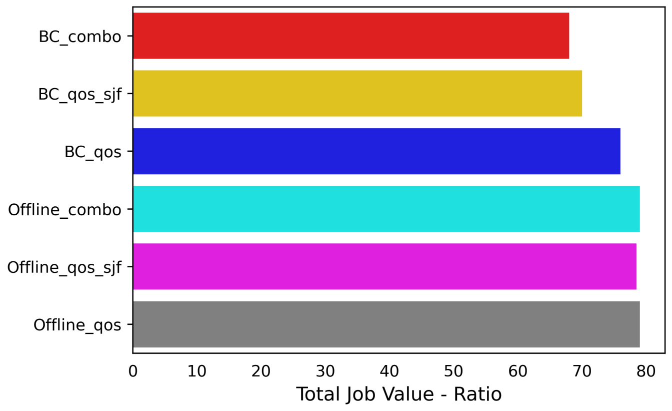


Figure 7.3: Performance of BC method when trained using prior datasets generated by heuristic policies with varying noise-to-signal ratio.

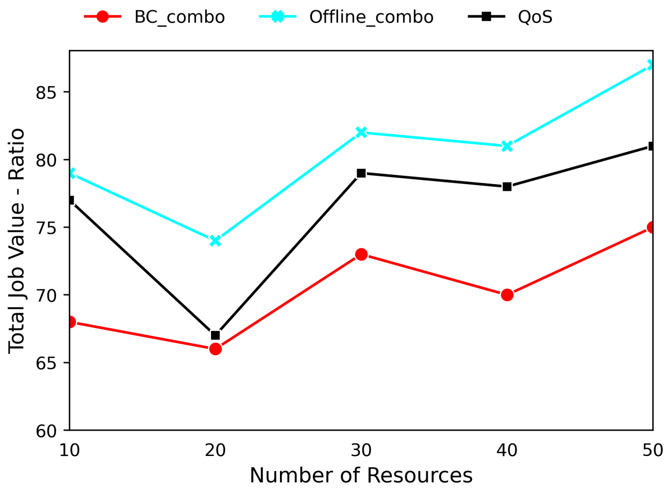
ratios of transitions from different heuristics. When the replay buffer consists of a mix of expert and poor quality transitions (e.g., QoS, SJF, FCFS, HVF), the BC method (BC_combo) learns a policy that performs worse than the QoS policy alone. Even with a 50% noise-to-signal ratio (i.e., 50% QoS and SJF), BC_qos_sjf performs poorly in some instances. When the reply buffer is populated with a higher ratio of good transitions (i.e., 100% QoS, best heuristic policy), then BC_qos effectively learns to mimic that policy. The BC method performs significantly better when trained with high-quality demonstrations.

Note that BC learning methods are incredibly beneficial in two scenarios. First, suppose we have prior datasets from an Oracle or a good heuristic policy but need to learn a general policy that works in a similar but slightly different setting. In that case, BC can learn the general policy from transitions generated by the Oracle for the new setting. Second, suppose we have transitions from a legacy or proprietary system and cannot or do not have access to the system's code base that generated the transitions. In that case, BC learning can be applied to learn the underlying policy from the transitions generated by the legacy system.

Offline RL: Next, we demonstrate the performance of the second data-driven method called offline RL. Unlike BC, the performance of offline RL is resilient to training datasets with mixed (both well-performing and poor) heuristic policies. Suppose prior datasets do not contain enough transitions generated by a well-performing policy, or the fraction of poor data is too large. In that case, offline RL methods can leverage the benefits of stitching parts of suboptimal trajectories. Similar to the previous experiment, we load a combination of heuristic transitions (SJF, QoS, FCFS, and HVF policies) into the replay buffer for training the offline RL agents.



(a) Varying quality of prior datasets (10 resources).



(b) BC and offline RL performance, scaled up to 50 resources.

Figure 7.4: Performance comparison: BC and offline learning with varying quality of the prior datasets.

Analysis: From the Figure 7.4a, we note that BC_qos agent performs better than BC_qos_sjf and BC_combo. The BC_qos_sjf and BC_combo agents were trained with a mix of poor-quality transitions. Since the BC agent learns to merely mimic the transitions in the replay buffer, the presence of poor-quality transitions affects the performance of these two agents. Figure 7.4a shows the performance of offline RL compared to BC, with the replay buffer of prior data consisting of a mix of well-performing and poor transitions. The Offline_combo performs significantly better than BC_combo when the training data has a mix of good and bad transitions. Note that BC_combo performs worse than QoS due to a higher percentage of poor-quality data, shown in Figure 7.4b.

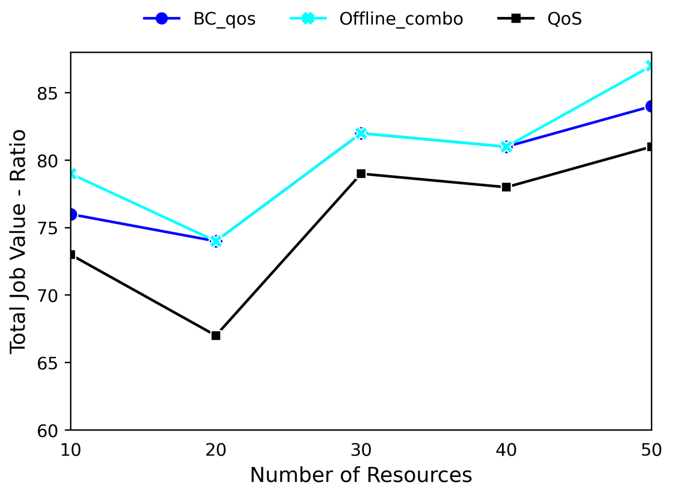


Figure 7.5: Performance comparison: BC, offline learning and best heuristic policy (i.e., QoS).

Figure 7.5 shows the performance of offline RL compared to BC and Qos heuristic methods when scaled to 50 resources. At scale, the Offline_combo significantly outperforms both BC_combo and QoS policies. Additionally, Figure 7.5 shows that Offline_combo agent trained on sufficiently noisy suboptimal data can attain at-par or better performance than even BC_qos method trained with expert demos. Furthermore, the Offline_combo, Offline_qos_sjf, and Offline_qos agents perform similarly (Figure 7.4a), showing that the offline Learning method is resilient to noisy datasets during training. The offline RL methods benefit from stitching parts of suboptimal trajectories. For example, suppose the dataset contains a subsequence illustrating arrival at state $x + 1$ from state x and another subsequence illustrating arrival at state $x + 2$ from state $x + 1$. In that case, an effective offline RL method should be able to learn how to arrive at state $x + 2$ from state x, which might provide a substantially higher final reward than any of the subsequences in the dataset. This sort of "transitive induction" occurs on a portion of the state variables, effectively inferring potentially optimal behavior from highly suboptimal components.

The offline RL method facilitates generalization, i.e., it can be adapted to learn from any policy (e.g., Oracle, heuristic policies) that optimizes a specific objective, such as job value, resource utilization, or energy efficiency. We remark that the offline RL method is significantly better when training buffers are loaded with a mix of transitions generated by more than one policy. With mixed transitions (well-performing and poor), offline RL can learn a general policy, possibly stitching multiple policies to get a better one.

7.5.2 Online learning

The most obvious challenge with offline RL is that because the learning algorithm must entirely rely on the static dataset, there is no possibility of improving by exploration. If the dataset does not include transitions that illustrate high-reward regions of the state space, it may be impossible to learn such high-reward regions. For example, when the dataset size is limited, some learning algorithms tend to overfit on the small dataset, or if the dataset state-action distribution is biased, neural network training may only provide brittle, non-generalizable solutions. Additionally, not all environments have historical or high-quality datasets that can be readily used for training.

In online learning, the agent interacts with the environment and explores numerous state-action pairs to learn a generalizable policy. Model-free deep RL methods are notoriously expensive in terms of their sample complexity. Even relatively simple tasks can require millions of data collection steps, and complex behaviors with high-dimensional observations might need substantially more. What is considered an upside, the exploration process, is also the downside because the exploration process is time-consuming, where the agent alternates between the exploration and exploitation phases to learn decent policy. We trained the BC and offline agents for 500k steps and the online agent for 1 million steps. Everything else being the same (e.g., neural net configuration, batch size), the online agent was trained twice as long as the BC and offline methods to achieve similar performance goals.

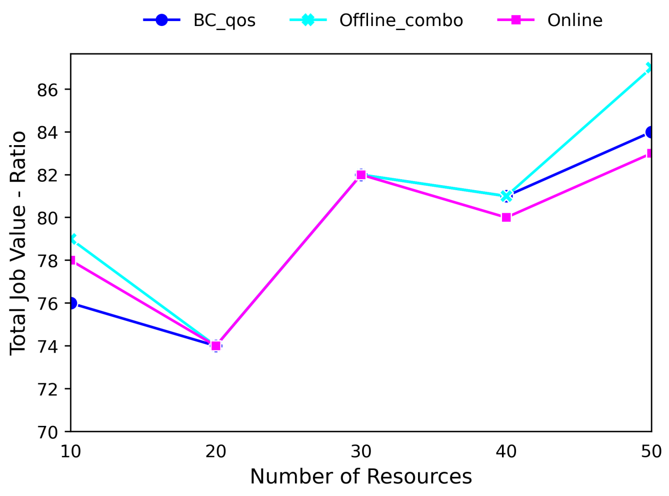


Figure 7.6: Performance comparison: BC, offline, and online learning methods.

Analysis: Figure 7.6 shows the performance of online RL compared to BC_qos and Offline_combo Learning methods. We note that the online method performs as well as the BC and offline methods when the state space is small (10-30 resources). The performance drops as the state space increases (40-50 resources). This is because exploring large state space needs more interactions with the environment and, therefore, a longer time to train. With enough time and random starting points, the online agent might eventually match or outperform the BC and offline methods for large state-space problems.

7.5.3 Combining Offline and Online learning

Online RL provides an appealing formalism for learning control policies from experience. However, the classic active interactions with the environment require a lengthy active exploration process for each behavior. Suppose we allow RL algorithms to use historical datasets to aid online learning effectively. In that case, the learning process could be made substantially more practical: the prior data would provide a starting point, a **launchpad**, that mitigates challenges due to exploration and sample complexity, while the online training enables the agent to perfect the desired skill. Such prior data could either constitute expert demonstrations or, more generally, sub-optimal prior data that illustrates potentially useful transitions. Given the dataset, $\mathcal{D}$, of transitions generated by a heuristic policy, our goal is to leverage $\mathcal{D}$ for pre-training and use some number of online interactions to learn the good generalizable scheduling policy.

For this experiment, we used a modified version of advantage weighted actor-critic (AWAC) [154](Algorithm 10) framework, which enables rapid learning of skills with a combination of prior datasets and online experience. This framework leverages offline data and quickly performs online fine-tuning of RL policies. Additionally, incorporating prior datasets can reduce the time required to learn with large state space.

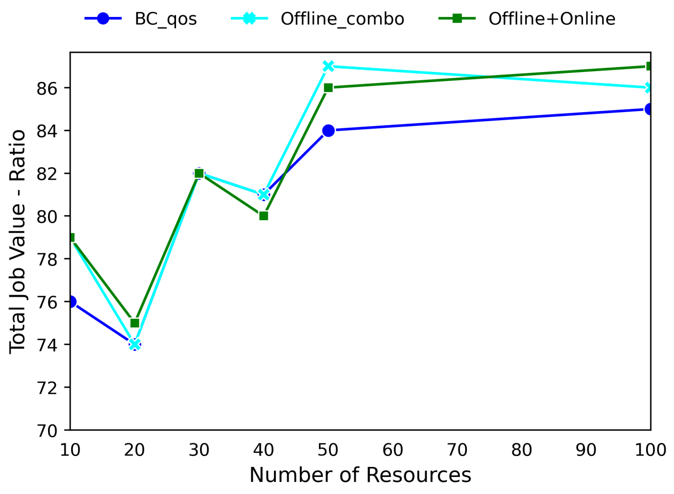


Figure 7.7: Performance comparison: BC, offline learning and online learning with offline pre-training methods.

Analysis: Figure 7.7 shows the performance of online training using offline data (offline+online) compared to BC_qos and Offline_combo learning methods. For this experiment, we trained the agent for 50k offline steps (pre-loading QoS transitions to the replay buffer) and 1 million online steps. Even though the offline+online was trained for merely 50k offline steps, the offline+online agent's performance is at par or better than either BC or offline methods for large problem sizes (100 resources). This framework is effective for pre-training from off-policy datasets but is also well suited to continuous improvement with online data collection. Additionally, this framework can utilize different types of prior data: demonstrations, suboptimal data, or random exploration data.

7.5.4 Effectiveness of Offline and Online learning methods

In this section, we will discuss the effectiveness of the offline and online learning methods and present the scenarios under which a given RL technique is applicable.

First, we collected 80 rollouts (sequences of 100-200k samples) of offline experience data where the heuristic scheduling policies (QoS, SJF_QoS, and Combo) select the actions. Second, we load the offline experience into the empty replay buffer during the RL scheduler's training. We trained three RL scheduler agents with BC, offline RL, and offline+online methods. In each of these methods, the RL scheduler's actor net learns to mimic the action choices of the heuristic data in its replay buffer to varying degrees. To evaluate the success of the learning process, we simulated new rollouts controlled by the original heuristic and measured the percentage of steps where the policy's action is equal to the heuristic's decision in the current state. This *action agreement* metric provides some insight into each of the RL method's ability to learn heuristic policies.

Action Agreement	QoS	SJF_QoS	Combo
BC	95%	99%	98%
Offline	91%	91%	87%
Offline+Online	21%	21%	17%

Table 7.1: Action agreement: BC, offline and offline+online learning methods

Analysis: The action agreement percentage for 10 resource environment are shown in Table 7.1. The BC policy, trained with SJF_QoS and Combo datasets, has the highest action agreement (99% and 98% respectively), confirming that the BC learning method directly mimics the underlying policy in the dataset. When the replay buffer dataset consists of multiple policies (BC_qos_sjf and BC_combo), it is more likely that BC will mimic the actions of one of the underlying policies attributing to the high action agreement. When trained using a dataset with only QoS policy, BC_qos scheduler agrees with the QoS policy 95% of the time. The remaining 5% of the time BC is likely generating random actions. On the other hand, the offline RL scheduler demonstrates a lower action agreement with the policy in the dataset. When trained using a dataset with QoS and SJF_QoS policies, the offline RL agent agrees 91% of the time. The offline RL agent's action agreement drops to 87% with combo datasets confirming that offline RL is resilient to low-quality datasets and extracts good policies even when the noise-to-signal ratio is high. The offline+online method, pre-trained with offline data and using online interactions for improvement, shows 21% action agreement when pre-trained with QoS and SJF_QoS and 17% with combo data. This low action agreement indicates that the offline+online method learns policies significantly different from the underlying heuristic policies leading to better performance.

Data Quality	BC	Offline	Offline+Online	Online
High	✓	✓	✓	NA
Medium	×	✓	✓	NA
Low	×	×	✓	NA
No data	×	×	×	✓

Table 7.2: Choosing between BC, Offline and Offline+Online learning methods

Finally, Table 7.2 presents our recommendations on which RL method is suitable based on the quality of prior datasets and learning environments. Suppose the practitioners have access to high-quality datasets (expert demonstrations) and want to extract the general underlying policy. In that case, BC is a better choice, although offline methods will also perform equally well. One caveat is that if the expert demos are generated by a highly customized policy for a specific domain, then BC might fail to generalize well in a modified domain. If the prior datasets are of mixed quality (expert demonstrations mixed with suboptimal transitions), then offline and offline+online RL methods are suitable. Offline RL is more resilient to noise but needs some good transitions closer to high-reward areas.

Similarly, the offline+online method is resilient to noisy datasets, but this method also has the advantage of interacting with the environment to improve the pre-trained policy. Therefore, the offline+online method can be used even when the noise-to-signal ratio in the prior dataset is high. The agent (pre-trained with a noisy dataset) will eventually learn good policies after some interactions with the environment. Finally, if practitioners have access to prior datasets or the learning environment is novel (e.g., to learn new actions to avoid QoS violations in green datacenters), then the online RL method is most suitable. In terms of training time, BC training is generally faster because the supervised actor update is easier to learn than the Q-function (offline learning). For the same reason, offline learning is faster than offline+online learning. We observed that, for our problem setting, online learning took the longest to learn reasonable policies, given all other parameters being the same.

7.6 Future Work

In future work, we would like to empirically analyze the performance of offline learning with diverse datasets generated from different workloads. The goal of data-driven methods is to train a model that attains good performance on datasets coming from the same distribution as the training data. In offline RL, the basic idea is to learn a policy that does something differently (likely better) from the behavior pattern observed in dataset $\mathcal{D}$. Distributional shift issues can be addressed in several ways, and we would like to explore this further. Additionally, we would like to analyze the effect of the size of the prior datasets used in offline learning and training duration to achieve the best performance for each method and performance with various workloads.

7.7 Chapter Summary

Job scheduling using online RL methods has demonstrated efficient scheduling strategies in datacenters. Unfortunately, online RL methods often take hundreds of thousands of timesteps to explore the environment and adapt from a randomly initialized DNN policy. Moreover, large amounts of historical data are readily available for the job scheduling domain. Effective data-driven RL methods can use prior datasets to pre-train offline while improving with online fine-tuning. Furthermore, designing reward functions that elicit desired behaviors in complex environments is challenging. The RL schedulers can leverage custom heuristic schedulers' designed specifically for unique workloads or environments to learn and improve overall performance.

This chapter emphasizes the importance of leveraging decades worth of research on heuristic scheduling policies and recent developments in data-driven RL methods. We show the use of transitions from heuristic methods to learn a policy and improve it further without spending excessive time and computational effort using online learning alone. Specifically, combining the Offline+Online method speeds up learning while leveraging existing heuristic/expert demonstrators. We demonstrate how to generate datasets of varying qualities (by combining transitions from good and not-so-good heuristic policies) applicable to the job scheduling problem.

We explored two data-driven RL methods, namely 1) Behaviour Cloning and 2) Offline RL, which aim to learn policies from logged data without further interaction with the environment. These methods address the challenges concerning the data collection costs and safety, particularly pertinent to real-world applications of RL. Although offline learning methods produce good results, we showed that the performance is highly dependent on the quality of the historical datasets. Finally, we utilize offline RL as a **launchpad** to learn effective scheduling policies from a prior dataset collected using expert demonstrations or heuristic policies. By effectively incorporating prior datasets to pre-train the RL agent, we short-circuit the random exploration phase to learn reasonable policies with online learning. The contribution of this study is not just another RL scheduler but a systematic study of what makes sense - standard offline RL, pure online RL methods, or offline pre-training with online fine-tuning.

Chapter 8

Conclusions

This dissertation focuses on designing and evaluating RL-based schedulers that effectively adapt to dynamic conditions and learn strategies that meet the specific objectives of power-modulated datacenters.

The sustained demand for digital services has led to record datacenter build-outs and increased energy consumption. Datacenters in the U.S. consume 1.8% of the total electricity; electricity predominantly generated using non-renewable sources, emitting an estimated $\sim$ 230 Million Metric tons of greenhouse gases every year. Given high carbon emissions and growing societal awareness of climate change, government agencies, non-profits, and the general public demand cleaner (greener) goods and services. Consequently, cloud service providers are heavily investing in green datacenters, i.e., partially or entirely powered by renewable energy, either by self-generation or co-location.

The difficulty with using renewables to power datacenters is intermittent power supply, accompanied by inaccuracies in power predictions. The degree of inaccuracy varies from one renewable energy source to another, requiring smart systems and system software to carefully balance and intelligently adapt computing to energy generation. Traditional heuristics-based job schedulers [14] [15] [16] use hand-crafted scheduling policies suitable for datacenters with constant power supply. Hand-engineering domain-specific heuristics-based schedulers to meet specific objective functions of complex dynamic green datacenters is time-consuming, error-prone, and requires domain expertise.

Reinforcement Learning has solved sequential decision tasks of impressive difficulty [18] [19] [20] [21] by maximizing reward functions through trial and error. The growing body of research [22] [23] [24] have shown RL schedulers can learn effective job scheduling policies in traditional datacenter environments with constant power supply.

8.1 Summary of Contributions

In this dissertation, **we hypothesized that Reinforcement Learning based schedulers perform better than heuristics schedulers for power-aware distributed scheduling in datacenters. RL schedulers adapt to varying conditions and learn strategies that meet the specific objectives set forth by the datacenter operators**. These objectives include single-criteria optimization, constrained optimization with opposing goals, or multi-criteria optimization.

To address the shortcomings of existing RL schedulers, this dissertation proposed a unified green datacenter simulator that allows experimenting with synthetic and real workloads and integrates various renewable energy sources along with Energy Storage Devices (batteries). We implemented a discrete event simulator to simulate the green datacenter environment. Our simulator aims to evaluate and compare the RL scheduler's performance under various operating conditions, i.e., power availability levels (synthetic, constant, and intermittent power supply), varying system loads, and different workloads (synthetic and real).

We showed that our RL scheduler, RARE [118] performs better than heuristics-based algorithms in the dynamic green datacenter environment for synthetic and real HPC workloads for a cluster of up to 1200 resources. The RL scheduler adapts exceptionally well to the intermittent power supply (synthetic and actual power prediction data). We demonstrated that accurately tuning the system parameters like planning horizon and proper DNN configurations leads to increased performance.

This dissertation identifies that current RL schedulers do not optimize for multiple objectives simultaneously. We presented a Constraint-controlled RL scheduler, CoCoRL [155], that automatically learns conflicting reward and cost functions. We applied proportional-integral-derivative Lagrangian methods in Deep Reinforcement Learning to job scheduling problems in the green datacenter environment. We showed that a Constraint-controlled RL scheduler learns policies that satisfy the conflicting constraints and optimize for multiple objectives such as resource utilization and job completion. Our experiments demonstrate the CoCoRL scheduler's performance for both the primary objective (maximizing total job value) and the secondary objective (minimizing overall job delay). The CoCoRL scheduler yields significantly higher total job value while exhibiting comparable job completion and system utilization ratio than baseline heuristic scheduling policies. We showed that our CoCoRL scheduler efficiently adapts to HPC workloads and intermittent power supply (solar and wind). Additionally, we showed the significance of accurately tuning hyperparameters to satisfy various optimization goals set by datacenter operators.

This dissertation investigates data-driven RL methods, namely 1) Behaviour Cloning (BC) and 2) Offline RL (historically known as batch RL), which aim to learn policies from logged data without further interaction with the environment. These methods address the challenges concerning the cost of data collection and safety, particularly pertinent to real-world applications of RL. Although the BC learning approach generates good results, we showed that the performance is highly dependent on the quality of the historical data. We explored Offline RL as a **launchpad** [156] to learn effective scheduling policies from prior experience collected using expert demonstrations or heuristic policies. Finally, we demonstrated that by effectively incorporating prior datasets to pre-train the agent, we short-circuit the random exploration phase to learn a reasonable policy with online training. Such a framework is effective for pre-training from off-policy datasets and well suited to continuous improvement with online data collection.

8.2 Limitations and Future Work

While this dissertation has taken significant steps toward exploring RL schedulers for green datacenters, the presented methods have some limitations that are worth exploring in the future. In this section, we first review these limitations and potential solutions to extend the proposed solutions and then highlight some research directions toward production-ready RL schedulers. Below are some of the limitations of our proposed work and also existing work:

8.2.1 State space explosion

The datacenter environment described so far is for a single-agent reinforcement learning scheduler, i.e., the case where the central resource manager is the scheduling agent directly sensing the state and scheduling jobs based on the resource availability in a single datacenter. The advantages of a central agent are: 1) The agent has a global view of the state space and hence can optimize the schedule to maximize the objective function, and 2) There is no communication overhead in terms of coordinating with other agents. The disadvantage is that the state space can become combinatorial, and learning can take a long to converge to a near-optimal policy. Furthermore, as the problem size increases, so do the model sizes, which have memory and storage implications. Training these exceedingly large models may not be feasible on limited hardware resources available to researchers.

8.2.2 Standardized environments and models

Researchers have proposed many RL-based schedulers for a variety of systems. Even within a single datacenter, RL schedulers are specialized for various workloads, including machine learning training jobs [22], DAG jobs [97], HPC jobs [24] and many more. While other RL schedulers focus on energy efficiency [23], application profiling and monitoring [98], and delayed scheduling [99] based on the availability of power. Several other implementations use dissimilar representations of states, actions, rewards, and environments. The proliferation of RL schedulers with different representations and models renders it hard, if not impossible, to make a fair one-to-one comparison between various implementations.

8.2.3 Future work

For future work, we have short-term and long-term research goals. First, explore multi-criteria optimization using Model Predictive Control (MPC) methods, similar to PID Lagrangian methods discussed in Chapter-6. In this research, multi-criteria optimization includes jointly optimizing for Quality of Service (QoS), Fairness, and Energy Management in green datacenters. Second, interpret the learned models for various workloads and optimization objectives. Using this understanding, develop generic RL scheduler models that work well for a wide range of workloads, optimization objectives, and operating conditions. Third, separating the resource selection and job selection into two agents acting in tandem to alleviate, to some degree, the state-space explosion problem.

Our long-term research goals are 1) addressing and mitigating the state-space explosion problem and 2) standardizing the datacenter environments.

To mitigate the state-space explosion problem, we would like to explore two directions, namely 1) multi-agent distributed scheduler and 2) hierarchical scheduler. First, using Multi-agent RL [157] methods, a central manager distributes the scheduling control to local agents. A local agent may dynamically decide to run the jobs or cooperatively forward unallocated jobs to another agent to optimize the global utility of the system. There are two main settings in MARL: 1) Cooperative; and 2) Non-cooperative. In cooperative settings, the agents can share full or partial information about the state space they perceive. In a non-cooperative setting, local agents may not share their local states with peer agents. Using multi-agent systems provides many advantages compared to a centralized solution with a single agent. The advantages are the ability to distribute the required computations over several entities, increased robustness, and scalability. The disadvantage of choosing a distributed approach is that agents may make potentially suboptimal decisions (in that environment becomes partially observable) until they learn better policies producing only an approximate joint policy. The communication overhead may also affect the final schedule, and the agents must deal with communication delays and failures.

Second, hierarchical reinforcement learning aims to discover and exploit hierarchical structure in a given Markov decision problem. The assumption is that the tasks can be divided into terminating subroutines that the agents at each sub-level can call. The root-level agent will receive the reward by aggregating the agents' rewards immediately below it. Those agents will receive rewards based on the policies followed by agents at the next level and so forth. The learning agents have a partial view of the environment (Partially Observable MDP or POMDP [158]), and the agents communicate their state with other agents to maximize overall reward. When the learning has finished, the policy for each sub-routine will be an optimal solution to a sub-MDP of the original MDP. The policy of the overall MDP will be a combination of the policies of the various subroutines. An essential benefit of this approach is that these sub-MDPs (and the learned policies) can be reusable in new tasks.

By dividing the state space among multiple agents, we can alleviate both problems. Each agent can learn policies on smaller state space, reducing the training time, and the smaller models will fit on existing hardware available to the researchers.

To address the need for standardized environments, we propose homogenizing different datacenter implementations, similar to OpenAI Gym [20], and provide customization options in the datacenter environment for experimentation. Additionally, we would extend the datacenter environment to simulate other essential infrastructures, including networking, storage, and their associated overhead. This standardized datacenter environment will provide a solid foundation for the systems research community to conduct experiments and compare results with other researchers. Ideally, this effort will facilitate collaboration between systems researchers and build effective RL schedulers for real-world datacenters.

Appendices

Appendix A

A.1 Markov Process

Markov Process is a memory-less random process. That is a sequence of random states S_1, S_2, and so on, with the Markov property. A Markov process, a.k.a Markov chain, is a tuple (S, P), state space S, and transition function P. The dynamics of the MDP can be defined by these two components, S, and P. When we sample from an MDP, the sample is a sequence of states, a.k.a an episode. A summary of notations used throughout this dissertation can be found in Table A.1.

A.1.1 Markov Reward Process (MRP)

MRP is a Markov process with value judgment, i.e., how much reward can be accumulated by following a particular sampled sequence. An MRP is a tuple (S, P, R, γ), S is the state space, P is the state transition probability function, and R is a reward function,

$$R_s = E[R_{t+1}|S_t = s] \tag{A.1}$$

i.e., the expected immediate reward we get from state S at that moment. The total discounted reward, G_t, is given below, from time step t. The goal is to maximize this return,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{0}^{\infty} \gamma k R_{t+k+1} \tag{A.2}$$

γ is a discount factor, $\gamma \epsilon [0, 1]$.

The discount factor decides how much the agent should care about rewards now to future rewards. If ($\gamma = 0$), the agent only cares about the first reward (short-sighted). If ($\gamma = 1$), the agent cares about all future rewards (far-sighted). The discount factor γ is mathematically convenient because we can guarantee that the algorithm will converge and avoid infinite returns (with loops) in Markov processes. Another reason to discount rewards is that the agent is not certain about the future. The agent may be better off taking the immediate reward rather than waiting to get a larger reward later in the future. So, γ defines a form of the finite horizon for a specific duration. Intuitively, it encodes the human cognitive model, which shows preferences for immediate rewards.

The agent tries to get the maximum expected sum of rewards from every state in which it ends up. To achieve maximum rewards, the agent must use the optimal value function, i.e., the maximum sum of cumulative rewards. The optimal value function is solved using Bellman equations. For brevity, we are not explaining the formalism of Bellman equations here. Refer [138] for a detailed overview of Bellman equations.

We have the value function for MRPs, but there are no decisions that an agent can choose from. With a policy, a rule for choosing an action in a Markov process, the agent has a choice for picking actions. Formally, a policy is a mapping from the set of states S to the set of actions A.

A.1.2 Markov Decision Process (MDP)

MDP is a Markov Reward Process with decisions. An MDP is a tuple (S, A, P, R, γ), where S is the state space, A is a set of actions (finite), P is the state transition probability function given by,

$$P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a] \tag{A.3}$$

R is the reward function given by, and γ is a discount factor $\gamma \epsilon [0, 1]$. A rule for choosing actions in any state is called a policy. Formally, a policy is a mapping π from the set of states S to the set of actions A. An agent's policy π is a distribution over actions given states. A policy completely defines the behavior of an agent,

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{A.4}$$

If an agent follows a specific policy over many trials, the average reward the agent receives is the value of that policy. Along with computing the value of a policy averaged over many trials, we can also compute the value of a policy starting in a particular state s. The value of a policy is denoted $V_\pi(s)$. This is the expected cumulative reward of executing policy π starting in state s, and we can write it as

$$V_\pi(s) = E[r_{t+1} + r_{t+2} + ...|S_t = s, \pi] \tag{A.5}$$

where r_t is the reward, s_t is the state at time t. Finally, the expectation is taken over the stochastic results of actions in the environment. An MDP can have more than one optimal policy denoted by π_* that maximizes the expected value of the policy. All the optimal policies share the same optimal value function, V^*. The optimal value function satisfies the *Bellman equation*,

$$V_*(s) = max_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + V_*(s')] \tag{A.6}$$

where a denotes an action in state s, s' is the resulting state reached as per the transition probability $P(s'|s,a)$, $R(s'|s,a)$ denotes the expected one-step reward for performing action a and moving from state s to s. $V_*(s')$ is the value of the resulting state. The sum on the right-hand side is the expected value of the one-step reward $R(s'|s,a)$ plus the value of the next state s. We can think of it as the backed-up value of a one-step lookahead search and choosing the action with the best backed-up value with max_a. The sum is so important that it is given a special name $Q_*(s,a)$,

$$Q_*(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + V_*(s')] \tag{A.7}$$

$Q_*(s,a)$ is the expected total reward received when the agent performs action a in state s and then behaves optimally after that. Substituting this into the Bellman equation, we see that the value function is just the maximum (overall actions) of the Q-function,

$$V_*(s) = max_a Q_*(s,a) \tag{A.8}$$

Substituting this into the Q-equation to obtain the Q version of the Bellman equation:

$$Q_*(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + max_{a'} Q_*(s',a')] \tag{A.9}$$

The problem of RL is to compute the optimal policy, given no prior knowledge about the MDP, by interacting with the MDP. The RL agent can observe state s, try action a, and observe the resulting state s and the reward r by interacting with the MDP. By accumulating information over many interactions, the agent can form an estimate of the probability transition function $(\hat{P}(s'|s,a))$ and the expected one-step reward function $(\hat{R}(s'|s,a))$. An alternative is to construct an estimate of V_* or Q_* directly, without learning the probability transition function $\hat{P}$ and the expected one-step reward function $\hat{R}$ first using Q-learning algorithm. Let $Q_t(s,a)$ be the current estimate of the optimal Q-function at time t. At each time step t, the agent observes the current state s, chooses action a according to policy π_x, resulting in state s' and the one-step reward r, and updates as follows:

$$Q_{t+1}(s,a) = (1-\alpha)Q_t(s,a) + \alpha[r + max_{a'} Q_t(s',a')] \tag{A.10}$$

The parameter α is a learning rate (between 0 and 1). The expression on the right-hand side is computing a moving average between the previous value of $Q(s,a)$ and a new "estimated value" resulting from the current experience. If α gradually decreases according to certain standard conditions, and if π_x ensures that every action is executed infinitely often in every state, then with probability 1, Q_t converges to Q_*. It is important to note that the action a can be very simple or very complex, and this algorithm will still work. Indeed, action a can be a call to a subroutine that takes many primitive actions and then exits. When that subroutine exits, it will leave the environment in some new state s'. If we define r as the total reward received while the subroutine a was being executed, then the Bellman equation is still satisfied, and Q-learning will still converge to Q_*. Technically, this variant of Q-learning is called semi-Markov Q-learning (or SMDP Q-learning) because an MDP in which actions can take multiple timesteps is known as a semi-Markov decision problem.

A general setting shown in Figure 3.1 represents an agent interacting with the environment. At each time step t, the agent observes system state s_t, and it should choose an action a_t. For the chosen action, the state of the environment transitions from s_t to s_{t+1}, and the agent receives a reward r_t for that action. The state transitions and corresponding rewards are stochastic and have the Markov property - the state transition probabilities and rewards depend on the state of the environment s_t and the action a_t taken by the agent.

The agent can control only its actions, not the reward, after taking action. During training, the agent interacts with the environment and observes quantities of the system for various actions. The agent's goal is to maximize the expected discounted reward expressed as,

$$E[\sum_{i=1}^{\infty} \gamma_t r^t] \tag{A.11}$$

where $\gamma \epsilon (0, 1]$ is a factor discounting future rewards. The discounting factor specifies how important future rewards are with respect to the current state. If the reward r occurs n steps in the future from the present state, then the reward is multiplied by γ^n to describe its importance to the present state.

Policy space: The act of selecting an action in each state is called "policy" and is denoted as π. The agent selects the next actions based on a policy (π). Policy is a probability distribution over actions $\pi : \pi(s, a) \rightarrow [0, 1]$. Thus $\pi(s, a)$ is the probability that an action a is taken in state s. There are many possible (s, a) pairs, exponential in our case. Therefore, it is not practical to store the policy in vector format. Instead, we use function approximators. A function approximator has considerably fewer parameters,

θ represented as $\pi_\theta(s,a)$. The idea is that by approximating the policy, the agent would take similar actions for similar or close-by states.

Gradient Descent and Policy gradients: The class of RL algorithms that learn by performing gradient descent on the policy parameters is the focus of this dissertation. The policy-gradient method gives the directions that the parameters should be adjusted in order to improve a given policy's performance. The process of training RL agents is just optimizing the objective function where the objective is to get the maximum expected cumulative discounted reward (given by the above equation) by taking the gradient of the objective function.

$$\bigtriangledown_\theta E_{\pi_\theta}[\sum_0^t r_t] = \sum_0^t \log_{\pi_\theta}(s,a) Q_{\pi\theta}(s,a) \tag{A.12}$$

Here, $Q_{\pi_\theta}(s,a)$ is the expected cumulative discounted reward from choosing action a in state s and subsequently following policy π_θ. The main idea of policy gradient methods is to estimate the gradient by observing the trajectories of executions obtained after following a policy. The agent samples multiple trajectories and uses the cumulative discounted reward, v_t, as an unbiased estimate of $Q_{\pi_\theta}(s_t,a_t)$. The agent then iteratively updates the policy parameters in the direction of the gradient.

A.2 Preliminaries - Constrained Controlled PID Lagrangian Method

This section briefly introduces CMDP and is reproduced here from the literature.

Constrained Reinforcement Learning: The Constrained Markov Decision Processes (CMDP) [120] extend MDPs [122] to include constraints into RL. A CMDP is the extended tuple $(S, A, R, T, \mu, C_0, C_1, \ldots, d_0, d_1, \ldots)$. The cost functions $C_i : S \times A \times S \to \mathbb{R}$ defined with the same form as the reward functions, and $d_i : \mathbb{R}$ representing cost limits. For this work, we will consider a single, all-encompassing cost.

In RL, the expected sum of discounted rewards computed over $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots)$

trajectories, using the policy $\pi(a|s)$ is a common performance objective: $J(\pi) = E_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t R(s_t, a_t, s_{t+1})]$. The corresponding value function for the cost is defined as $J_C(\pi) = E_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t C(s_t, a_t, s_{t+1})]$. Then the constrained RL problem is solving for the best possible policy:

$$\pi^* = \arg\max_{\pi} J(\pi) \quad \text{s.t. } J_C(\pi) \leq d \tag{A.13}$$

Deep RL uses a Deep Neural Network (DNN) for the policy, $\pi_\theta = \pi(\cdot|s;\theta)$ with θ as a parameter vector. The policy gradient algorithms improve the policy over time by gathering experience in the task of estimating the reward objective gradient, $\nabla_\theta J(\pi_\theta)$ iteratively. Therefore our constrained optimization problem is expressed as maximizing score at some iterate, π_k, ideally obeying constraints at each iteration:

$$\begin{gathered} \max_{\pi} J(\pi_k) \;\; \text{s.t. } J_C(\pi_m) \leq d \\ \text{where } m \in \{0, 1, \ldots, k\} \end{gathered} \tag{A.14}$$

Optimal Control and Dynamical Systems: Dynamical systems are processes that are subject to external control. A generic formulation for discrete-time systems with feedback control is:

$$\begin{aligned} \mathrm{x}_{k+1} &= F(\mathrm{x}_k, u_k) \\ \mathrm{y}_k &= Z(\mathrm{x}_k) \\ u_k &= h(\mathrm{y}_0, \mathrm{y}_1, \ldots, \mathrm{y}_k) \end{aligned} \tag{A.15}$$

Where x is the state vector, F is the dynamics function, u applied control, y is the measurement outputs, and the subscript denotes the time step. The feedback rule, h, can access past and present measurements. The optimal control problem is to design a control rule, h, that results in a sequence $\mathrm{y}_{0:T} = \{\mathrm{y}_0, \mathrm{y}_1, \ldots, \mathrm{y}_T\}$ (or states $\mathrm{x}_{0:T}$) that scores well for some cost function C. For instance, reaching a goal condition, $C = |\mathrm{y}_T - \bar{\mathrm{y}}|$, or closely following a desired trajectory, $\bar{\mathrm{y}}_{0:T}$.

In general, systems with simpler dependency on the input are easier to analyze and control (i.e., simpler h performs well), even though the dependence on the state is complicated [159].

$$F(\mathrm{x}_k, u_k) = f(\mathrm{x}_k) + g(\mathrm{x}_k)\ u_k \tag{A.16}$$

where f and g can be nonlinear in the state and may be uncertain or unknown.

Lagrangian Methods for Constrained Optimization: Lagrangian methods are a classic family of approaches to solving constrained optimization problems. The work in [160] analyzed the dynamics of a continuous-time neural learning system applied to this problem. The authors start with the component-wise differential equations:

$$\dot{x}_i = -\frac{\partial \mathcal{L}(\mathrm{x}, \lambda)}{\partial x_i} = -\frac{\partial f}{\partial x_i} - \lambda \frac{\partial g}{\partial x_i} \tag{A.17}$$

$$\dot{\lambda} = \alpha \frac{\partial \mathcal{L}(\mathrm{x}, \lambda)}{\partial \lambda} = \alpha g(\mathrm{x}) \tag{A.18}$$

where the scalar constant α is the learning rate on λ. Differentiating eq A.17 and substituting with eq A.18 leads to the second-order dynamics, written in vector format:

$$\ddot{\mathrm{x}} + A\dot{\mathrm{x}} + \alpha g(\mathrm{x}) \nabla g = 0 \tag{A.19}$$

is a forced oscillator with a damping matrix:

$$\begin{gathered} A_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} + \lambda \frac{\partial^2 g}{\partial x_i \partial x_j} \\ \text{or, } A = \nabla^2 f + \lambda \nabla^2 g \end{gathered} \tag{A.20}$$

The work in [160] showed that if A is positive definite, the system eq A.19 converges to a solution that satisfies the constraint. Also, [160] noted that the system eq A.17 and eq A.18 is prone to oscillations, with frequency and settling time depending on α.

In eq A.18, λ merely integrates the constraint. To encourage the dynamics towards fast and stable constraint satisfaction, a new term is introduced [123] in λ that is proportional to the current constraint value. In the differential equation for λ, this term appears as the time-derivative of the constraint:

$$\dot{\lambda} = \alpha g(\mathrm{x}) + \beta \dot{g}(\mathrm{x}) = \alpha g(\mathrm{x}) + \beta \sum_j \frac{\partial g}{\partial x_j} \dot{x}_j \tag{A.21}$$

where β is the strength coefficient. Replacing eq A.18 by eq A.21 and combining with eq A.17 yields similar second-order dynamics as eq A.19 with an additional term in the damping matrix:

$$\ddot{\mathrm{x}} + (A + \beta \nabla g \nabla^T g)\dot{\mathrm{x}} + \alpha g(\mathrm{x}) \nabla g = 0 \tag{A.22}$$

The new term is beneficial because it is positive semi-definite - the outer product of a vector with itself - thereby increasing the damping eigenvalues and boosting convergence.

A similar analysis [123] applies to the addition of a term in λ based on the derivative of the constraint value. It appears in $\dot{\lambda}$ as the second derivative of the constraint:

$$\dot{\lambda} = \alpha g(\mathrm{x}) + \gamma \ddot{g}(\mathrm{x}) \tag{A.23}$$

where γ is the strength coefficient. The resulting dynamics are:

$$\ddot{\mathrm{x}} + B^{-1} A \dot{\mathrm{x}} + (\alpha g(\mathrm{x}) + \gamma \dot{\mathrm{x}}^T \nabla^2 g \dot{x}) B^{-1} \nabla g = 0 \tag{A.24}$$

with $B = I + \gamma \nabla g \nabla^T g$, and I the identity matrix.

The combination of the previous two developments gives the Proportional-Integral-Derivative (PID) multiplier method inducing independent changes in the dynamics (i.e., insert the damping matrix of eq A.22 into eq A.24). The PID multiplier method proposed in [123] is a recent result where a PID update rule is considered for a learned Lagrange multiplier.

A.3 Hyperparameters

Many RL papers investigate in-depth learning paradigms. Nevertheless, it is less visible that behind successful experiments in deep RL, complicated code bases contain many low-

and high-level design decisions that are usually not discussed in research papers. While one may assume that such “choices” do not matter, there is some evidence that they are, in fact, crucial for or even driving good performance [161] [162] [109].

While there are open-source RL algorithm implementations available that practitioners can use, this is still unsatisfactory because often, different algorithms implemented in different code bases are compared one-to-one. This makes it impossible to assess whether improvements are due to the algorithms or their implementations. Furthermore, without an understanding of lower-level choices, it is hard to assess the performance of high-level algorithmic choices, as performance may strongly depend on tuning hyperparameters and implementation-level details. Our goal in this section is to provide such lower-level choices we used during our experimentation.

Notation	Description
s	State
a	Action
S	Set of all non-terminal states
$S+$	Set of all states, including the terminal state
$A(s)$	Set of actions possible in state s
R	Set of possible rewards
C	Cost function S_t
State at t	
A_t	Action at t
R_t	Reward at t, dependent, like S_t, on A_{t-1} and S_{t-1}
G_t	Return (cumulative discounted reward) following t
π	Policy, decision-making rule
$\pi(s)$	Action taken in states under deterministic policy π
$\pi(a\|s)$	Probability of taking action a in states under stochastic policy π
$p(s', r\|s, a)$	Probability of transitioning to state s', with reward r from s, a
$v_\pi(s)$	Value of states under policy π (expected return)
$v_\pi(s)$	Value of states under the optimal policy
$q_\pi(s, a)$	Value of taking action a in states under policy π
$q^*(s, a)$	Value of taking action a in states under the optimal policy
$V_t(s)$	Estimate (a random variable) of $v_\pi(s)$ or $v_\pi(s)$
$Q_t(s, a)$	Estimate (a random variable) of $q_\pi(s, a)$ or $q_*(s, a)$
γ	Discount-rate parameter
ϵ	Probability of random action in ϵ-greedy policy
α, β	Step-size parameters
λ	Decay-rate parameter for eligibility traces
$\mathbf{T}$	Datacenter simulator environment dynamics
$\mathcal{D}$	Replay buffer
$\mathcal{L}$	Lagrangian
d	Cost Limit
B	Randomly sampled batch
g_ψ	Encoder network
π_θ	Actor network
Q_ϕ	Critic network
$\mathcal{L}_{critic}$	Critic loss
$\mathcal{L}_{actor}$	Actor loss
f and g	linear or non-linear functions depending on the context

Table A.1: Common notations used in RL literature

Hyperparameter	Value
Batch size	1024
NN hidden layer size	512
Actor learning rate	$1e-4$
Critic learning rate	$1e-4$
Encoder learning rate	$1e-4$
Discount, γ	0.99
Number of critics	1
PopArt	True
BC Warmup steps	20,000

Table A.2: Hyperparamaters used in Chapter-5.

Hyperparameter	Value
NN hidden layer size	512
Hidden Layers	2
LSTM size	512
Learning rate	1×10^{-3}
NN Nonlinearity	tanh
Batch dimension, time	200
Batch dimension, envs	100
PPO ratio-clip	0.1
Discount, γ	0.99
λ_{GAE}	0.93
Optimizer	Adam
Cost scaling	1/10
Exponential moving average, K_P	0.95
Exponential moving average, K_D	0.9
Difference iterates delay, K_D	15
Observation Normalization	True
Cost limit	anl=7k, sdsc_sp2=20k, sdsc_blue=23k, pik_iplex=100k, hpc2n=25k

Table A.3: Hyperparamaters used in Chapter-6.

Hyperparameter (Common)	Value
Batch size	1024
Replay buffer size	1,750,000 and 50,000
NN hidden layer size	512
Actor learning rate	$1e-4$
Critic learning rate	$1e-4$
Encoder learning rate	$1e-4$
Discount, γ	0.99
Behavior Cloning	
Number of critics	1
PopArt	False
BC Warmup steps	20,000
Offline RL	
Number of critics	2
PopArt	False
Number of offline steps	100,000
Offline+Online RL	
Number of critics	2
PopArt	True
BC Warmup steps	20,000
Number of offline steps	50,000
Number of online steps	500,000

Table A.4: Hyperparamaters used in Chapter-7.

Common parameters	Value
Kernel width	4
Horizontal stride	2
MLP hidden	256
Number of filters	32
Job array MLP hidden	128
Small encoder	
Extra convolutional layers	1
Job array MLP hidden	128
Medium encoder	
Extra convolutional layers	2
Job array MLP hidden	300
Large encoder	
Extra convolutional layers	3
Job array MLP hidden	400

Table A.5: Configuration for small, medium and large encoders used in Chapters-5 and 7.

Appendix B

Table B.1 shows the most relevant parameters to configure a power-modulated datacenter. These parameters can be changed to simulate various configurations during training and evaluation of various scheduling policies. For instance, increasing the total_resource_slots generates a cluster with more resources (e.g., 10 to 100 resources). The price per resource per unit of time can be decided using the res_price option. The default is 0.5 for both CPUs and GPUs, but this can be altered to cater to the practitioner's needs.

Similarly, various power sources can be selected by changing the power_avail_rate. A number between 0.0 to 1.0 is used to select a synthetic power setting. For example, power_avail_rate=0.9 means 90% power supply 90% of the time. To select real power data, set power_avail_rate=-1.0, and to select real power data plus battery, set power_avail_rate=-2.0.

To increase the system load, we change the job_arrival_rate. Selecting the type of workload is done by changing workload_type with possible choices of synthetic, ANL, SDSC-SP2, SDSC-Blue, PIK-IPLEX, and HPC2N. Other job-related properties include max_job_duration and max_job_resources. These parameters, provided by the user, dictate how long a job must run and the maximum number of resources any job can request.

To select various reward and cost types, set the reward_type and cost_type parameters accordingly. Currently, we have 15 different types of reward functions and 9 different types of cost functions. Similarly, other costs can be altered, such as waiting cost (waiting_cost) values.

The simu_len, max_jobs, and max_episode_length control the simulation duration, maximum numbers of jobs during that simulation run, and maximum episodic length for RL scheduler training. The time_horizon dictates how much the scheduler can look into the future to schedule the jobs. Typically, this parameter is set to 24, 36, 48, or 72 time units.

Field	Values	Description
total_resource_slots	10	number of units of each resource sub-types
max_job_resources	5	max resources each job can request
ready_pool	5	number of jobs in the ready pool
num_resource_types	2	number of resource types
res_price	0.5	cpu and gpu resource price
workload_type	“synthetic”	workload type, synthetic or hpc
is_suspend	“no”	run with or without job suspension option
job_arrival_rate	1.0	job arrival rate, system load
max_job_duration	30	max length/duration of each job
job_value_knob	0.1	for scaling down the job’s value
power_avail_rate	0.0	percentage of power supply, $1.0 = 100\%, 0.5 = 50\%$ power
reward_type	13	type of the reward function, ranges between 1-15
cost_type	9	type of the reward function, ranges between 1-9
longrunning_cost	-1	cost incurred for running too long
readypool_cost	-1	cost incurred for waiting in ready pool
waitpool_cost	-1	cost incurred for waiting in wait pool
suspend_cost	-1	cost incurred for suspending a job
is_newseed	True	select new seed for every run, during training and evaluation
run_eval	False	select between training or evaluation run
simu_len	10000	max simulation length
max_jobs	1000000	max number of jobs for this simulation
max_episode_length	10000000	max episode length
time_horizon	48	max time units for the simulation
waitpool_size	48×3	max size of the wait pool
suspendpool_size	48×3	max size of the suspend pool
end	“no_new_job”	status when the simulation ends
backfill_type	“FCFS”	order in which jobs get admitted from waitpool to readypool
classic_observation	False	select between image only or image + vector formats
heuristic_agent	False	select DRL or heuristic agent actions during evaluation

Table B.1: Simulator parameters to configure power-modulated datacenter with various properties.

Bibliography

[1] R. Bashroush, "Data center and ict energy consumption: A fact-check on "factchecking"," 2020. [Online]. Available: https://www.linkedin.com/pulse/data-center-ict-energy-consumption-fact-check-rabih-bashroush

[2] Google, "We're sourcing clean energy for a better future." [Online]. Available: https://www.google.com/about/datacenters/renewable/

[3] Facebook, "Facebook sustainability." [Online]. Available: https://sustainability.fb.com

[4] SB Energy, "Sb energy inks 942 mw ppa with google to power data center." [Online]. Available: https://www.prnewswire.com/news-releases/sb-energy-inks-942-mw-ppa-with-google-to-power-data-center-301664786.html

[5] "Lancium Inc." [Online]. Available: https://lancium.com

[6] Terascale. [Online]. Available: https://terrascale.org

[7] U. E. I. Administration, "Wind explained: Electricity generation from wind." [Online]. Available: https://www.eia.gov/energyexplained/wind/electricity-generation-from-wind.php

[8] J. Hamilton, "Cost of power in large-scale data centers," 2008. [Online]. Available: https://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers

[9] Arman Shehabi et.al, “United states data center energy usage report,” 2016. [Online]. Available: https://www.osti.gov/servlets/purl/1372902/

[10] Z. Song, X. Zhang, and C. Eriksson, “Data center energy and cost saving evaluation,” *Energy Procedia*, vol. 75, pp. 1255–1260, 2015, clean, Efficient and Affordable Energy for a Sustainable Future: The 7th International Conference on Applied Energy (ICAE2015). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1876610215009467

[11] Tryggvi Adalbjornsson, “Iceland’s data centers are booming—here’s why that’s a problem,” 2019. [Online]. Available: https://www.technologyreview.com/s/613779/icelands-data-centers-are-booming-\heres-why-thats-a-problem/

[12] O. of ENERGY EFFICIENCY RENEWABLE ENERGY, “Advantages and challenges of wind energy.” [Online]. Available: https://www.energy.gov/eere/wind/advantages-and-challenges-wind-energy

[13] ——, “Wind vision: A new era for wind power in the united states.” [Online]. Available: https://www.energy.gov/eere/wind/wind-vision

[14] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, “Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters,” in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901355

[15] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, “Gandiva: Introspective cluster scheduling for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011.

[17] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015.

[18] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," ser. ICML'16, 2016.

[19] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castañeda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, N. Sonnerat, T. Green, L. Deason, J. Z. Leibo, D. Silver, D. Hassabis, K. Kavukcuoglu, and T. Graepel, "Human-level performance in 3d multiplayer games with population-based reinforcement learning," *Science*, 2019.

[20] (Accessed 2022) Openai. [Online]. Available: https://openai.com/five/

[21] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research. PMLR, 09–15 Jun 2019.

[22] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 1676–1684. [Online]. Available: http://proceedings.mlr.press/v80/gao18a.html

[23] Y. Ran, H. Hu, X. Zhou, and Y. Wen, "Deepee: Joint optimization of job scheduling and cooling control for data center energy efficiency using deep reinforcement learn-

ing," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 645–655.

[24] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "Rlscheduler: An automated hpc batch job scheduler using reinforcement learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'20. IEEE Press, 2020.

[25] T. Casavant and J. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[26] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, p. 1114–1120.

[27] J. Ullman, "Np-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, 1975. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000075800080

[28] C.-C. Shen and W.-H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Transactions on Computers*, vol. C-34, no. 3, pp. 197–203, 1985.

[29] "A* search algorithm." [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[30] V. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384–1397, 1988.

[31] J. A. Stankovic and I. S. Sidhu, "An adaptive bidding algorithm for processes, clusters and distributed groups," in *Proceedings of the 4th International Conference on*

Distributed Computing Systems, San Francisco, California, USA, May 14-18, 1984. IEEE Computer Society, 1984, pp. 49–59.

[32] J. Stankovic, "Stability and distributed scheduling algorithms," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1141–1152, 1985.

[33] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Networks (1976)*, vol. 8, no. 3, pp. 199–217, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0376507584900485

[34] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, vol. 2, 2005, pp. 776–783 Vol. 2.

[35] W. Emeneker, D. M. Jackson, J. Butikofer, and D. C. Stanzione, "Dynamic virtual clustering with xen and moab," in *ISPA Workshops*, 2006.

[36] D. B. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 87–102.

[37] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience: Research articles," *Concurr. Comput.: Pract. Exper.*, vol. 17, no. 2–4, p. 323–356, feb 2005.

[38] M. Litzkow, M. Livny, and M. Mutka, "Condor-a hunter of idle workstations," in *[1988] Proceedings. The 8th International Conference on Distributed*, 1988, pp. 104–111.

[39] "Moab." [Online]. Available: https://adaptivecomputing.com/cherry-services/moab-hpc

[40] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer-Verlag, 2002, pp. 44–60.

[41] "A taxonomy and survey of grid resource management systems for distributed computing," *Softw. Pract. Exper.*, vol. 32, no. 2, p. 135–164, feb 2002. [Online]. Available: https://doi.org/10.1002/spe.432

[42] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-g: a computation management agent for multi-institutional grids," in *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, 2001, pp. 55–63.

[43] I. T. Foster, "A globus toolkit primer," 2005.

[44] B. Nitzberg, J. M. Schopf, and J. P. Jones, "Pbs pro: Grid computing and scheduling attributes," 2004.

[45] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw, "The legion resource management system," 01 1999, pp. 162–178.

[46] F. Berman and R. Wolski, "Scheduling from the perspective of the application," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '96. USA: IEEE Computer Society, 1996, p. 100.

[47] I. T. Foster and C. Kesselman, "Globus: a metacomputing infrastructure toolkit," *International Journal of High Performance Computing Applications*, vol. 11, pp. 115 – 128, 1997.

[48] "Genesisii." [Online]. Available: https://genesis2.virginia.edu/wiki/uploads/Main/GenesisII_omnibus_reference_manual.htm

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the

data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 295–308.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523633

[51] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated slos for enterprise clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 117–134.

[52] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," ser. OSDI'16. USA: USENIX Association, 2016, p. 81–97.

[53] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 65–80.

[54] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901355

[55] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan, "Hydra: A federated resource manager for datacenter scale analytics," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 177–191.

[56] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 285–300.

[57] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *SIGPLAN Not.*, vol. 48, no. 4, p. 77–88, mar 2013. [Online]. Available: https://doi.org/10.1145/2499368.2451125

[58] ——, "Quasar: Resource-efficient and qos-aware cluster management," *SIGPLAN Not.*, vol. 49, no. 4, p. 127–144, feb 2014. [Online]. Available: https://doi.org/10.1145/2644865.2541941

[59] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. USA: USENIX Association, 2015, p. 499–510.

[60] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USA: USENIX Association, 2012, p. 2.

[61] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *Proceedings of the 12th USENIX*

Conference on Operating Systems Design and Implementation, ser. OSDI'16. USA: USENIX Association, 2016, p. 99–115.

[62] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of long running applications in shared production clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190549

[63] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190517

[64] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[65] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Kairos: Preemptive data center scheduling without runtime estimates," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 135–148. [Online]. Available: https://doi.org/10.1145/3267809.3267838

[66] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 595–610.

[67] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 485–500.

[68] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[69] Q. Sun, S. Ren, C. Wu, and Z. Li, "An online incentive mechanism for emergency demand response in geo-distributed colocation data centers," in *Proceedings of the Seventh International Conference on Future Energy Systems*, ser. e-Energy '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2934328.2934331

[70] I. n. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Greenhadoop: Leveraging green energy in data-processing frameworks," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2168836.2168843

[71] A. Krioukov, S. Alspaugh, P. Mohan, S. Dawson-Haggerty, D. E. Culler, and R. H. Katz, "Design and evaluation of an energy agile computing cluster," EECS Department, University of California, Berkeley, Tech. Rep., 2012. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-13.html

[72] I. n. Goiri, W. Katsak, K. Le, T. D. Nguyen, and R. Bianchini, "Parasol and greenswitch: Managing datacenters powered by renewable energy," *SIGPLAN Not.*, 2013.

[73] Í. Goiri, M. E. Haque, K. Le, R. Beauchea, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Matching renewable energy supply and demand in green datacenters," *Ad Hoc Networks*, vol. 25, pp. 520–534, 2015.

[74] A. D. Carnerero, D. R. Ramirez, D. Limon, and T. Alamo, "Particle based optimization for predictive energy efficient data center management," in *2020 59th IEEE Conference on Decision and Control (CDC)*, 2020.

[75] "Markov property." [Online]. Available: https://en.wikipedia.org/wiki/Markov_property

[76] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[77] D. Yarats, I. Kostrikov, and R. Fergus, "Image augmentation is all you need: Regularizing deep reinforcement learning from pixels," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=GY6-6sTvGaf

[78] P. Ambati, N. Bashir, D. Irwin, and P. Shenoy, "Waiting game: Optimally provisioning fixed resources for cloud-enabled schedulers," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[79] (Accessed May 2022) Amazon ec2 spot instances. [Online]. Available: https://aws.amazon.com/ec2/spot/

[80] (Accessed May 2022) Azure spot virtual machines. [Online]. Available: https://azure.microsoft.com/en-us/pricing/spot/

[81] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware cluster scheduling policies for deep learning workloads," in

14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, Nov. 2020.

[82] T. Jain and G. Cooperman, "Crac: Checkpoint-restart architecture for cuda with streams and uvm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[83] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Int. Res.*, vol. 4, no. 1, p. 237–285, may 1996.

[84] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, 09 2013.

[85] S. Mahadevan and G. Theocharous, "Optimizing production manufacturing using reinforcement learning," in *FLAIRS Conference*, 1998.

[86] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.

[87] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, ser. NIPS'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, p. 671–678.

[88] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 123–134. [Online]. Available: https://doi.org/10.1145/2486001.2486020

[89] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "Pcc: Re-architecting congestion control for consistent high performance," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. USA: USENIX Association, 2015, p. 395–408.

[90] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 191–205. [Online]. Available: https://doi.org/10.1145/3230543.3230551

[91] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: ACM, 2016, p. 50–56. [Online]. Available: https://doi.org/10.1145/3005745.3005750

[92] T. E. Thomas, J. Koo, S. Chaterji, and S. Bagchi, "Minerva: A reinforcement learning-based technique for optimal scheduling and bottleneck detection in distributed factory operations," in *2018 10th International Conference on Communication Systems Networks (COMSNETS)*, 2018, pp. 129–136.

[93] G. Domeniconi and E. K. Lee, "Cush: Cognitive scheduler for heterogeneous high performance computing system," in *Workshop on Deep Reinforcement Learning for Knowledge Discover, DRL4KDD*, 2019.

[94] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2019.

[95] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 505–513.

[96] D. Klusácek and H. Rudová, "Alea 2: job scheduling simulator," in *SimuTools*, 2010.

[97] Z. Hu, J. Tu, and B. Li, "Spear: Optimized dependency-aware task scheduling with deep reinforcement learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 2037–2046.

[98] A. Souza, K. Pelckmans, and J. Tordsson, "A hpc co-scheduler with reinforcement learning," in *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, W. Cirne, and G. P. Rodrigo, Eds. Springer International Publishing, 2021.

[99] X. Liu, Y. Hua, X. Liu, L. Yang, and Y. Sun, "Smoother: A smooth renewable power-aware middleware," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 249–260.

[100] ANL., "The argonne national laboratory intrepid log." 2009. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_anl_int/

[101] SDSC-SP2., "The san diego supercomputer center (sdsc) sp2 log." 2000. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_sp2/index.html

[102] SDSC-Blue., "The san diego supercomputer center (sdsc) blue horizon log." 2000-2003. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_blue/index.html

[103] PIK-IPLEX., "The potsdam institute for climate impact research (pik) ibm idataplex cluster log." 2009-2012. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_pik_iplex/index.html

[104] HPC2N., "The hpc2n seth log." 2002-2006. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/

[105] T. Jain and G. Cooperman, "Crac: Checkpoint-restart architecture for cuda with streams and uvm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[106] H. M. N. M. S. Muhammad Naveed Akhter, Saad Mekhilef, "Review on forecasting of photovoltaic power generation based on machine learning and meta heuristic techniques," 2018. [Online]. Available: https://doi.org/10.1049/iet-rpg.2018.5649

[107] "GLEAMM Facility at TTU." [Online]. Available: https://www.depts.ttu.edu/gleamm/

[108] Z. Salameh, "Battery capacity," 2014. [Online]. Available: https://www.sciencedirect.com/topics/\engineering/battery-capacity

[109] S. Fujimoto and S. Gu, "A minimalist approach to offline reinforcement learning," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=Q32U7dzWXpc

[110] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[111] P. Christodoulou, "Soft actor-critic for discrete action settings," *arXiv preprint arXiv:1910.07207*, 2019.

[112] Z. Wang, A. Novikov, K. Zolna, J. S. Merel, J. T. Springenberg, S. E. Reed, B. Shahriari, N. Siegel, C. Gulcehre, N. Heess *et al.*, "Critic regularized regression," *Advances in Neural Information Processing Systems*, vol. 33, pp. 7768–7778, 2020.

[113] "Computing resources, dept of computer science, uva," 2023. [Online]. Available: https://www.cs.virginia.edu/wiki/doku.php?id=compute_resources

[114] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, 2003.

[115] C. Jee, "Deepmind's ai is predicting how much energy google's wind turbines will produce," 2019. [Online]. Available: https://www.technologyreview.com/2019/02/27/239459/deepmind-creates-algorithm-\to-squeeze-more-out-of-wind-power/

[116] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to schedule long-running applications in shared container clusters at scale," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[117] V. Venkataswamy and A. Grimshaw, "Scheduling in data centers running on renewable energy with deep reinforcement learning," ser. 2nd workshop on Machine Learning for Computing Systems, hosted at Supercomputing'2020, 2020.

[118] V. Venkataswamy, J. Grigsby, A. Grimshaw, and Y. Qi, "Rare: Renewable energy aware resource management in datacenters," in *Job Scheduling Strategies for Parallel Processing*, 2022.

[119] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2008.

[120] E. Altman, "Constrained markov decision processes with total cost criteria: Lagrangian approach and dual linear program," *Mathematical Methods of Operations Research*, Dec. 1998.

[121] D. P. Bertsekas, "Constrained optimization and lagrange multiplier methods," 2014. [Online]. Available: https://www.mit.edu/~dimitrib/Constrained-Opt.pdf

[122] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning.* , Cambridge, MA, USA: MIT Press, 1988.

[123] A. Stooke, J. Achiam, and P. Abbeel, "Responsive safety in reinforcement learning by pid lagrangian methods," ser. ICML'20. JMLR.org, 2020.

[124] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: Association for Computing Machinery, 2007.

[125] T. Springer and E. Linstead, "Adaptive qos-based resource management framework for iot/edge computing," in *2018 9th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, 2018.

[126] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das, "Mete: Meeting end-to-end qos in multicores through system-wide resource management," *SIGMETRICS Perform. Eval. Rev.*, jun 2011.

[127] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control Real-Time scheduling: Framework, modeling, and algorithms*," *Real-Time Systems*, Jul. 2002.

[128] G. Zhou, W. Tian, and R. Buyya, "Deep reinforcement learning-based methods for resource scheduling in cloud computing: A review and future directions," *CoRR*, vol. abs/2105.04086, 2021. [Online]. Available: https://arxiv.org/abs/2105.04086

[129] J. García and F. Fernández, "A comprehensive survey on safe reinforcement learning," *J. Mach. Learn. Res.*, jan 2015.

[130] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[131] L. Xie, S. Wang, S. Rosa, A. Markham, and A. Trigoni, "Learning with training wheels: Speeding up training with a simple controller for deep reinforcement learning," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[132] P. Abbeel, A. Coates, and A. Ng, "Autonomous helicopter aerobatics through apprenticeship learning," *The International Journal of Robotics Research*, 2010.

[133] C. Gehring and D. Precup, "Smart exploration in reinforcement learning using absolute temporal difference errors," ser. AAMAS '13. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[134] H. Mao, M. Schwarzkopf, H. He, and M. Alizadeh, "Towards safe online reinforcement learning in computer systems," 2019.

[135] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[136] D. Muthirayan, M. Parvania, and P. P. Khargonekar, "Online algorithms for dynamic matching markets in power distribution systems," *IEEE Control Systems Letters*, 2021.

[137] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *Journal of Fluids Engineering*, 1942.

[138] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* London: MIT Press, 2018.

[139] T. Gabel, S. Lange, and M. Riedmiller, *Batch Reinforcement Learning*, 01 2017.

[140] D. A. Pomerleau, "Efficient training of artificial neural networks for autonomous navigation," *Neural Computation*, vol. 3, no. 1, pp. 88–97, 1991.

[141] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine, "D4{rl}: Datasets for deep data-driven reinforcement learning," 2021. [Online]. Available: https://openreview.net/forum?id=px0-N3_KjA

[142] P. Florence, C. Lynch, A. Zeng, O. A. Ramirez, A. Wahid, L. Downs, A. Wong, J. Lee, I. Mordatch, and J. Tompson, "Implicit behavioral cloning," in *5th Annual Conference on Robot Learning*, 2021. [Online]. Available: https://openreview.net/forum?id=rif3a5NAxU6

[143] M. Hahn, D. S. Chaplot, S. Tulsiani, M. Mukadam, J. M. Rehg, and A. Gupta, "No RL, no simulation: Learning to navigate without navigating," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=8vXYx6d8Wc

[144] A. Mandlekar, D. Xu, J. Wong, S. Nasiriany, C. Wang, R. Kulkarni, L. Fei-Fei, S. Savarese, Y. Zhu, and R. Martín-Martín, "What matters in learning from offline human demonstrations for robot manipulation," 2021. [Online]. Available: https://arxiv.org/abs/2108.03298

[145] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 1407–1416. [Online]. Available: https://proceedings.mlr.press/v80/espeholt18a.html

[146] A. Krishnakumar, S. E. Arda, A. A. Goksoy, S. K. Mandal, U. Y. Ogras, A. L. Sartor, and R. Marculescu, "Runtime task scheduling using imitation learning for heterogeneous many-core systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4064–4077, 2020.

[147] S. K. Mandal, G. Bhat, C. A. Patil, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "Dynamic resource management of heterogeneous mobile platforms via imitation learning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2842–2854, 2019.

[148] A. L. Sartor, A. Krishnakumar, S. E. Arda, U. Y. Ogras, and R. Marculescu, "Hilite: Hierarchical and lightweight imitation learning for power management of embedded socs," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 63–67, 2020.

[149] S. Fujimoto, H. Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in *International conference on machine learning.* PMLR, 2018, pp. 1587–1596.

[150] M. Hessel, H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. van Hasselt, “Multi-task deep reinforcement learning with popart,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3796–3803.

[151] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[152] A. Kumar, J. Fu, M. Soh, G. Tucker, and S. Levine, “Stabilizing off-policy q-learning via bootstrapping error reduction,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[153] A. Nair, M. Dalal, A. Gupta, and S. Levine, “Accelerating online reinforcement learning with offline datasets,” *arXiv preprint arXiv:2006.09359*, 2020.

[154] ——, “{AWAC}: Accelerating online reinforcement learning with offline datasets,” 2021. [Online]. Available: https://openreview.net/forum?id=OJiM1R3jAtZ

[155] V. Venkataswamy and A. Grimshaw, “Job scheduling in datacenters using constraint controlled rl,” 2022. [Online]. Available: https://arxiv.org/abs/2211.05338

[156] V. Venkataswamy, J. Grigsby, A. Grimshaw, and Y. Qi, “Launchpad: Learning to schedule using offline and online rl methods,” 2022. [Online]. Available: https://arxiv.org/abs/2212.00639

[157] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* USA: Cambridge University Press, 2008.

[158] T. Dietterich, “An overview of maxq hierarchical reinforcement learning,” vol. 1864, 06 2000.

[159] R. Skelton, *Dynamic Systems Control: Linear Systems Analysis and Synthesis. Dynamic Systems Control*. John Wiley Sons, 1988.

[160] J. C. Platt and A. H. Barr, "Constrained differential optimization," in *NIPS*, 1987.

[161] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep rl: A case study on ppo and trpo," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=r1etN1rtPB

[162] M. Andrychowicz, A. Raichuk, P. Stanczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, "What matters in on-policy reinforcement learning? A large-scale empirical study," *CoRR*, vol. abs/2006.05990, 2020. [Online]. Available: https://arxiv.org/abs/2006.05990