Spring 6-1-2023

Mixed-Criticality Scheduling Using Reinforcement Learning

Omar ElSeadawy
omarelseadawy@aucegypt.edu

Graduate Studies

Mixed-Criticality Scheduling using Reinforcement Learning

A THESIS SUBMITTED BY

Omar ElSeadawy

Under the supervision of
Dr. Nouri Sakr

TO THE

Master of Science in Computer Science

01/31/2023

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Declaration of Authorship

I, Omar ElSeadawy, declare that this thesis titled, "Mixed-Criticality Scheduling using Reinforcement Learning" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Omar Ashraf El-Seadawy

Date:

01/31/2023

Abstract

Mixed-criticality (MC) scheduling is necessary for many safety-critical real-time embedded systems, as a failure of high-criticality jobs could lead to fatal accidents. With the emergence of software technologies in software-defined vehicles in the automotive and avionics industries, studying Mixed-Criticaly (MC) systems is essential to their safety standards, similar to ISO26262. The real-time operation of MC systems makes it an inherently online problem, such that the scheduler is only aware of the jobs that are currently released at any point in time and has no knowledge of future jobs. Due to the overhead cost of preemption, this study focuses on enforcing non-preemption, which makes the problem NP-hard. The literature presents solutions for offline models that allow the scheduler to know about all jobs that are yet to be scheduled from time unit zero and also for systems that allow preemption. Researchers also simplify the modeling of the dynamic elements of the problem, e.g., varying-speed processors, by using simple assumptions that the processor's speed doesn't recover from degradation, which simplifies the problem but is not very realistic. To the extent of our knowledge, we are the first to schedule dual-criticality systems upon non-preemptive, varying-speed processors online. With plenty of researchers approaching the schedulability of such systems with various objectives, our aim in this study is to shed light on the promising nature of emergent machine learning technologies, specifically Reinforcement Learning. We propose a somewhat unconventional approach, where we tackle the modeling complexities using deep reinforcement learning, particularly suitable for problems that generate a sequence of decisions in dynamic environments. Our customized Ape-X model is capable of successfully scheduling sets of jobs of size 50 with an average accuracy of 95% in comparison to other Reinforcement learning algorithms benchmarks conducted, e.g., Augmented Random Search, Proximal Policy Optimization, and Deep Q Networks. Sensitivity analysis shows that training the model with randomized parameters yields a stable performance that is relatively robust to some changes in the generated instances. As part of our future work, we also introduced a simple preemptive version of our system and showed its potential, which reached an average accuracy of 96%. We hope that our study and results motivate the scheduling community to explore the adoption of this effective approach as a promising potential for other dynamic scheduling problems. Thus, we also introduce our recommendations on modeling variants of the problem and discuss possible future extensions.

Acknowledgments

I would like to thank God for reaching this point in my life. I was lucky enough to be chosen by AUC to do my master's degree during the difficult period of COVID. I have received much help from many professors during my journey, and I'm thankful to every one of them. I would like to give special thanks and sincere gratitude to my great therapist and advisor, Dr. Nouri Sakr, for tolerating my falls throughout this journey and for her continuous support and encouragement. It would not have been possible for me to finish my program without her guidance.

I would like to express my deepest thanks to my parents, my lovely fiancee, and my supportive family and friends for their support throughout my degree and for constantly pushing me to finish and do great work. For believing in me and never giving up on me. I can never forget their efforts, and I can never be thankful enough for what they all did for me. I would like to thank all my managers and colleagues who were flexible with me, gave me the time I needed to work on my degree, and for believing in me that I was going to finish this degree as fast as I could.

To my father, my mother, my fiancee, my sister, my family, my friends, my advisor, my colleagues, and everyone who ever helped me throughout my journey. Thank you, I could not have done it without your help and support.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

MC	**Mixed-Criticality**
MCS	**Mixed-Criticality Scheduling**
RL	**Reinforcement Learning**
MDP	**Markov Decision Process**
DRL	**Deep Reinforcement Learning**
DL	**Deep Learning**
ML	**Machine Learning**
LP	**Linear Programming**
CP	**Cyber-Physical**
EDF	**Earliest-Deadline First**
ANN	**Artificial Neural Network**
OCBP	**Own-Criticality Based Priority**
PPO	**Proximal Policy Optimization**
DQN	**Deep Q Networks**
ARS	**Augmented Random Search**

List of Symbols

J	Set of jobs j
j	Single job
r_j	Release time of job j
d_j	Deadline of job j
p_j	Processing time of job j
χ_j	Criticality level of job j
$l^t{}_j$	Laxity of job j at time t
$\rho^t{}_j$	Remaining processing time of job j at time t
t	Time
o_t	Observation at time t
S	State space
A	Action space
B_t	Buffer at time t
v	Processor speed
$v^{min}{}_J$	Degradation bound for instance J
v_t	Speed at time t
a_t	Action at time t
π	Policy of agent
μ	Total load of instance
γ	LO percentage of instance
ω	Degradation Chance
ζ	Job Density of instance
$\lvert B_t \rvert$	Size of buffer B_t
ψ	Laxity Rank
$R^S{}_{j,t}$	Release status of job j at time t
$E^S{}_{j,t}$	Execution status of job j at time t
$S^S{}_{j,t}$	Starvation status of job j at time t

Chapter 1

Introduction

1.1 Overview

In real-time embedded systems, many researchers look into improving the throughput and performance of systems to perform the tasks more efficiently and work is also conducted on guaranteeing the schedulability of the running tasks with different criticalities. The reason is that real-time embedded systems have stringent non-functional costs, weight, and energy requirements. These requirements can be modeled via a defined level of criticality[1] for any of the ***jobs*** (or *tasks*[2]) that need to run in a given system. This model gave rise to the study of ***mixed-criticality*** (MC) systems, where jobs of different criticality levels are consolidated into a shared hardware platform (Barhorst, 2009; Burns& Davis, 2016, Sakr N. et al., 2021). Additionally, these systems may operate on processors that are subject to temporary failure that affects their speed. The development of mixed-criticality (safety-critical) systems is most frequently seen in automotive and avionics, where the system is heavily regulated, especially with the progression of self-driving cars and software-defined vehicles. The design of MC systems is regulated by industrial safety standards that strictly enforce different requirements on each criticality level (Draskovic, Huang, and Thiele, 2016). Different industries have various definitions and standards for criticality levels, but the fundamental concept remains the same. ISO26262 is one example that defines a risk classification system for creating safe software within vehicles. The classification system defines four criticality levels that are labeled according to different factors and consequences, starting from level A (Low-risk tasks such as brake lights malfunctioning) up to level D (High-risk tasks such as failure of electric steering) (Alhaj Ali, 2017). Another example is DO178B, which has five criticality levels with the highest level A is considered catastrophic in the case of failures and the lowest level E is considered harmless with no effect (Johnson, 1998). Authorities have to verify the safety standards of the system and that it abides to relevant rules and safety guarantees. The mixed-criticality system is responsible for handling the execution of jobs of different criticalities while ensuring the system does not pose any risk to humans. All these examples and constraints formulate an interesting dynamic scheduling problem for safety-critical

[1] We distinguish between priority attribution and criticality management: Priority gives an order in which jobs shall run, whereas criticality defines the level of assurance that a job (or system component) would run despite the system's state. For instance, in an attempt to avoid a car crash, camera sensors are allowed to overload the automotive system with caution messages.

[2] We use the terms ***jobs*** and ***tasks*** interchangeably.

real-time embedded systems.

1.2 Background

The literature encapsulates the details of MC systems in an abstract model from a realistic problem, where a job is defined by four parameters: release time, processing time[3], deadline, and a criticality level, which ensures the level of guarantee at which a job would run irrespective of the changing system state. This concept of criticality gives an interesting dimension to MC systems scheduling, or MC scheduling, in short.

MC scheduling is a challenging ***dynamic*** problem, as MC systems usually assume several modes of operation during ***run-time*** (usually as many as criticality levels). For instance, a ***dual criticality*** system operating on a ***varying-speed*** processor can operate in (1) ***normal*** mode or lapse into (2) ***degradation*** mode if the processor is subject to disturbance (e.g., heat, high supply voltage, or disturbed clock frequency) causing its speed to drop, i.e., the processor ***degrades***.

In real-time systems, it cannot be predetermined if, when, or for how long the processor would degrade. In many applications, the consequences of missing a deadline vary in severity from one job to another (Sakr N. et al., 2021). Failing to meet a deadline for a car electronic steering task could be lethal, while failing to meet a deadline for an entertainment system task can be overlooked. Therefore, MC jobs are scheduled so that deadlines are met to the degree that the assumptions of the model are guaranteed to hold (Vestal, 2007). Hence, under speed stochasticity, it is common to guarantee the running of higher criticality jobs by their deadlines, sometimes even at the cost of not running lower criticality jobs. These guarantees make the schedulability of such systems challenging and unconventional.

The literature is rich with works that discuss the schedulability of MC systems under various objectives and conditions (S. Baruah, Easwaran,& Guo, 2016; S. Baruah & Guo, 2013b, 2014; De Niz, Lakshmanan, & Rajkumar, 2009; Gu, Guan, Deng, & Yi, 2013; Guo & Baruah, 2015,2014; Mollison, Erickson, Anderson, Baruah, & Scoredos, 2010). Researchers address simple models of MC schedulability in an offline context, such that the scheduler is aware of all jobs that will be executed during an instance from time unit zero. In particular, Baruah and Guo propose a Linear Programming (LP) to preemptively schedule dual-criticality jobs on a varying-speed processor in an offline context (S. Baruah and Guo, 2013b). Meanwhile, Agarwal and Baruah discuss the intractable online nature of MC scheduling, which is a more complex model of the problem in a way that the scheduler is not aware of all jobs that will be released in the future (Agarwal and Baruah, 2018). We agree with the researchers that the problem is inherently online, where it makes sense not to have prior knowledge of future job arrivals - as is the case with real-time scheduling of embedded systems. We, therefore, focus on an online, dynamic scheduling problem with an imposed non-preemption constraint (S. Baruah and Guo, 2013b; Sakr N. et al., 2021). In fact,

[3] The job processing time models the *worst-case execution time* (WCET) of a task in real-time systems.

scheduling MC systems non-preemptively (even without degradation) is NP-hard and, therefore, challenging (Lenstra, Kan, & Brucker, 1977). However, we believe that real-time embedded systems operate in preferably optimized environments, where it may be impractical to carry preemption overhead costs. For example, systems with mechanical actuators, such as conveyor belts in factories, are very costly to preempt. We, therefore, believe in the value of addressing this complex aspect.

1.3 Problem Statement

Following the framework in (Sakr N. et al., 2021), our problem is defined as follows: With the inherent online nature of real-time mixed-criticality systems, the scheduling of such systems under non-preemptive constraints on a varying-speed uniprocessor poses an NP-hard problem. An unconventional approach is devised using deep reinforcement learning (DRL) to tackle such a problem. In this work, we study online dual criticality scheduling on a varying-speed uniprocessor. While we believe that these characteristics model the nature of real-time embedded systems in a better way, they add a lot of complexity to the scheduling strategy and general intractability of the problem. We, therefore, devise a somewhat unconventional approach, where we address this complexity by devising different ***deep reinforcement learning*** (deep RL or DRL) approaches to tackle our problem.

With the emergence of intelligent methods such as reinforcement learning (RL) and more advanced deep reinforcement learning (DRL), many researchers adopted these methods to discover new perspectives on traditional problems (Aydin & Ö̇ztemel, 2000; Shyalika, Silva, & Karunananda, 2020; Sutton & Barto, 1998; Zhang & Dietterich, 1995). RL models are particularly suitable for problems with a sequence of decisions in dynamic environments. Therefore, we believe that RL and DRL approaches have excellent potential in modeling and approximating the solution to our problem.

1.4 Contribution

Our contribution in this paper can be summarized as follows:

- With the inherent online nature of the problem, we expand upon the work by (Sakr N. et al,. 2021) by focusing on the online version while tackling both non-preemptive and preemptive models for it and doing further sensitivity analysis on these models. Existing works in the literature discuss offline solutions and the intractability of online solutions, but to the best of our knowledge, no one developed an online solution before.
- We take up and focus on the non-preemption challenge as we believe in the practical need to find a scheduling policy that considers such inflexibilities, especially in real-time embedded systems where preemption costs are rather high. We also discuss a preemptive version of our problem as a basis for comparison and future work that can be expanded beyond this paper.

- We model the speed variation in a way closer to what happens in reality. In contrast, previous work simplifies the model to assume that once degradation happens, it remains throughout the rest of the time horizon.
- We propose a rather unconventional approach, as opposed to traditional scheduling strategies, where we recommend leveraging DRL in mixed-criticality dynamic scheduling problems. We design a custom environment to handle the problem characteristics we tackle with a DRL model. We also consider different RL algorithms and study their applicability and proficiency in the dynamic nature of MC scheduling problems compared to our suggested model.

We hope that by showcasing this work and elaborating on our promising results, we provide the research community with a novel, promising direction that may deem useful for modeling and approximating further challenging open problems in dynamic mixed-criticality scheduling.

1.5 Paper outline

This paper is structured as follows: Chapter 2 introduces related work on MC scheduling problems and RL/DRL for scheduling problems. Chapter 3 describes our problem formulation, notation, and RL framework. Chapter 4 presents the model design, and Chapter 5 elaborates on implementation aspects and experimental results. Finally, we conclude in Chapter 6 and discuss further modeling variations and extensions in Chapter 7. This paper is also augmented with a few appendices that outline the details of instance generation, implementation, and algorithms used to run our experiments.

Chapter 2

Literature review

2.1 Mixed-Criticality Scheduling

Barhorst produced a white paper that defines MC systems and explains the rationale behind having an MC architecture: Many embedded systems can be referred to as *cyber-physical* (CP) systems, which bind the physical world with a cyber system that controls it. Some failures in CP systems can threaten human safety and cause fatalities, while other functionalities of such systems might have little to no effect on the physical environment. This explains why many CP systems have notions of different criticalities depending on the executed jobs, which maps to our abstract model of MC systems (Barhorst, 2009). Criticality standards are industry-based, and their definitions and applications vary; however, the core concept of criticalities remains the same, which can be seen in some field standards in ISO26262, IEC 61508, and DO-178B (Johnson, 1998; Alhaj Ali, 2017).

As it became fundamental to balance component separation (for safety) and resource sharing (for efficiency), MC scheduling has gained immense attention in the past fifteen years. Vestal extends the standard fixed priority scheduling to the modified preemptive fixed priority schedulability analysis algorithms and shows that neither rate-monotonic nor deadline-monotonic priority assignment is optimal for scheduling MC systems (Vestal, 2007). A year later, Baruah and Vestal conducted an analysis of the scheduling-theoretic issues in the context of preemptive uniprocessor systems of independent jobs and confirmed that *earliest deadline first* (EDF) scheduling is not optimal and, in some cases, fails to schedule an MC system (S. Baruah & Vestal, 2008).

Further impetus to explore MC systems and solve its schedulability problem results in a series of publications, most of which focus on single processor platforms and independent jobs. Baruah et al. studied the scheduling of MC jobs in real-time systems (Baruah et al., 2012). They present the corresponding model introduced in (S. Baruah, Li, & Stougie, 2010) and (S.K. Baruah, Li, & Stougie, 2010), then prove that deciding the schedulability of a given MC system is NP-hard and analyze the performance of reservation-based and priority-based scheduling. Further variations on work related to uniprocessor scheduling of finite sets of MC jobs include minimizing makespan an (S. Baruah et al., 2016), OCBP-scheduling (Own Criticality Based Priority) (Gu et al., 2013), utilization-based scheduling (Bhuiyan, Sruti, Guo, & Yang, 2019), demand-bound function-based schedulability (Guo et al. 2018), slack scheduling (De Niz et al., 2009), and more. Park and

Kim developed a dynamic scheduling algorithm for certifiable restricted MC systems (Park and Kim, 2011). The algorithm is based on EDF and outperforms OCBP scheduling (Gu et al., 2013). Socci, Poplavko, Bensalem, and Bozga present the time-triggered scheduling paradigm as a promising approach to single and multi-core processors (Socci et al., 2015). Some researchers analyze having a multiple number of criticality levels(Anderson, Baruah, & Brandenburg, 2009; Ekberg & Yi, 2014, 2016; Fleming, 2013; Mollison et al., 2010), while a lot of literature invests in the more restricted *dual-criticality* system, where only two modes are allowed: {HI} or {LO} (Alahmad & Gopalakrishnan, 2018; S.K. Baruah, Burns, & Davis, 2011; Burns & Baruah, 2011; Park & Kim, 2011). Without loss of generality, our paper likewise considers the dual-criticality version.

More relevant to our study is the work by Baruah and Guo, who considers power issues that could lead to a processor having variable speed. As a processor slows down, the processing time of a job increases, which in turn influences scheduling decisions. We study this model and describe it in detail in Chapter 3 (S. Baruah and Guo, 2013). The authors show that EDF does not schedule the respective MC system as desired and extends their work on MC scheduling upon varying-speed processors in (S. Baruah & Guo, 2013a, 2014; Guo & Baruah, 2015, 2014). This speed variation lends the scheduling problem its dynamic nature. To the extent of our knowledge, this dynamic scheduling problem was only discussed in an offline context, although MC scheduling in real-time systems is inherently online e (Agrawal & Baruah, 2018). Therefore, unlike previous work, we focus on analyzing the system under non-preemption conditions and in an online environment.

Scheduling MC systems non-preemptively (even without degradation) is NP-hard (Lenstra et al., 1977). Likewise, many scheduling problems are instances of NP-hard or NP-complete problems. This has encouraged some researchers to merge machine learning (ML) approaches with the solution procedures of selected scheduling problems to provide reasonable heuristics or approximations. The literature is replete with artificial neural network (ANN) models for job shop and flow shop scheduling (Akyol, 2004; Li & Chen, 2009; Philipoom, Rees, & Wiegmann, 1994; Sabuncuoglu & Gurgun, 1996; Zhou, Cherkassky, Baldwin, & Olson, 1991), as well as task scheduling (Agarwal, Jacob, & Pirkul, 2006; Gritzo, 1992). Akyol and Bayhan also provide a review of the ANN approach for solving production scheduling problems (Akyol and Bayhan, 2007). We take a similar, yet more contemporary, direction on the intractable problem at hand as we explore the applicability of reinforcement learning (RL) and Deep RL to approximate it.

2.2 Reinforcement Learning and Deep Reinforcement Learning

Reinforcement learning (RL) is an autonomous self-teaching system that learns through trial and error. RL is a sub-field of Artificial Intelligence that aims to train intelligent ***agents*** to take reward-maximizing ***actions*** in dynamic environments. In particular, the agent interacts with a dynamic environment receiving an ***observation*** of the state of the environment and reacting correspondingly, choosing an ***action*** from a given action space.

After an action is taken, it receives a positive or a negative reward using a ***reward function***. The agent aims to maximize the reward in an iteration of so-called ***episodes***. Each episode is a series of sequential decisions dependent on the problem's nature. The ultimate goal of the agent is to self-learn the optimal policy that maximizes these rewards. Although RL proved successful in many applications, it also showed many limitations in complex problems and lacked scalability (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017). The popular techniques of deep learning (DL), with its wide variety of approaches, including different neural network arrangements, adversarial networks, and many others, proved to be capable of boosting the scalability and functionality of RL to problems that were previously too complex or intractable (Arulkumaran et al., 2017; Shyalika et al., 2020). The new combined field is known as Deep Reinforcement learning (DRL). More details on how we use RL/DRL to model our problem are presented in Sections 3.2 and 4.2.

In principle, dynamic scheduling RL techniques belong to two major categories; Model-Free and Model-based techniques. Model-free techniques can be further divided into on-policy and off-policy and Model-based into Learn Model and Model Given. In the modern day, some techniques combine both model-free, and model-based algorithms, such as Dyna-Q (Shyalika et al., 2020). Model-free techniques rely on trial and error experiences to reach an optimal (max reward) policy. They do not pre-assess the environment's dynamics but rather rely more on acting to learn. Model-free off-policy techniques (Value-Iteration methods) use the Bellman Equation to find the optimal action and policy (Sutton & Barto, 1998). Popular off-policy techniques in dynamic scheduling are Q-learning, Deep Q-Networks, and Monte Carlo Methods. On the other hand, Model-free on-policy techniques (Policy-based Methods) greedily find better policies acquired from improved previous policies, and its most popular technique is PPO (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017). Model-based techniques use transition and reward functions in the environment to estimate the optimal policy (Shyalika et al., 2020; Sutton & Barto, 1998). Other techniques exist that do not fall under a specific paradigm, such as MARWIL and Augmented Random Search (Wang et al., 2018; Mania, Guy, & Recht, 2018).

RL and DRL are suitable for developing dynamic strategies because the agent's goal is to learn a reward-maximizing policy over time while maintaining the state of the environment and coping with uncertainties. Moreover, the agent has no ``right answer'' like in supervised learning but instead learns from the consequences of its actions. Therefore, RL and DRL techniques are conceptually well-suited for dynamic scheduling problems, in general, and for modeling online MC scheduling problems, in particular (Shyalika et al., 2020). It is also worth noting that it is essential to understand the basic parts of the RL learning process, which is mainly defined using the Markov Decision Process (MDP), which is further explained in Section 3.2.

In the scheduling literature, Zhang and Dietterich (1995) apply RL methods to learn domain-specific heuristics for job shop scheduling. Aydin and ̈Oztemel (2000) use RL agents for the dynamic version of the problem. Shyalika conducts a review comparing

different techniques of RL and DRL in the field of dynamic task scheduling. Kong and Wo (2005) proposes a Q-learning approach to tackle dynamic single-machine scheduling problems with random arrivals and processing times. Bouazza, Sallez, and Beldjilali (2017); Shahrabi, Adibi, and Mahootchi (2017) also tackle job-shop scheduling problems with different objectives using Q-learning approaches and policies. Further work related to DRL solutions for scheduling problems includes a combination of graph convolutional networks with actor-critic algorithms for heterogeneous dynamic scheduling (Grinsztajn, Beaumont, Jeannot, & Preux, 2021), which was capable of competing with state-of-the-art offline scheduling algorithms such as HEFT. Luo develops a deep Q-Network (An evolution of traditional RL Q-learning); this DRL technique utilizes neural networks for flexible job shop scheduling, thereby outperforming composite and well-known dispatching rules (Luo, 2020). These examples show that RL techniques have great potential to compete with state-of-the-art offline methods. It also shows that the majority of the RL/DRL literature focuses on dynamic job-shop and task scheduling without much advancement in the MC scheduling field. This is why in this work, we are motivated to explore the benefit of leveraging deep RL to design a dynamic online scheduling policy for dual-criticality systems that operate on a varying-speed processor with non-preemptive constraints. Following the path defined in the most recent work by (Sakr N. et al,. 2021) in the abstract introduces mixed-criticality scheduling under offline and online contexts with preliminary results using deep reinforcement learning, our work shifts towards the online version of the problem. We extend our problem by tackling both non-preemptive and preemptive models of the online problem. In addition, we add different RL benchmark algorithms and introduce a more detailed analysis of the system and performance of the RL agent and its flexibility and robustness.

Chapter 3

Methodology

An MC instance is a set of MC jobs J that are *schedulable* on a varying-speed processor. Following Lemma 1 in (S. Baruah & Guo, 2013b), J is schedulable if the earliest deadline first (EDF) policy schedules all jobs on a speed-1 processor and all HI jobs on a speed-v_{min} processor. That is, there is a degradation bound v_{min}, below which J would no longer be schedulable. Finally, a *correct schedule* runs all jobs by their deadlines as long as v=1 and guarantees all HI jobs meet their deadlines regardless of speed. It is worth emphasizing that, under degradation, no rate-monotonic or deadline-monotonic priority assignment is optimal for scheduling MC systems (Vestal, 2007). In particular, EDF does not work for our problem. Our objective is to study how different DRL algorithms behave when they are trained to schedule MC job instances. Figure 1 summarises the architecture of our system.

3.1 Scheduling Formulation and Notation

The system functionalities or tasks are modeled by a set of n independent jobs J. Without loss of generality, we will consider a dual criticality system, where each job j in J is characterized by its release time r_j, deadline d_j, processing time p_j and criticality level χ_j in {HI,LO}. A speed-v processor runs a job j in $\frac{pj}{v}$ time units. The run-time horizon considered for scheduling all jobs is [minj $\{r_j\}$, max_j $\{d_j\}$].

The dynamic system assumes two modes of operation: *normal* mode (v=1) and *degradation* mode (v<1). We assume a self-monitoring system that immediately knows when degradation occurs during ***runtime.*** The degradation is observed only when it takes place and not predetermined apriori.

Our objective is to leverage DRL techniques to correctly schedule dual-criticality jobs on a non-preemptive, varying-speed processor in an online environment where future job arrivals are not known.

While our research is inspired by the work of (S. Baruah and Guo, 2013b), it is essential to emphasize that our model is fundamentally different as we attempt to model our problem in a way that considers more practical aspects of the problem. We, therefore, cannot adopt the exact system conditions or utilize any previous results as a benchmark. To provide a clear contrast, we summarize the differences between the work of Baruah and Guo as follows:

- We study the online version to capture the inherent nature of this problem, whereas their work is solely offline.
- They allow preemption and discuss that non-preemptive scheduling is NP-hard. We focus on the non-preemptive version of the problem as we believe it avoids impractical preemption costs that their solution can suffer from in real systems. Additionally, we also discuss a preemptive version of the problem to gain insight into how RL performs in both versions.
- They use an LP (with fractional solutions) to find the schedule that tolerates the minimum speed of degradation that can happen. This LP operates offline. We, on the other hand, leverage an RL agent to schedule jobs correctly online using a discretized time horizon.
- They assume that once a processor degrades, it never recovers. In our case, we find this simplification restricting. Therefore, we model degradation more flexibly in a way closer to reality, i.e., processors can reset back to normal modes of operation after periods of degradation.

3.2 Reinforcement Learning Framework

As previously discussed, RL agents can be trained to make sequential decisions that maximize their rewards and are particularly suitable for dynamic problems. In our case, we will attempt to use RL agents to make scheduling decisions that result in correct schedules for a given MC instance J.

We start by modeling this sequential decision-making process as a Markov decision process (MDP), where at each timestep t, an RL agent receives an observation o_t from a state space S. In particular, the observation is a buffer $B_t \subseteq J$ of jobs released for scheduling and the processor speed v_t. Given B_t and v_t, the agent takes a corresponding action a_t from a defined action space A. The action implies the selection of the index of the job to be scheduled at t. This action is tightly connected to a transition probability described by the function $P = p(o_t+1 \mid a_t, o_t)$. After a job is selected, the agent is (later) rewarded or penalized (i.e., negative reward) using a reward function $r(a_t,o_t)$. MDP environments store a value V representing rewards of different states to each other and are calculated based on value function V(s), which is dependent on state s. The value function equation

$V\pi(s) = \sum_{a} \pi(a|o_t) \sum_{(o_{t+1} \in S)} \sum_{r \in R} P(r + \gamma V(o_{t+1}))$ at state s is the average rewards offered by all possible actions at this state s according to policy $\boldsymbol{\pi}$, in which actions give certain probabilities p of transitioning from state s to s' with the future reward of $(r + \gamma V (o_{t+1}))$. The Q-learning approach expands a similar value function into the Q function, which checks the contribution of every action to the optimal policy and finds the best course of action. The Q function can be seen as $V_*(s) = \max_a Q_*(a,s)$, which is applied to all possible actions a at state s to find the highest Q value.

The agent aims to maximize the reward in an ***episode*** by maximizing the findings from the

value function. We consider one episode a series of decisions until all jobs in J either run (and complete by the deadline) or expire (cannot meet the deadline). We discuss the choices of the parameters and functions of this model in Chapter 4. Details of our custom scheduling environment are elaborated in Appendix B.

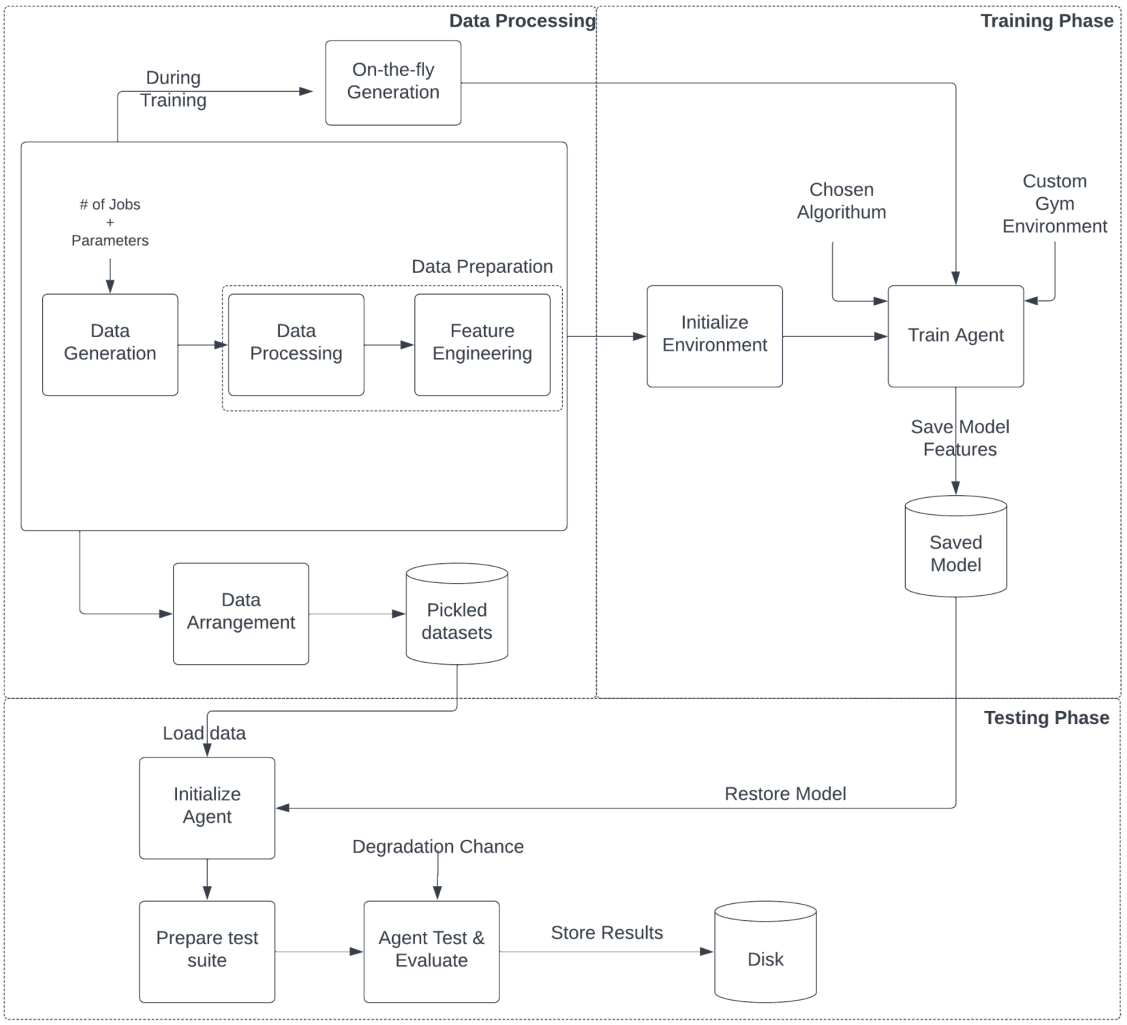

Figure 1: Brief representation of the process starting with the data processing phase, which handles data generation based on the work by Baruah and Guo (S. Baruah and Guo, 2013b) and is followed by the data preparation phase that handles the discretization and filtration of data explained in Chapter 4. Data for training is generated dynamically, while data for testing is arranged into fixed datasets for a fair evaluation. The training phase combines Custom Gym Environment with RLLIB initialization based on the choice of the algorithms, which are listed in section Chapter 5. The agent's model is saved and only restored during the Testing phase, which loads the testing data and evaluates our algorithms. The evaluation framework is explained thoroughly in section 5.3.

Chapter 4

Model Design

We expand upon the foundation by (Sakr N. et al., 2021) to use RL/DRL agents for scheduling MC systems online with the non-preemption constraint. Hence, we have no previous benchmarks for the problem. We modify the workload generation in (S. Baruah and Guo, 2013b) to fit the data generation module in our problem.

4.1 Input Modeling

This section highlights the notation of the parameters needed to understand the rest of the paper. However, the exact details of workload generation, discretization, and filtration are encapsulated in Appendix A so as not to break the flow of the paper. We shall, therefore, only highlight what is needed for the subsequent sections.

- **Job Parameters:** Each generated job j is identified by its tuple r_j, d_j, p_j, and χ_j and monitored using five status parameters computed at every time step t. Two of these parameters are the job's remaining processing time (ρ^t_j) and its laxity (l^t_j), computed as (d_j - ρ^t_j).
- **Workload Generation:** To generate an MC instance of a schedulable job set J, we use some parameters to regulate the workload and its complexity in scheduling. Two of these parameters are highlighted here and used for sensitivity analysis in Chapter 5: The expected percentage of LO jobs in J, denoted by γ, and the total load of J, denoted by μ and computed as the normalized sum of all processing times in J.
- **Environment Parameters:** To ensure the design of a well-controlled environment that properly mimics the behavior of real-time systems, we set some input parameters that specify the degradation chance ω and the minimum allowed degradation speed v_J^{min}, which is unique to every instance J. Both of these parameters are developed in Section 4.3.

Finally, we design a scaling and filtration process to guarantee the integrality and schedulability of any generated workload. See all further details in Appendix A.

4.2 Online Environment Modeling

Online scheduling mimics real-time scheduling, as future job arrivals are generally not known a priori. At any given time t, the agent is able to observe B_t. In particular, it is able to see χ_j, ρ^t_j, and l^t_j for each job in the buffer. The agent can also observe the processor speed v_t, i.e., monitor when a degradation happens and identify the speed. Thus, to define the full state space, we need to model degradation and find a proper representation for the buffer. The details on degradation modeling are encapsulated in Section 4.3. We now describe the buffer modeling.

The RL algorithms we use expect a constant-sized state space. This constraint introduces a modeling challenge because the number of released jobs (that the buffer is expected to hold) may change over time, while the buffer size needs to remain fixed. Our remedy is to let the agent observe a fixed buffer size (independent of the job instance) and do the following modifications: If the number of released jobs so far is greater than $|B_t|$, we only add the jobs with the least laxities[4] to the buffer. Otherwise, we add all jobs plus several dummy jobs[5] that would bring the buffer size to $|B_t|$. To be added to B_t, a job must have been released ($r_j \leq t$) and have enough time to complete ($l^t_j >= t$). As such, all dynamic features of our system are captured. The size of the buffer $|B_t|$ in our system is designed to be 10. The value is based on the assumption that the system would not usually have more than ten jobs overlapping simultaneously, and it would give enough choices to the agent to learn from if less than ten jobs are present. When the agent takes action, it selects a job from the buffer to be scheduled. Hence, the action space is a discrete list of indices corresponding to the jobs in the buffer B_t.

The reward system in the online environment depends on the processor mode (normal or degraded) and the laxity rank ψ of buffer jobs, computed as the order of the job when the buffer is sorted according to the job laxities. For example, in a buffer with five jobs, the job with the highest laxity has a laxity rank of 5, and the job with the lowest laxity would rank 1. Figure 4 illustrates a decision tree to help visualize the reward system along with the resulting reward values based on each action (job selection), and Section 4.4 elaborates further on the tree.

4.3 Degradation Modeling

The LP developed by (S. Baruah and Guo, 2013b) solves for the minimum degradation speed, beyond which the set J would no longer be schedulable. We denote this minimum threshold by v_J^{min} and emphasize that it is unique to every instance J.

Inspired by the LP objective function, we observe that v_J^{min} needs to be an input parameter to

[4] The laxity of a job gives early warnings on expiry. When $l^t_j \leq t$, this means that even if the agent schedules job j at time t, the time would not be sufficient for it to be complete before d_j. Thus, the agent is penalized for choosing such jobs.

[5] To avoid scheduling dummy jobs, we label them as expired and set their criticality to LO.

the system to ensure that we maintain a proper and "fair" training environment for the agent, where J remains schedulable during runtime. Since the LP operates offline under preemption, we cannot use it to define v_J^{min} for our instances. On the one hand, our models are expected to tolerate lower thresholds than those produced by the LP because we do not assume that degradation continues once it takes place. On the other hand, restricting the model to non-preemption may restrict us to higher thresholds compared to the LP solution. Both effects are not guaranteed to produce a consistent final result, i.e., our v_J^{min} may be higher or lower than the LP solution produced by (S. Baruah and Guo, 2013b). Hence, we cannot use the LP to set this parameter. Instead, we approximate v_J^{min} empirically through an iterative process for the training phase described in Appendix A.

As mentioned in Section 4.1, our input parameter ω specifies the degradation chance. That is, if ω = 0.4, then there is a 40% chance for the processor to degrade at each time step. The degradation in our environment is modeled following a two-stage process at every time step t:

1. Draw a uniform random number p from U (0, 1). If p < ω, the processor degrades. Otherwise, it doesn't
2. If the processor degrades, v_t is drawn from U (v_J^{min} , 1). Otherwise, v_t is 1.

Figures 2 and 3 demonstrate the effect of ω on speed behavior over a series of 10,000 experiments: We can identify patterns of more frequent spikes as ω increases. We can also see that this modeling approach allowed us to resemble real-life processor behavior where degradation may happen randomly, at any point in time, and in sudden spikes, where the speed drops temporarily and then goes back to 1.

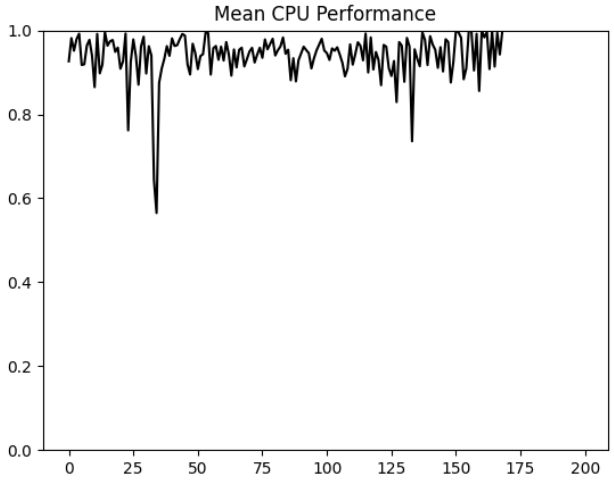

Figure 2: Speed behavior vs time when ω = 0.15 over a series of 10,000 experiments for a job instance with an average v_J^{min} of 0.5.

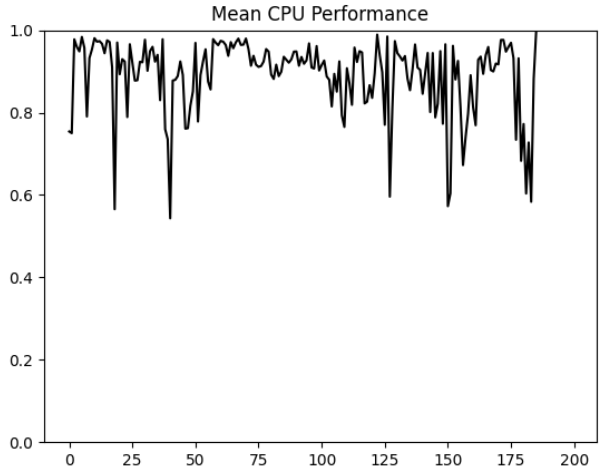

Figure 3: Speed behavior vs time when ω = 0.8 over a series of 10,000 experiments for a job instance with an average v_J^{min} of 0.5.

4.4 Reward Modeling

In reinforcement learning, the reward function basically defines the objective of the agent. It draws a map for each taken action and state in the environment and links it to a numerical value, a reward (or punishment for a negative value). The goal of the agent is to maximize the total reward during the entire runtime of each independent iteration. The reward function is defined based on the context of the problem to highlight the positive or negative events in which the agent is judged upon (Yunior F. et al., 2015). The numerical values of reward functions are defined based on the context of the problem. Different researchers approach the design of the reward function relative to the nature of their problem aiming to reach the objective or goal of their problem. In our case, we aim to maximize the agent's reward based on the priority of the job it schedules. The reward function in our model is designed to help the agent train to schedule jobs correctly. The reward function prioritizes high-criticality jobs slightly over low-criticality jobs at normal speed by rewarding the agent for scheduling high jobs, with no effect on scheduling low-criticality jobs.

We use a numerical value C to test the sensitivity of the biases towards high criticality jobs. The value C is tested for values [1,5,10,30] and the results in Section 5.4 show that the reward function is consistent with the objective of our agent maintaining an average above 95% of completion. On the other hand, during degradation, the agent is heavily punished for choosing low-criticality jobs over high-criticality jobs (if they exist in the buffer) and better rewarded for choosing high-criticality jobs. All experiments in Chapter 5 are done with constant value C=5.

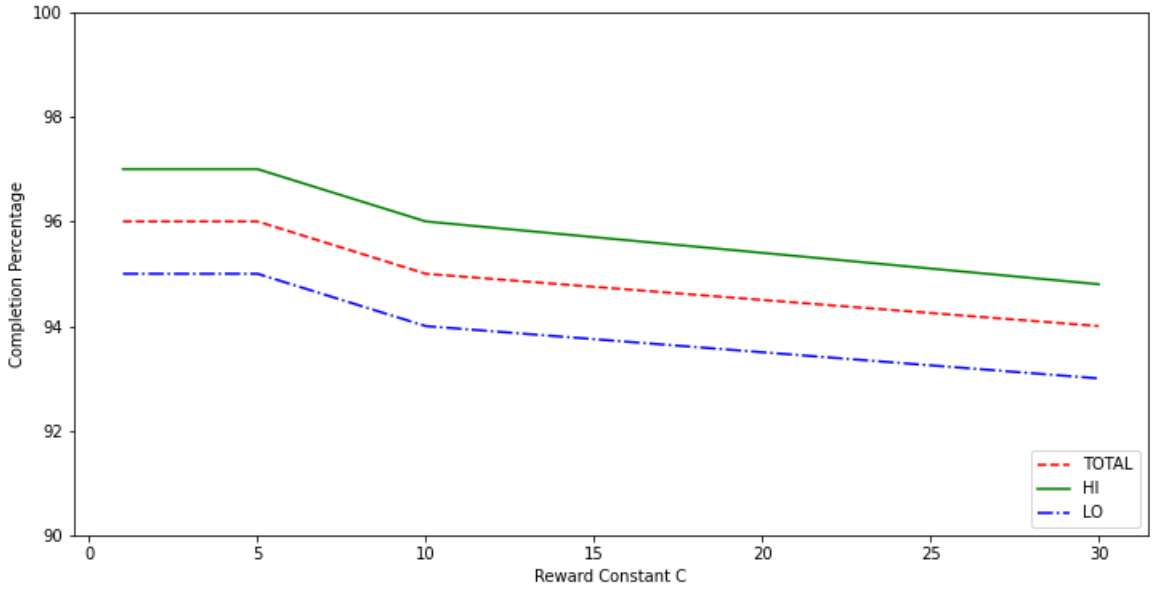

Figure 4: Sensitivity of Agent vs Reward constant C in the reward function.

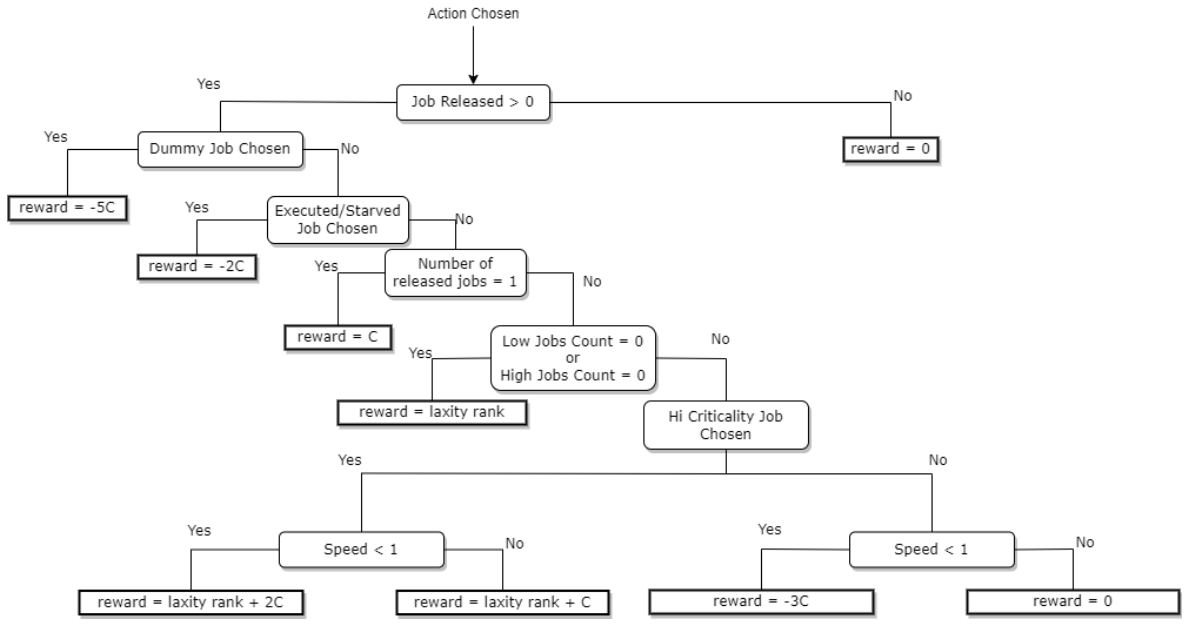

Figure 5: Reward process decision tree visualizing the agent's reward function at a given time t.

Figure 4 shows our reward function in a decision tree shape to help visualize the process and how it affects the agent. The agent's action is to select a job to be scheduled, and the reward is applied based on the choice. Initially, if no jobs are available in the buffer and it is filled with dummy jobs, the reward is zero, and the agent moves on in time. If available jobs exist, choosing a dummy job is penalized with -25 to disallow the agent from picking dummy jobs over released jobs. Choosing a previously executed or starved job is also penalized with -10

to disallow the agent from picking them. If the agent picks a correct job, but it is the only available job in the buffer, the agent is rewarded with a small value of 5. If there is more than one job in the buffer and all jobs in the buffer are either of high criticality only or vice versa, the agent is rewarded with the laxity rank of the job in the buffer. Suppose there are both high and low-criticality jobs in the buffer. In that case, the reward depends on the processor mode, the number of available high and low jobs, and the criticality of the chosen job. Suppose the processor is operating in normal mode. In that case, the agent gets a positive reward based on the laxity rank of the job with an extra constant for picking a high-criticality job and a zero reward for picking a low-criticality job. Suppose the processor operates in a degraded mode. In that case, the agent is rewarded with a positive value of laxity rank plus a larger constant for picking a high-criticality job and is penalized for picking a low-criticality job. Now our whole environment is modeled according to our problem characteristics; the next sections focus on the experiments done on this problem model and explain our implementation choices and analytical results.

Chapter 5

Experimental Results

5.1 Algorithm Implementation

There exist many different RL and DRL algorithms in the field, as mentioned in the literature review in Chapter 2. For this study, we decided to test different algorithms and how they behave in trying to solve a dynamic MC scheduling problem. To avoid overlapping algorithms that use the same core concept, we decided to test four different algorithms. The first three algorithms were used with default configurations from RLLIB as a benchmark for our custom gym environment. The Ape-X DQN was built as a custom model with action masking inherited from the RLLIB TFModelV2 model with the provided Ape-X Trainer (Liang et al., 2017). The list of our choice of algorithms is as follows:

- Augmented Random Search (ARS), a random search method capable of training policies for continuous control problems, is known for being fast and efficient compared to model-free and model-based algorithms (Mania et al., 2018). This algorithm was chosen as a benchmark for our model, considering its nature of random search.
- Proximal Policy Optimization (PPO), a model-free on-policy technique that uses multiple stochastic gradient descent passes over the same experiences and is suitable for scalability (Schulman et al., 2017). According to Shyalika et al. (2020), PPO showed great potential in dynamic scheduling problems; however, it might suffer from high variance and slow convergence based on the context of the problem.
- Deep Q Networks (DQN), a model-free off-policy combination of RL Q-learning variant with a convolutional neural network (Mnih et al., 2013). DQN showed great success in large-scale and high-dimensional MDP problems and is capable of achieving fast, and stable training in dynamic sequential decision-making processes (Shyalika et al., 2020; Sun and Tan, 2019).
- Ape-X DQN (APX), a distributed architecture that uses Deep Q Networks with a combination of neural networks, that relies on a distributed architecture that separates acting from learning asynchronously (Horgan et al., 2018). It follows the same concept as DQN; however, Ape-X introduces more efficient training and experience replay with its distributed nature and combination with neural networks.

DQNs effectively solve MDP models resembling dynamic decision-making problems in dynamic, unpredictable environments and continuous observation spaces. This inspired us to develop Ape-X DQN as our custom model and compare it to different algorithms. The

details of the implementation[6] of our algorithm and RL environments are mentioned in Appendix B.

5.2 Training Stage

In the workload generation and filtration stages, we guarantee that all generated instances of J are schedulable using non-preemptive EDF under normal speed v=1 (See Appendix A for details). Our agent is initially trained with on-the-fly generated instances of J and randomized input parameters. When the training process saturates, the agent becomes ready for evaluation.

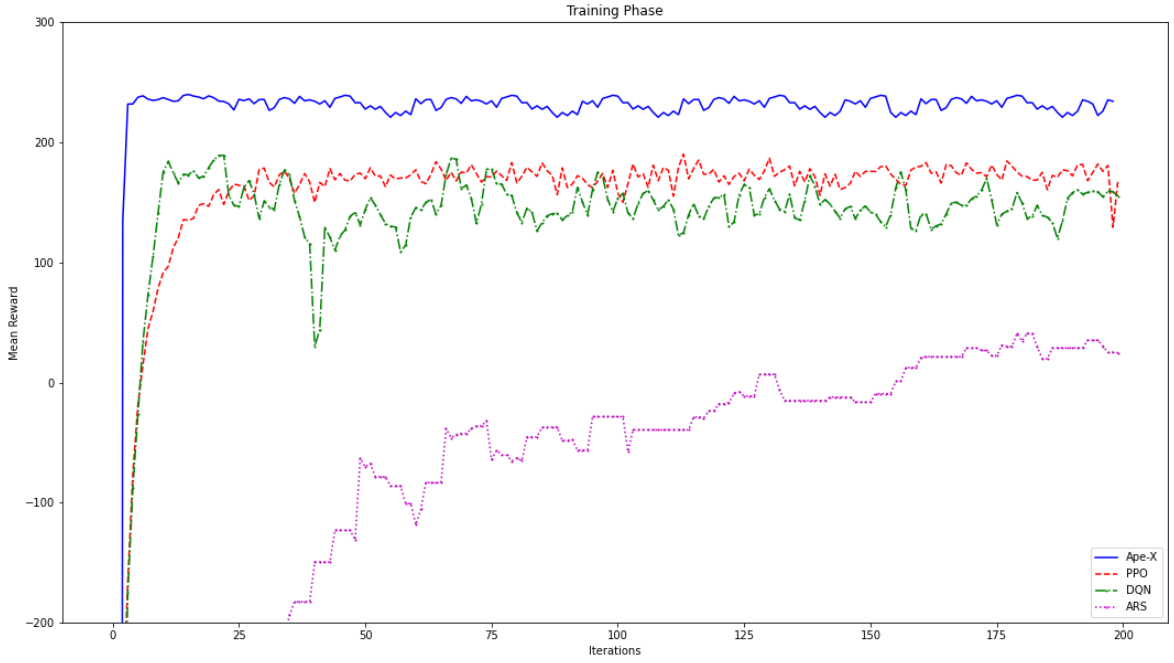

Figure 6: Reward mean for all algorithms during Training Phase

In particular, we generate MC instances, i.e., sets of jobs, each of size 50, using randomized parameters, as detailed in Appendix A. The agent learns through a series of iterations, in which each iteration trains on average 1000-2000 episodes. The mean reward of each iteration is monitored to analyze if the agent is improving its performance or not. The mean reward of iterations usually starts with a negative value since the agent begins with random choices and starts to converge after a couple of hundred of episodes of learning. Once training is finished, the model is loaded and used for testing. Figure 5 shows the performance of each of our algorithms during the training phase, how they start and how fast they learn over time. The figure shows that our custom Ape-X model is the fastest

[6] Implementation was done on the following specifications: Processor Intel(R) Core i7-8750H @ 2.20GHz with 6 cores and 12 threads, 16 GB of RAM, and an Nvidia GeForce RTX 1060 GPU. The operating system used was Ubuntu 20.04.3 LTS and Python Version 3.8.

learner, with DQN and PPO being a close second. The random search ARS shows that it is a very slow learner and stabilizes at a very low reward.

We also test the speed of the algorithms during the evaluation because the execution speed of making decisions can be critical to systems with safety requirements. Each algorithm was tested on 50 job instances from 9 different datasets, and an average was taken. Figure 6 and Table 1 show that ARS is indeed the fastest algorithm for evaluation, even though it is not the best in terms of accuracy.

Table 1: Average Execution time per second for 100 runs and per single run. Calculated as an average of 9 different sample datasets with 50 job instances each.

Algorithm	Avg Total/s	Avg per run/s
Ape-X	7.91	0.08
PPO	7.34	0.07
DQN	7.79	0.08
ARS	4.92	0.05

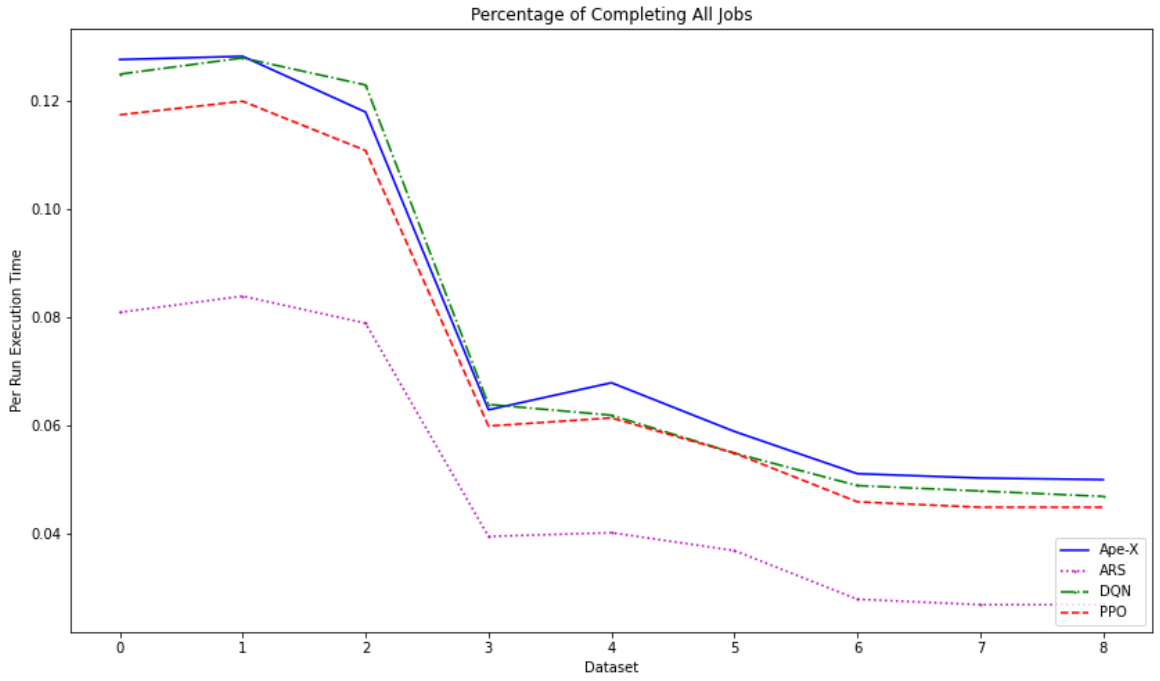

Figure 7: Average Execution time per dataset for all algorithms.

DQN, Ape-X, and PPO all have almost similar execution times, and they are only slightly slower than ARS. Figure 6 also shows that execution time decreases in the datasets with the highest total load. We can deduce that since datasets with higher loads are usually busy with

long jobs, the algorithm takes less time to make decisions and generally waits for jobs to finish. To assess the performance of our RL agent and, thus, the viability of our proposed model, we build the evaluation framework described in the next section.

5.3 Evaluation Framework

To contrast the behavior of the agent under a variety of scenarios, we choose to build the cross-combinations of the following values for our main input parameters: Total load $\mu \in$ (0.3, 0.6, and 0.9) to cover different scenarios of workloads with series of jobs with low overall processing time for low μ and series of jobs with high overall processing time for high μ. The parameter expected LO percentage $\gamma \in$ (0.3, 0.5, and 0.8), which tries to cover instances with the majority of jobs being low criticality, high criticality, or a balanced combination of both. Finally, degradation chance $\omega \in$ (0.2, 0.4, 0.6, 0.8, 1) to assess the agent under various degradation chances (assuming an environment could be chaotic and forces lots of degradations while other environments could rarely expect degradations). Each parameter combination builds a set of 2000 test instances. The variety of these combinations allows us to evaluate the agent's performance under different algorithms while observing their sensitivity to different input parameters. Appendix A outlines the impact of each parameter variation on the instance characteristics. However, it is worth emphasizing that μ and γ mainly affect the instance generation while ω only affects the run time environment and is generated dynamically with the instances. In other words, the first parameter combination would be when $\mu = 0.3$ and $\gamma = 0.2$. These parameters generate a set of 2000 test instances with 50 jobs. Afterward, this set is tested five times, once for every value of ω. Finally, the results are averaged across the different iterations first, then across different values of ω for each algorithm. The rest of this section reports significant evaluation and sensitivity analysis results.

For each test instance J, the agent schedules it according to its learned policy. The evaluation is based on the percentage of total correctly scheduled jobs. Table 2 shows the percentage of correctly scheduled jobs for every algorithm across all datasets with different parameters. The agent is evaluated on a combination of values for $\mu \in$ (0.3, 0.6, and 0.9) and $\gamma \in$ (0.3, 0.5, and 0.8). The best agent performance was seen in the Ape-X DQN model with an overall average completion of 95.1%, followed by the default DQN model with an overall average completion of 93%. The job completion percentage drastically falls with the increase in μ, which is the total load of the dataset affecting the processing times in the job instance J. The agent with the lowest performance was surprisingly the PPO agent with an overall average completion of 79%.

Our custom Ape-X model proved to be the most accurate compared to the rest of the benchmarks and is considered the second fastest algorithm in evaluation speed, which is why it will be studied for further analysis of its performance.

Table 2: Job Completion Percentage for each algorithm for every dataset. Average of 18000 experiments across varying ω per dataset with 50 jobs.

#	μ	γ	**Ape-X**	DQN	ARS	PPO
1	0.3	0.2	98.1	97.8	96.1	93.6
2	0.3	0.5	97.9	97.8	95.1	93.4
3	0.3	0.8	97.2	97.1	94.4	92.4
4	0.6	0.2	95.7	94	81.1	77.2
5	0.6	0.5	95.4	93.6	80.4	76.9
6	0.6	0.8	94	92.2	78.4	75.3
7	0.9	0.2	93.9	89.6	74.6	69.2
8	0.9	0.5	93.2	88.7	73.4	68.9
9	0.9	0.8	90.9	86.5	69.9	66.9

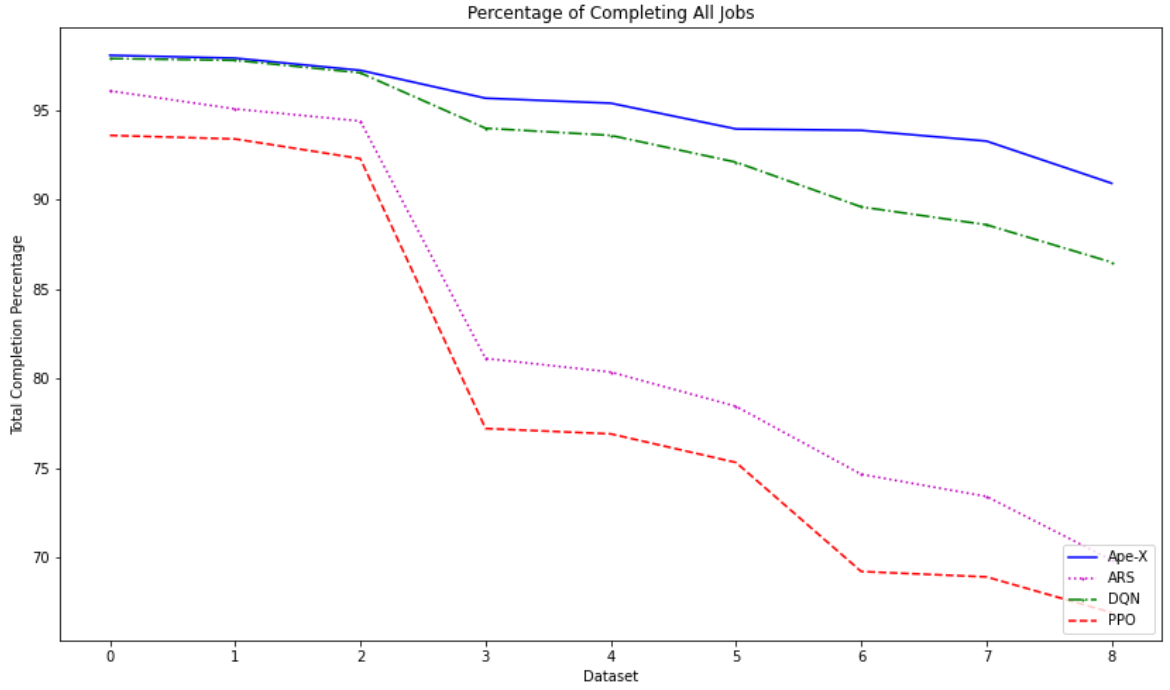

Figure 8: Performance of different agents in terms of total completion percentages for each dataset

5.4 Sensitivity analysis

Further analysis of Ape-X Model can be seen in Figures 8,9,10 and 11, which shows the graphical results of HI and LO completion percentages against multiple different parameters ω, μ, γ, and v_{min}. For the increasing degradation chance ω, the graph shows a slight drop in performance as the completion percentage drops from an average of 96% to an average of 94%. This drop is logical because systems with a 100% degradation chance are more likely to have scheduling failures. When it comes to the total load parameter μ, it shows the biggest fall in performance with its increase. A low μ value of 0.3 had an average completion percentage of around 98%, while a high μ value of 0.9 had significantly dropped to below 93%. This figure shows that in most instances, J with a high sum of processing times degrades the agent's performance significantly, mostly because long processing times cause jobs to overshadow others and thus starve many jobs in the instance. The parameter (γ) LO percentage almost has a negligible effect on the completion percentage of the HI criticality jobs and faces a very slight drop in LO criticality job completion. Finally, Figure 11 shows the HI and LO completion percentages for each dataset, which confirms the findings.

Our sensitivity analysis shows that our model is susceptible to some parameter variations, such as μ, while being unaffected by other parameters, such γ. The output shows that the model was mildly sensitive to the changes in γ with a very slight drop in performance, while μ and ω significantly impacted the agent's performance. In the following subsection, we briefly introduce a preemptive version of our model using our best-performing algorithm Ape-X. It was designed to test the capabilities of the RL agent in the preemptive model vs. our focused non-preemptive model.

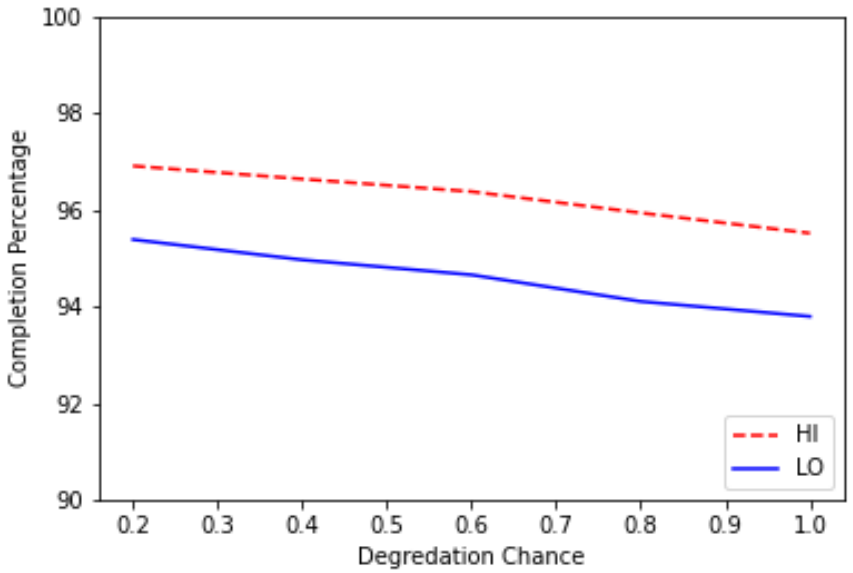

Figure 9: The performance of the agent in terms of HI and LO completion percentages vs. Degradation Chance ω

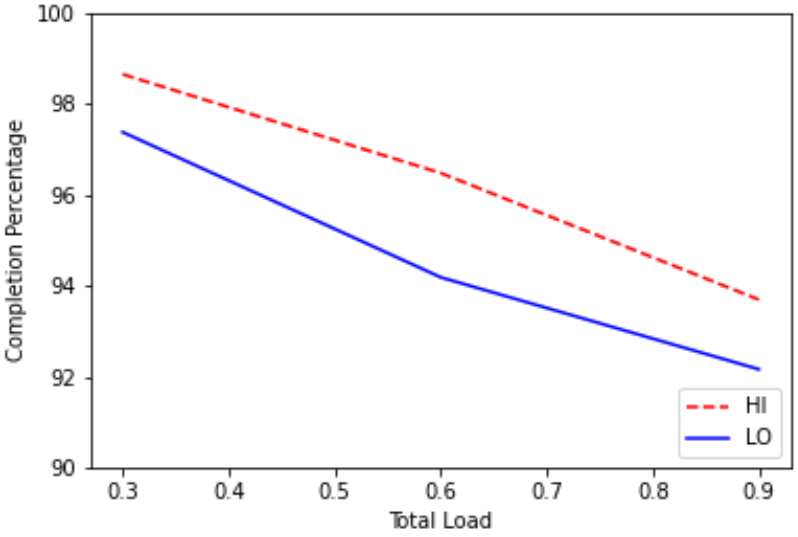

Figure 10: The performance of the agent in terms of HI and LO completion percentages vs. Total Load μ.

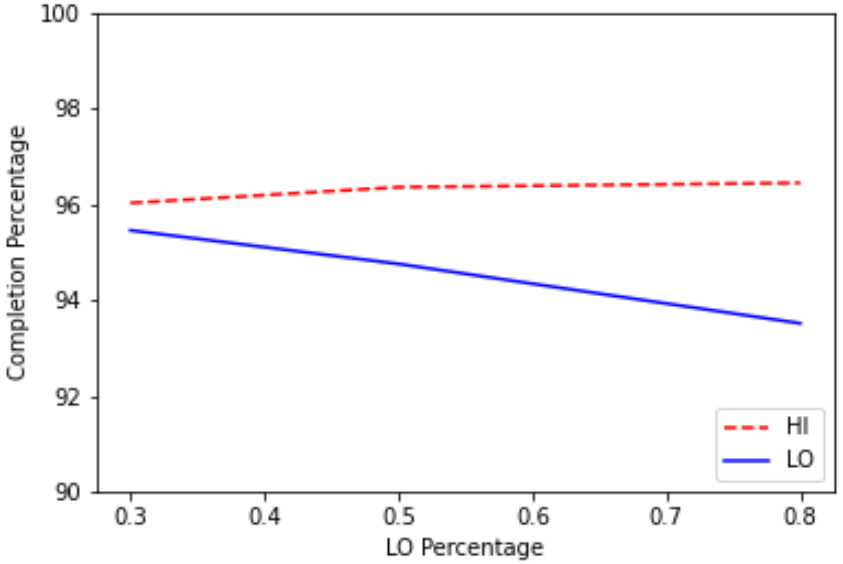

Figure 11: The performance of the agent in terms of HI and LO completion percentages vs. LO Percentage γ

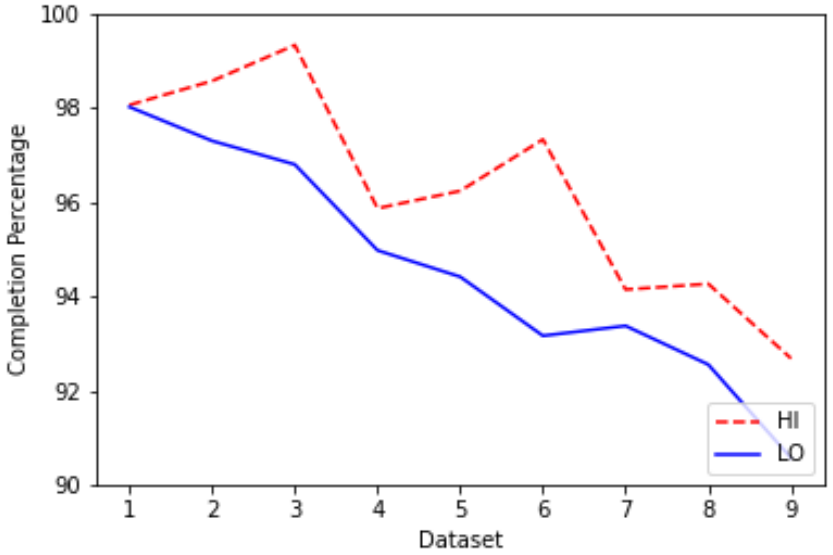

Figure 12: The performance of the agent in terms of HI and LO completion percentages for each Dataset

5.5 Preemptive Model

In real-time embedded systems, some systems require preemption to exist as a way of completely stopping any low-criticality jobs in favor of completing high-criticality jobs. This is why we decided to study the performance of our algorithm on a preemptive version of the problem. The custom environment of our RL agent is adjusted to handle preemption following the simple assumption: Whenever the system is executing a low-criticality job, and the processor falls into degradation mode, the agent preempts the execution and chooses a new job. The choice of the new job is based on the reward function and is favored toward high-criticality jobs. In contrast, the low-criticality job that was stopped is updated based on its remaining time and added back to the buffer until it gets resumed or starved, depending on the duration of high-criticality jobs taking its place. The purpose of introducing a preemptive model is to encourage future work in this area and also to compare the performance of a preemptive vs. non-preemptive model of our custom environment and RL algorithm. The preemption model was only tested for the Ape-X algorithm as it showed its performance and potential.

Figures 12, 13, 14, and 15 show the performance of our agent in terms of HI and LO completion percentages for the preemptive model of the problem. Overall, the graphs show a slightly higher average completion percentage than the non-preemptive model, increasing from around 95% to 96%. The sensitivity analysis of the model is almost similar to the performance of our agent in the non-preemptive version of the problem. Parameter variation in μ, ω causes the biggest drop in performance, while variation in γ can be negligible for high-criticality jobs with a faint drop for low-criticality jobs.

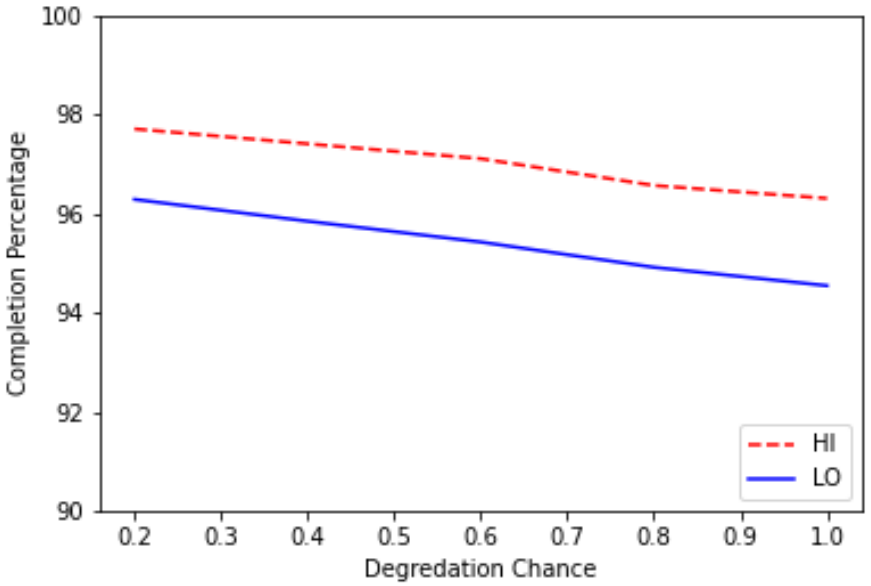

Figure 13: The performance of the agent in terms of HI and LO completion percentages vs. Degradation Chance ω

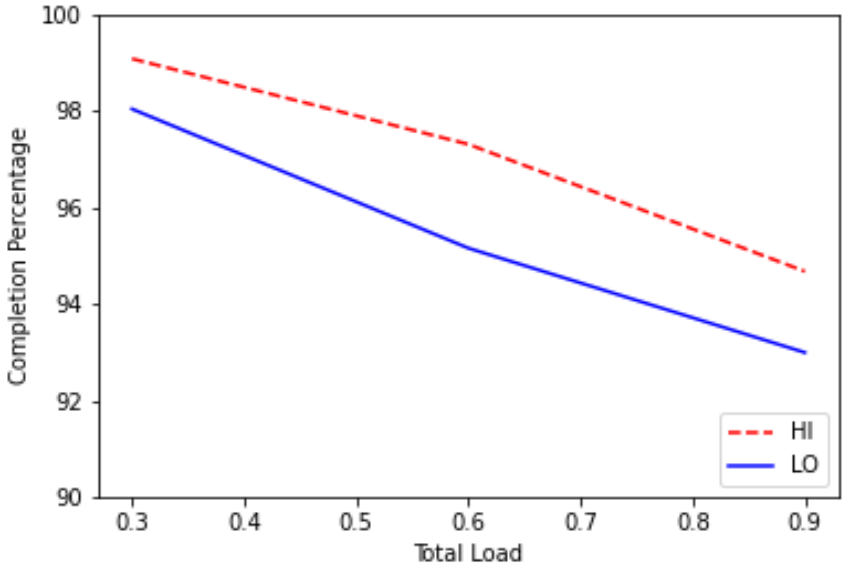

Figure 14: The performance of the agent in terms of HI and LO completion percentages vs. Total Load μ.

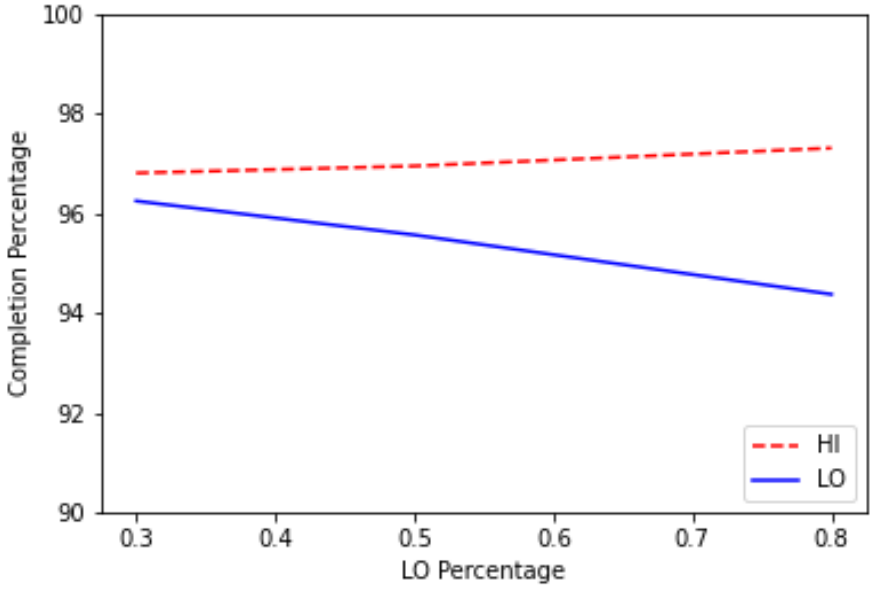

Figure 15: The performance of the agent in terms of HI and LO completion percentages vs. LO Percentage γ

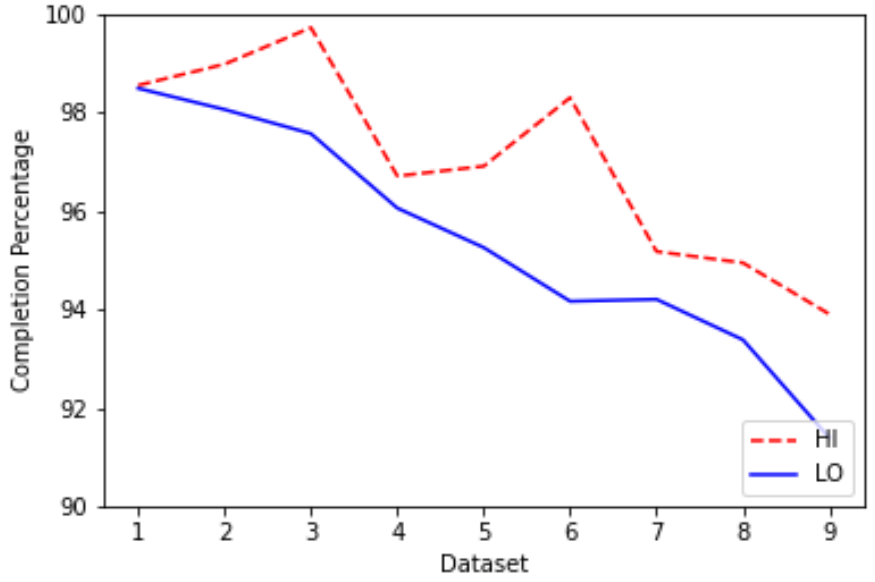

Figure 16: The performance of the agent in terms of HI and LO completion percentages for each Dataset

5.6 Comparison

To our knowledge, we are the first to use RL agents for scheduling MC systems non-preemptively and online. Hence, no previous RL-based benchmarks or test datasets were published. Moreover, the literature already confirms that standard scheduling policies, such as EDF, do not apply to this problem and do not provide us with known approximations or heuristics that can be used for the online problem. On the other hand, many of the existing solutions for the offline versions of this problem are not concerned with dynamic scheduling as they do not consider varying-speed processors. But even the offline solutions for varying-speed systems, such as in the seminal work of (S. Baruah and Guo, 2013b) are not comparable to ours due to the multitude of differences in our model assumptions, which we present in Section 3.1. We cannot guarantee how these differences will impact the quality of any comparative study based on their work. It is also worth emphasizing that our modeling approach also enabled us to model complex aspects, i.e., degradation, in a way that made our model converge closer to real-time systems behavior when compared to the models in the existing literature. Last but not least, our objective function is fundamentally different from those considered in the literature.

These reasons justify why it would be infeasible to compare the performance of our agent to the performance of any previously published work without making considerable changes to the model design and assumptions.

Chapter 6

Conclusion

With the rapid development of technologies in unmanned vehicles and drones in automotive and avionics industries with the enforcement of safety standards, we believe the unconventional combination of reinforcement learning with mixed criticality scheduling is prominent and shows great potential in the field. We believe in the inherent online nature of real-time embedded systems even though it adds complexity to the problem model and we also believe that some cases require the removal of preemption cost and context switching overheads so our focus shifted towards the non-preemptive version of the problem. Our work leverages different RL and DRL agents to correctly schedule dual-criticality systems on a non-preemptive, varying-speed processor in an online environment. We additionally adopt a more realistic model for the varying-speed processor behavior and the nature of degradation in our problem.

To the best of our knowledge, we are the first to address the problem under non-preemption and in an online context. Our results show the performance and speed of different RL/DRL algorithms in tackling this complex problem, concluding that the Ape-X DQN algorithm was the most suitable in comparison to the other benchmarks PPO, DQN and ARS. Non-preemptive Ape-X agent was capable of correctly scheduling 95% of jobs per instance on average and showed its flexibility and robustness. Our results also show that this rather unconventional approach to a very critical dynamic scheduling problem is promising and has potential for further research as explained in our future work, which suggested different variants and extensions on the problem. Experimenting with more complex preemption models, formulating a scalable mathematical model for reward function, adding multi-processor models and finding other AI approaches are all different perspectives that could give more insight to this approach. Moreover, according to the sensitivity analysis results, the agent was able to endure high values for degradation chance to an extent but is very sensitive to a total load of its instance, which means that the higher the normalized sum of jobs per instance the more difficult for that agent to schedule everything perfectly.

Additionally, we discuss a preemption model of our problem using Ape-X DQN, obtaining an overall average completion percentage of 96% with robustness and flexible performance against the environment parameters. The results show that the performance of the agent shows great potential in tackling other preemption approaches as well. We also discuss moving to an offline context and testing out RL techniques in that perspective.

Chapter 7

Variations and Future Extensions

The results of the previous section confirm that our RL approach has great potential to serve one of the most critical dynamic scheduling problems in real-time embedded systems. This outcome motivates us to explore more aspects and extensions in future work. In this section, we discuss how our work may be modified to suit other variations of the problem that are out of the scope of our work, namely preemption and offline systems. We also discuss limitations and future considerations of the work.

In order to modify our model to incorporate preemption, we showed an approach in Section 5.5 that preempts the agent whenever it goes into degradation mode while executing a low-criticality job. Other approaches for preemption can be considered as follows: The agent can decide, at every time step, whether it first wants to preempt a currently running job or not, then which job will be scheduled if the system was preempted or just completed a job. We can add a hyperparameter similar to ω, representing the preemption chance, i.e., the probability by which the agent might preempt a running job at any given time. The implementation can then be modified to include this hyperparameter. After that, if we want to revert back to non-preemption, we would just set the preemption chance to zero.

Scheduling MC systems offline using RL agents can be tricky because the agent starts with information about all (future)job arrivals and produces the schedule in **one go**. To model the offline environment, we recommend the following changes to the model and evaluation framework:

- Set B_t = J at all times because the agent now has visibility of the agent's overall (future) arrivals.
- No a priori information exists about the degradation, so the RL agent will assume that v_t=1 is fixed at all times.
- The action space would be a list of all indices of released, non-expired, and non-scheduled jobs. This ensures that the agent is not overwhelmed by the amount of information in the buffer and is only using this information to make better decisions, not to make more mistakes, such as scheduling jobs that have not been released yet.
- One training episode for the offline model would let the agent schedule the entire list of jobs at t=0 without considering any degradation or getting any feedback from the reward system.
- The entire schedule is then played against a processor prone to degradation and is

evaluated accordingly to specify the reward. The agent may then be allowed to reschedule the same jobs, correct some decisions, produce a full schedule, and get the next reward. This process may be repeated multiple times but is threatened by overfitting, especially if the instance is not large enough.

With respect to the limitations of our problem, we next discuss the size of the data. While scalability is very important in machine learning techniques, our model is currently limited by various workload generation and filtration checks that limit the number of jobs that can be generated and verified per instance. This makes sense because deciding the schedulability of non-preemptive systems is NP-hard (S. Baruah et al., 2010), thereby making our current filtration process very time-consuming. In future work, it is worth investigating a remedy for this bottleneck and finding speed-up tricks that optimize both the data generation process, as well as the efficiency of the training process. Increasing instance sizes may unmask further modeling issues about our approach that may need to be addressed when handling a large number of jobs. It also opens the door for more elaborate discussions on scalability issues, which are essential in real-time systems.

Modeling this problem with multiple criticality levels similar to ISO26262 and DO178B could also give more insight on the performance of RL on such systems instead of only addressing a dual-criticality system with only two levels of criticalities. The multitude of criticalities might perform better with real data instead of simulated data or with a more complex data generation model.

Other future work may invest efforts in finding other benchmarks for this version of the problem and good datasets to be used for testing purposes. Additionally, it is worth investigating other objective functions and conducting a sensitivity analysis with respect to different formulations of reward functions. Building a more concrete mathematical model for the reward function that is more scalable and extensible is also a very good concept to tackle. It would also be interesting to test and extend this approach to multi-processor systems that allow resource sharing instead of relying on a single processor.

Last but not least, we believe it would be helpful to run an error analysis on unsuccessful schedules and use recurrent neural networks for partial observability. Using different artificial intelligence approaches such as deep neural networks for scheduling or combining RL with other classification techniques to improve the agent's performance.

References

1. Agarwal, A., Jacob, V., Pirkul, H. (2006, 02). An improved augmented neural-network approach for scheduling problems. INFORMS Journal on Computing, 18 , 119- 128.

2. Agrawal, K., & Baruah, S. (2018). Intractability issues in mixed-criticality scheduling. Proceedings of the 30th euromicro conference on real-time systems, (ECRTS 2018). Barcelona, Spain: IEEE.

3. Akyol, D.E. (2004). Application of neural networks to heuristic scheduling algorithms. Computers & Industrial Engineering, 46 (4), 679–696.

4. Akyol, D.E., & Bayhan, G.M. (2007). A review on evolution of production scheduling with neural networks. Computers & Industrial Engineering, 53 (1), 95–122.

5. Alahmad, B.N., & Gopalakrishnan, S. (2018). Risk-aware scheduling of dual criticality job systems using demand distributions. Leibniz Transactions on Embedded Systems., 5 , 01:1–01:30.

6. Alhaj Ali, A. (2017). ISO26262 functional safety standard and the impact in software lifecycle. 10.13140/RG.2.2.12486.16963

7. Anderson, J.H., Baruah, S., Brandenburg, B.B. (2009). Multicore operating-system support for mixed criticality. Proceedings of the workshop on mixed criticality: Roadmap to evolving uav certification.

8. Arulkumaran, K., Deisenroth, M., Brundage, M., Bharath, A. (2017). A brief survey of deep reinforcement learning. IEEE Signal Processing Magazine, 34 .

9. Aydin, M., & Oztemel, E. (2000). Dynamic job shop scheduling using reinforcement learn- ing agents. Robotics and Autonomous Systems, 33 (2), 169-178.

10. Barhorst (2009). White paper: A research agenda for mixed-criticality systems. Retrieved from http://www.cse.wustl.edu/cdgill

11. Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N., Stougie, L. (2012). Scheduling real-time mixed-criticality jobs. IEEE Transactions on Computers, 61 (8), 1140–1152.

12. Baruah, S., Easwaran, A., Guo, Z. (2016). Mixed- criticality scheduling to minimize makespan. Lipics-Leibniz international proceedings in informatics (Vol. 65).

13. Baruah, S., & Guo, Z. (2013a). Mixed-criticality scheduling upon unreliable processors.

14. Baruah, S., & Guo, Z. (2013b). Mixed-criticality scheduling upon varying-speed processors. Real-time systems symposium (RTSS), 2013 IEEE 34th (pp. 68–77).

15. Baruah, S., & Guo, Z. (2014). Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor. Real-time systems symposium (rtss), 2014 IEEE (pp. 31–40).

16. Baruah, S., Li, H., Stougie, L. (2010). Towards the design of certifiable mixed-criticality systems. Real-time and embedded technology and applications symposium (RTAS), 2010 16th IEEE (pp. 13–22).

17. Baruah, S., & Vestal, S. (2008). Schedulability analysis of sporadic tasks with multiple criticality specifications. Real-time systems, 2008. ECRTS'08. Euromicro conference on (pp. 147–155).

18. Baruah, S.K., Burns, A., Davis, R.I. (2011). Response-time analysis for mixed criticality systems. Real-time systems symposium 23 (RTSS), 2011 IEEE 32nd (pp. 34–43).

19. Baruah, S.K., Li, H., Stougie, L. (2010). Mixed- criticality scheduling: Improved resource-augmentation results. Cata (pp. 217–223).

20. Bhuiyan, A., Sruti, S., Guo, Z., Yang, K. (2019). Precise scheduling of mixed-criticality tasks by varying processor speed. Proceedings of the 27th international conference on real-time networks and systems (p. 123–132). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3356401.3356410 10.1145/3356401.3356410

21. Bouazza, W., Sallez, Y., Beldjilali, B. (2017). A distributed approach solving partially flex- ible job-shop scheduling problem with a Q-learning effect. IFAC-PapersOnLine, 50 (1), 15890-15895. (20th IFAC World Congress)

22. Burns, A., & Baruah, S. (2011). Timing faults and mixed criticality systems. In C.B. Jones & J.L. Lloyd (Eds.), Dependable and historic computing: Essays dedicated to brian randell on the occasion of his 75th birthday (pp. 147–166). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-24541-11210.1007/978-3-642-24541-112

23. Burns, A., & Davis, R. (2016). Mixed criticality systems - a review. Department of Computer Science, University of York, Tech. Rep.

24. De Niz, D., Lakshmanan, K., Rajkumar, R. (2009). On the scheduling of mixed-criticality real-time task sets. Real-time systems symposium, 2009, RTSS 2009. 30th IEEE (pp. 291–300).

25. Draskovic, S., Huang, P., Thiele, L. (2016, Novem- ber). On the Safety of Mixed-Criticality Scheduling. WMC 2016. Porto, Portugal. Retrieved from https://hal.archives- ouvertes.fr/hal-01438846

26. Ekberg, P., & Yi, W. (2014). Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. Real-time systems, 50 (1), 48–86.

27. Ekberg, P., & Yi, W. (2016). Schedulability analysis of a graph-based task model for mixed-criticality systems. Real-time systems, 52 (1), 1–37.

28. Fleming, T. (2013, September). Extending mixed criticality scheduling. Retrieved from https://etheses.whiterose.ac.uk/5688/

29. Fonseca-Reyna, Yunior & Martinez, Yailen & Cabrera, J.M. & Méndez-Hernández, Beatriz. (2015). A reinforcement learning approach for scheduling problems. Investigacion Operacional. 36. 225-231.

30. Grinsztajn, N., Beaumont, O., Jeannot, E., Preux, P. (2021). Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling. 2021 IEEE international conference on cluster computing (cluster) (p. 70- 81). 10.1109/Cluster48925.2021.00031

31. Gritzo, R.E. (1992). A simple neural network scheduler for real-time machine task scheduling. Retrieved from https://www.osti.gov/biblio/5292113

32. Gu, C., Guan, N., Deng, Q., Yi, W. (2013). Improving OCBP-based scheduling for mixed- criticality sporadic task systems. Embedded and real-time computing systems and applications (RTCSA), 2013 IEEE 19th international conference on (pp. 247–256).

33. Guo, Z., & Baruah, S. (2015). The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems. Proceedings of the 23rd international conference on real time and networks systems (pp. 247–256).

34. Guo, Z., & Baruah, S.K. (2014). Implementing mixed-criticality systems upon a preemptive varying-speed processor. Leibniz Transactions on Embedded Systems, 1 (2), 03–1.

35. Guo, Z., Yang, K., Vaidhun, S., Arefin, S., Das, S.K., Xiong, H. (2018). Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. 2018 IEEE real-time systems symposium (RTSS) (p. 373-383). 10.1109/RTSS.2018.00052

36. Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., Silver, D. (2018). Distributed prioritized experience replay. CoRR, abs/1803.00933 . Retrieved from http://arxiv.org/abs/1803.00933

37. Johnson, L. A. (1998). DO-178B, Software considerations in Airborne Systems and Equipment Certification. Retrieved February 1, 2023, from https://www.dcs.gla.ac.uk/~johnson/teaching/safety/reports/schad.html

38. Kong, L.-F., & Wu, J. (2005). Dynamic single machine scheduling using Q-learning agent. 2005 international conference on machine learning and cybernetics (Vol. 5, p. 3237-3241 Vol. 5). 10.1109/ICMLC.2005.1527501

39. Lenstra, J., Kan, A.R., Brucker, P. (1977). Complexity of machine scheduling problems. P. Hammer, E. Johnson, B. Korte, & G. Nemhauser (Eds.), Studies in integer programming (Vol. 1, p. 343 - 362). Elsevier. https://doi.org/10.1016/S0167-5060(08)70743-X

40. Li, Y., & Chen, Y. (2009). Neural network and genetic algorithm-based hybrid approach to dynamic job shop scheduling problem. Systems, man and cybernetics, 2009. SMC 2009. IEEE international conference on (pp. 4836– 4841).

41. Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Stoica, I. (2017). Rllib: Abstractions for distributed reinforcement learning. arXiv. Retrieved from https://arxiv.org/abs/1712.0938110.48550/ARXIV.1712.09381

42. Luo, S. (2020). Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning. Applied Soft Computing, 91 , 106208.

43. Mania, H., Guy, A., Recht, B. (2018). Simple random search provides a competitive approach to reinforcement learning. CoRR, abs/1803.07055 . https://arxiv.org/ abs/1803.07055

44. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A. (2013). Playing atari with deep reinforcement learning. CoRR, abs/1312.5602 . Retrieved from http://arxiv.org/abs/1312.5602

45. Mollison, M.S., Erickson, J.P., Anderson, J.H., Baruah, S.K., Scoredos, J.A. (2010). Mixed-criticality real-time scheduling for multicore systems. Computer and information technology (CIT), 2010 IEEE 10th international conference on (pp. 1864–1871).

46. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Stoica, I. (2017). Ray: A distributed framework for emerging AI applications. CoRR, abs/1712.05889 . Retrieved from http://arxiv.org/abs/1712.05889

47. Sakr, Nourhan & Hussien, Youssef & Farid, Karim. (2021). Dual-criticality scheduling on non-preemptive, dynamic processors using RL Agents.

48. Park, T., & Kim, S. (2011). Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. Proceedings of the ninth ACM international conference on embedded software (pp. 253–262).

49. Philipoom, P.R., Rees, L.P., Wiegmann, L. (1994). Using neural networks to determine internally-set due-date assignments for shop scheduling. Decision Sciences , 25 (5-6), 825–851.

50. Sabuncuoglu, I., & Gurgun, B. (1996). A neural network model for scheduling problems. European Journal of Operational Research, 93 (2), 288–299.

51. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Prox- imal policy optimization algorithms. CoRR, abs/1707.06347 . Retrieved from http://arxiv.org/abs/1707.0634725

52. Shahrabi, J., Adibi, M.A., Mahootchi, M. (2017). A reinforcement learning approach to parameter estimation in dynamic job shop scheduling. Computers Industrial Engineer- ing, 110 , 75-82.

53. Shyalika, C., Silva, T., Karunananda, A. (2020, Sep 24). Reinforcement learning in dynamic task scheduling: A review. SN Computer Science, 1 (6), 306. Retrieved from https://doi.org/10.1007/s42979-020-00326- 5

54. Socci, D., Poplavko, P., Bensalem, S., Bozga, M. (2015). Time-triggered mixed-critical scheduler on single and multi-processor platforms. High performance computing and communications (hpcc), 2015 IEEE 7th international symposium on cyberspace safety and security (css) (pp. 684–687).

55. Sun, Y., & Tan, W. (2019, 12). A trust- aware task allocation method using deep Q-learning for uncertain mobile crowdsourcing. Human-centric Computing and Information Sciences, 9 .

56. Sutton, R.S., & Barto, A.G. (1998). Reinforcement learning: An introduction (Vol. 1) (No. 1). MIT press Cambridge.

57. Vestal, S. (2007). Preemptive scheduling of multi- criticality systems with varying degrees of execution time assurance. Real-time systems symposium, 2007. RTSS 2007. 28th IEEE international (pp. 239–243).

58. Wang, Q., Xiong, J., Han, L., sun, p., Liu, H., Zhang, T. (2018). Exponentially weighted imitation learning for batched historical data.

59. S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), Advances in neural information processing systems (Vol. 31). Curran Associates, Inc.

60. Zhang, W., & Dietterich, T.G. (1995). A reinforcement learning approach to job-shop scheduling. Ijcai (Vol. 95, pp. 1114–1120).

61. Zhou, D.N., Cherkassky, V., Baldwin, T., Olson, D. (1991). A neural network approach to job-shop scheduling. IEEE Transactions on Neural Networks, 2 (1), 175–179.

Appendix A
Instance Generation

A.1 Instance Parameters

Each MC instance is a set of jobs J, generated according to four main parameters as outlined in (S. Baruah and Guo, 2013):

- n: the total number of jobs in J,
- μ: the total load for J, which is the normalized sum of all processing times of all jobs in J.
- γ: the percentage of jobs that have χ = LO in the set J.
- ζ: the job density, which reflects the average number of active jobs at any instant.

The generator's output is a set of jobs in a matrix format, $J_{n \times 4}$, s.t; each job has the tuple r_j, d_j, p_j, and χ_j. By randomizing these parameters during the training process, the generated sets of jobs have a variety of different job structures based on the value of those parameters, thus, helping the agent become flexible in various environments.

To improve the behavior of our RL model, additional status parameters are initialized and regularly updated as follows:

- $\rho_{j,t}$: remaining processing time of job j at any given time t, initialized as $\rho'_{j,0} = \rho_j$
- l^t_j: the laxity of job j at any given time at, initialized as $l^t_j = d_j - p_j$.

Three additional workload characteristics are used in evaluation and monitoring during training and testing.

- $R^S_{\{j,t\}}$: monitors the release status of any job j at any given time t. The binary status flag is initialized to 0 and switches to 1 once $r_j \leq t$.
- $E^S_{\{j,t\}}$: monitors the execution status of any job j at any given time t. The binary status flag is initialized to 0 and switches to 1 once a job is executed.
- $S^S_{\{j,t\}}$: monitors the starvation status of any job j at any given time t. The binary status flag is set to 1 if $d_j \geq t$ and $ES_{j,t} = 0$. Initialized as $SS_{j,0} = 0$

A.2 Discretization and Filtration

In our problem, we model the time horizon in an integral manner, which better fits our environment and degradation model. S. Baruah and Guo, however, operate on a continuous time horizon. Therefore, discretization of the tuples defining a job j (r_j, d_j, and p_j) is required to fit the integral time horizon. This is achieved by choosing a time step (0.1 in our case) and multiplying it by the aforementioned tuples across the entire set J.

Converting to an integral time horizon and constraining the problem with non-preemption requires us to check that J is still schedulable after the new adjustments. Filtration starts by using the non-preemptive EDF algorithm under normal speed (v=1) to ensure the schedulability and correctness of the set J.

An empirical analysis follows the normal-speed EDF by decreasing the speed in steps aiming to find the minimum speed($v_J^{min} < 1$) capable of scheduling J. The v_J^{min} is used in degradation modeling in Section 4.3.

Table A1 shows an example of a set of jobs J after the dataset generation, discretization, and filtration processes. Table A2 shows the scheduling of the workload in normal mode and the scheduling of the subset of HI jobs in a degraded mode with the empirically calculated minimum speed of 0.75 for the set J.

Table A1: Example of an instance of generated jobs J.

Job j	r_j	d_j	p_j	x_j
1	0	30	6	0
2	1	22	7	0
3	6	27	1	0
4	10	38	2	1
5	14	75	10	1
6	19	34	4	1

Table A2: Scheduling example for the instance J in Table A1 with unique $v_{min} = 0.75$. The second column shows scheduling the jobs with speed v = 1, and the third column shows scheduling with speed v = 0.75.

Job j	$Speed_1 : t_{start} - t_{end}$	$Speed_{0.75} : t_{start} - t_{end}$
1	0-6	-
2	6-13	-
3	13-14	-
4	14-16	10-13
5	16-26	13-27
6	26-30	27-33

A.3 Training Data

Before training, the parameters for all algorithms had to be set: num_cpus_per_worker = 4, $num_{workers}$ = 1. Our RL agent is trained to be a well-rounded agent capable of dealing with various sets of jobs J with different parameters. So the dataset for each training iteration is generated dynamically, right before the training episode, using randomly drawn parameters:

- ω : $U(v_J^{min},1)$.
- μ : U(0.2,1).
- γ : U(0,1).
- ζ : U(0.25*n,0.5*n).

All these parameters were set according to the dataset generation appendix in (S. Baruah and Guo, 2013b). All the data generated dynamically during training were collected and later analyzed. We sampled some training data and found that the agent trained with the following average values:

- v_{min} = 0.85
- ω = 0.5
- μ = 0.6
- γ = 0.465
- ζ = 10.5

Appendix B
Implementation

For the implementation of the environments, we used RLLIB (Moritz et al., 2017), which is an open-source, scalable RL library. We build our own custom environment using RL Gym standards for the online model tailored to our problem's dynamic non-preemptive nature. The main parts of custom environments are the initialization and reset functions, the step function, which is the primary core of our model, and a few helper functions. The initialization and reset functions both set the state of the environment back to default by setting the time to zero, resetting all variables, generating data dynamically, and defining the observation space and action space.

The initialization of the online environment can be seen in Algorithm 1. The first three lines initiate the processor speed to v = 1, JobNum depending on the number of jobs n for each experiment, and ω for the environment, which is randomly generated for each workload instance J. The following three lines define the parameters for workload generation mentioned in Appendix A. The GenerateData() function returns the workload for the next training iteration and its minimum possible scheduling speed v_J^{min}. Lines 9-10 define the action space and observation space for the environment. The reset function behaves similarly to the initialization, with minor differences related to how the RL library operates.

During initialization, reset, and step, other functions are required that help with the flow of the environment. These are some of the helper functions used in our environment:

- get_obs(): updates the observation space of the environment based on the updates in the online buffer.
- update_available(): updates the action space of the environment
- UpdateBuffer(): updates all the jobs currently in the buffer by adding newly released jobs and removing expired or starved jobs. It fills empty slots with dummy values -1.
- update_workload(): updates the original workload instance J, which is later mapped to the online buffer. The updated values are the ρ_j^t, l_j^t, $RS_{j,t}$, $ES_{j,t}$, $SS_{j,t}$ for any given time t.

The step function, the core of our RL environment, is responsible for all the logic behind training the agent. The agent enters the step function with a given action based on the outcome of the RL algorithm and the observation space. Algorithm 2 shows the pseudo-code for the step function. Lines 2-6 check if there aren't any jobs currently released in the buffer, it moves to the next time step, updates the buffer, and moves on to the next action. Lines 8-11 punish the agent heavily for picking a dummy job if a real job exists in the buffer. For example, if the buffer had [1 3 -1 -1 -1] and the agent picked one of the dummies values -1 instead of job 1 or job 3, it would be heavily punished. Lines 13-21 first punish the agent for

picking a starved or an executed job, then check if a degradation happened when the agent chose this current action, as it will affect the reward. Finally, it moves in the time horizon towards r_j if it is greater than t. Lines 22 - 29 resemble the execution of the job. At each time step, the function adjusts the remaining processing time ρ^t_j based on the current processor speed ($v \leq 1$) and moves ahead in the time horizon to the end of the job if it has enough time to finish or to the deadline if it will not have enough time. For the preemption model, the agent does not move to the end of the job processing time but rather it could preempt if degradation happens while executing a low-criticality job. If at any point, the job reaches its deadline without finishing, the agent is punished. If the job completes successfully, we move to the reward function to determine the reward based on Figure 4 and as seen in Lines 30-33. Finally, lines 35-38 update the set of jobs J, the observation space, and the buffer of all currently released jobs to prepare for the following action.

Algorithm 1 Initialization Pseudocode

1: $v = 1$
2: JobNum $= n$
3: DegradationChance=random(low=0.05,high=0.95)
4: TotalLoad=random(low=0.2,high=1)
5: LoPercentage=random(low=0,high=1)
6: JobDensity=random(low=0.25*n,high=0.5*n)
7: WorkloadInstance, MinSpeed = GenerateData()
8: Define Online Buffer()
9: Define Action Space()
10: Define Observation Space()

Algorithm 1: Initialization Pseudo Code

Algorithm 2 OnlineStepFunction(action)

CheckDone()
if *currently_released_jobs* $= 0$ **then**
    $AgentReward \leftarrow 0$
    $t+ = 1$
    UpdateBuffer()
    Move to next action
end if
if OnlineBuffer[Action] = -1 **then**
    $AgentReward \leftarrow -25$
    CheckDone()
    Move to next action
else
    if $S^S_{j,t}$ or $E^S_{j,t} = 1$ **then**
        $AgentReward \leftarrow -10$
        $DoneFlag \leftarrow 1$
    end if
    $t \leftarrow Max(t, r_j)$
end if
while $p^t_j > 0$ **do**
    if $t > d_j$ **then**
        $DeadlineFlag = 1$
        break
    end if
    CheckDegradation()
    $rem = min((d_j - t), (p_j/s_v))$
    $p^t_j = p^t_j - rem$
    $t+ = rem$
    if $PreemptionFlag == 1$ **then**
        if $speed < 1$ and $\chi_j = 0$ **then**
            PreemptAgent
        end if
    end if
end while
if $DeadlineFlag = 1$ **then**
    $AgentReward \leftarrow -1$
else
    $RewardAccordingtoDecisionTree$
end if
UpdateReleasedJobs(CurrentTime)
UpdateStarvedJobs(CurrentTime)
UpdateBuffer()
UpdateObservationSpace()

Algorithm 2: Online Step Function