THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le *31/01/2023* **par :**

Igor FONTANA DE NARDIN

On-line scheduling for IT tasks and power source commitment in datacenters only operated with renewable energy

---

JURY

Premier MEMBRE	Professeur d'Université	Rapporteur
Second MEMBRE	Professeur d'Université	Rapporteur
Troisième MEMBRE	Professeur d'Université	Examinateur
Quatrième MEMBRE	Professeur d'Université	Examinateur
Cinquième MEMBRE	Professeur d'Université	Examinateur

École doctorale et spécialité :
MITT : Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Unité de Recherche :
IRIT (UMR 5505) et Laplace (UMR 5213)

Directeur(s) de Thèse :
Patricia STOLF et *Stéphane CAUX*

Rapporteurs :
Premier RAPPORTEUR et *Second RAPPORTEUR*

Acknowledgments

Acknowledgments

Abstract

Abstract

Résumé

Résumé

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Global warming is one of the biggest challenges humanity is facing. A recent rapport shows that we are walking toward a global mean temperature increase by 2100 of 2.8°C, well above the 1.5°C defined by the Paris Agreement [1]. A previous rapport predicts the rise in mean global temperature will be around 1.8°C even after implementing all announced Paris Agreement goals [2]. Achieving 1.5°C demands an engagement of all sectors to reduce greenhouse gas (GHG) emissions. GHG is generated during the combustion process of fossil fuel, one of the world's main sources of energy production [3].

One significant GHG emitter is the Information and Communications Technology (ICT) sector. It produces around 1.8-2.8% of the world's total GHG [4]. Inside ICT, Data centers and transmission networks are responsible for nearly 1% of global energy-related GHG emissions [5]. The data center sector is one of the most electricity-expensive ICT actors due to its uninterrupted operation. A rapport revealed that Google data centers consumed the same amount of energy as the entire city of San Francisco in 2015 [6]. In addition, the situation tends to get even worse due to the improvements reduction in processor technologies and the predicted expansion of internet usage [4, 7].

Big cloud providers such as Google and Amazon are trying to reduce energy consumption and increase the power coming from Renewable Energy Sources (RES) [8]. RES is the most encouraging method to eliminate fossil fuel use [3]. Renewable sources generate energy from clean sources such as biomass, hydropower, geothermal, solar, wind, and marine energies [9, 10, 11, 12, 13]. A significant drawback of RES is the weather conditions dependency, creating power intermittence. These providers smooth this intermittence by not migrating entirely to RES, maintaining a connection to the grid [11]. Therefore, they are not 100% clean. A renewable-only data center must consider this intermittence in its decision-making. Another source of uncertainty comes from the user's demand. Users can send their requests at any time. Providing high availability is a challenge for a renewable-only data center.

A way to reduce the impact of RES power production intermittence is by adding storage elements [11]. Batteries and hydrogen tanks can shift generation and/or consumption over time. A renewable-only data center demands a massive storage capacity [11]. For example, Google plans to use energy from 350 MW solar panels connected to a storage system with 280 MW (megawatt-hour energy capacity undisclosed) [14]. While helping to deal with RES intermittence, storage management introduces another level of decision. For example, a battery can store energy during the day and then use the energy stored at night. Nevertheless, the demand during the day could be higher than at night, so maybe

it is better to use the energy during the day. This is another big challenge for migrating to a 100% clean data center.

Some works propose ways to deal with both demand and weather uncertainties using predictions [15, 16, 17, 18]. Forecasting the upcoming requests and the weather helps to plan storage usage. They use these predictions to maximize renewable usage but with the grid as backup. All these works are valuable and important to optimize renewable usage. However, the forecast can vary from the actual values. Other works focus on reacting to real events [19, 20, 21, 22]. They try to minimize the data center operational cost, maximize renewable usage, increase the revenue of job execution, or improve the Quality of Service. Usually, they define ways to schedule the jobs, optimizing their objective. However, they focus on short-term decisions without long-term management. Since these works also have the grid as backup, storage management is not a concern. Some works mix predictions with reactive actions. For example, Goiri et al. [18] propose a scheduling algorithm that predicts solar power production and uses it to define the best moment to start new jobs, using brown energy (from the grid) when necessary. Venkataswamy et al. [23] created a job scheduler that defines job placement according to the available machines. The available machines are given by a fixed plan (which can use power from renewable, batteries, or grid), with no modifications.

Few research initiatives are investigating how to design and operate a renewable-only data center. One of them is the ANR Datazero2 project [24]. This project aims to define a feasible architecture to maintain a renewable-only data center. This architecture includes several elements to provide energy to the IT servers, such as Wind turbines, Solar panels, Batteries, and Hydrogen tanks. Considering the decision-making, Datazero2 divides the problem into two parts: offline and online. The offline part finds the optimal solution for the problem, which takes time. On the other hand, the online part reacts to real events, making fast decisions on-the-fly. The offline module predicts power demand and production. Using these predictions and considering long-term constraints, this module creates a power and IT plan for the near future. This decision-making process can take several minutes.

The online module schedules the users' jobs, using the offline plan as a guide. Online is the only one that knows exactly the jobs submitted to the data center. So, it needs to place them in the available servers. Online could just apply the offline plan without modifications. However, this behavior would impact the Quality of Service (QoS). Online can improve the QoS, using more energy from storage to turn on more servers (to run more jobs) or speed up the running servers (to finish jobs earlier). Besides, online must be renewable production aware. For example, online can identify a lower production that can dry the energy storage faster, so it must reduce its usage. Finding a good trade-off between QoS and energy storage management is even harder in online mode, which demands fast decisions. In this thesis, we focus on these online decisions. The goal is to design and prove the efficiency of a novel approach for scheduling users' jobs, finding a good trade-off between QoS and energy storage management.

1.2 Problem Statement

A data center powered by renewable energy demands several levels of decision. Several works aim to optimize some of these decisions. We can cite demand and production predictions, cost optimization, sizing, shifting demand, battery management, admission control, and job scheduling, to mention a few. Usually, these works introduce a link to the grid, using it as a backup to cope with peak demand. Removing the grid of the

context adds several challenges. This context increases the need for predictions to manage weather and workload uncertainties. Another key element in renewable-only data centers is energy storage. Aligning prediction and energy storage elements allows it to define the best strategy to handle users' requests. However, actual demand and production can vary from the predictions. So, the online module must react to the actual values. This reaction can improve the QoS (e.g., when there is more production than expected). In addition, online reaction optimally absorbs the impact of the demand and production power to stay close to the offline plan.

Figure 1.1 illustrates all the elements in the decision process. We consider only renewable sources and energy storage elements without grid connection. An offline optimization gives an offline plan using production and demand prediction. The offline plan has a limited size named time window (e.g., three days). So, offline suggests actions to online during this time window. Online receives the actual renewable production from wind turbines and solar panels. Online adapts energy storage usage according to the actual production. Since hydrogen has a longer start-up time, it is difficult to manage it in online mode. Therefore, we let hydrogen usage from the offline optimization, using it to provide energy during periods with low renewable production (e.g., during the winter). So, online decides about battery usage only.

Battery management introduces two new challenges regarding the Battery's State of Charge (SoC):

- *Challenge 1:* SoC means the level of charge of a battery relative to its capacity. A good practice to extend the battery's lifetime is to avoid drying or overcharging it [25]. Therefore, our solution must maintain the SoC between reasonable levels;
- *Challenge 2:* Online has the entire time window to make modifications in battery usage. However, it must finish the time window close to the expected SoC (given by the offline plan). Since the data center runs continuously, it is not viable to always use more battery than expected for every time window. Therefore, our solution must finish with more or equal SoC than planned.

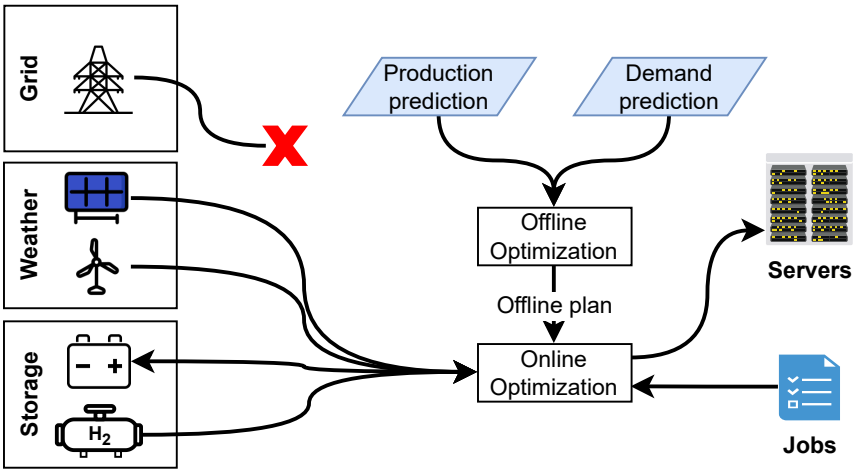

Figure 1.1: Problem overview. Online receives an offline plan, the actual renewable production, and the users' jobs. It must define energy storage usage, job placement in the servers, and server speed.

On the IT side, online receives the jobs from the users and must schedule them on the available servers. Online receives an offline plan for server configuration (machine on/off

and speed). However, it can modify the server configuration to react to incoming events (e.g., more production, demand peak). Changing the speed of a server is possible due to the Dynamic Voltage and Frequency Scaling (DVFS) technique. DVFS allows servers' speed reduction, spending less energy. However, putting a job on a server with a decreased speed can impact the job QoS. To sum up, online must manage the battery (maintaining the SoC between thresholds and finishing the time window with the battery level close to the target), schedule the jobs, and balance the servers' speed.

This thesis' objectives are:

1. Making online modifications in the power decisions given by the offline plan, coping with the uncertainty coming from renewable production and workload demand;

2. Mixing power, scheduling, and server online decisions, turning the scheduling energy storage aware. This mix allows the scheduling to make better decisions than usual algorithms;

3. Adding the predictions to the online decision;

These goals help to find a better trade-off between QoS and energy storage management. Different contributions address these questions in this manuscript.

1.3 Main contributions

Proposing a simulation environment

A crucial step to simulate data center management is defining the workload, weather, and server configuration, in a complete simulation tool. We detail in Section 3 the simulation environment, providing a dedicated framework. Regarding the workload, some traces are used in literature, such as Google [26], Parallel Workloads Archive [27], and Alibaba [28]. We propose a trace from Parallel Workloads Archive named Metacentrum [29]. We detail the filtering process of this trace. Considering the weather, it is possible to collect data from everywhere in the world. We present the methodology to generate power production from a NASA trace, using the framework Renewables.ninja[1] [30]. The third input is the server configuration. We demonstrate the data collected from a server in GRID5000[2] used in this thesis. Finally, we present the simulation tool named BATSIM[3], based on SIMGRID[4]. We introduced in this simulation tool the modifications needed to manage battery and power production. The whole of these data and definitions allows future work inside and outside the Datazero2 project.

Defining offline power and IT decisions

As illustrated in Figure 1.1, an important part is the offline plan. This plan must consider the power and demand predictions to define the actions for the next time window. We demonstrate in Section 3 a model to use both predictions. We separate the problem into two parts. First, we present the optimization problem to define power engagement, giving a power prediction. This optimization problem results in expected renewable power production, energy storage usage, and expected SoC. The sum of the expected renewable

[1] https://www.renewables.ninja/
[2] https://www.grid5000.fr
[3] https://batsim.org/
[4] https://simgrid.frama.io/

power production and energy storage usage is named the power envelope. The second part is the IT servers' state (on/off) and speed definition. This optimization problem defines the state and speed according to the power envelope. The objective of this optimization problem is to maximize the servers' speed. The results of both optimizations are the input for the online module.

Reacting to power fluctuations

Given the result of the optimization problem, next, we propose a heuristic to react to the power fluctuations. Since there is no perfect prediction, one source of divergence is the difference between the prediction and actual values. This divergence occurs in both power demand and production. Additionally, the offline model considers that the servers will maintain constant power usage. However, the server consumption can vary according to the scheduling and/or job. Yet, the scheduling can modify the battery usage to improve the QoS (e.g., avoiding killing jobs). Considering all these sources of power fluctuations, the heuristic must adapt the usage, aiming to approximate the state of charge of the target level at the end of the time window. Since this is an online problem, we can not re-run the offline optimization solution with the actual values. Therefore, we propose four policies to compensate for these divergences in the power envelope. Each one finds a different moment in the future to place the compensation.

Learning the actions to deal with power fluctuations

The four compensation policies apply the same behavior throughout the entire execution. However, different moments inside the time window can demand distinct policies. So, our next goal is to learn when to use each policy. So, we introduce two Reinforcement Learning (RL) algorithms to discover the best mix of policies. Considering each policy as RL's action, we present the RL's state and reward. The premise of applying RL is that optimizing the decisions locally generates a global optimal. In other words, if the algorithm chooses the best action each time, in the end, we will have the best results. We implemented two well-known RL algorithms named Contextual Multi-Armed Bandit and Q-Learning. We present the learning results and a comparison between the RL algorithms and random choices.

Defining energy storage-aware scheduling using production and demand predictions

Finally, the last contribution is an energy storage-aware scheduling heuristic. This algorithm is based on the well-known EASY-Backfilling. The algorithm is named BEASY (Battery-aware EASY backfilling). BEASY uses the predictions given by the offline to predict dangerous moments, where it must be careful in the scheduling. Furthermore, we introduce another level of validation, verifying if the servers allocated to the job would be available during the entire execution. Regarding power compensations, it creates several possible scenarios of production and demand using the forecasts. According to these scenarios, the heuristic finds the best moment to make the compensations. For example, BEASY tries to reduce the usage before the moments when the predictions indicate that the battery could be lower than a critical value. This heuristic mixes all decisions providing a well-balanced answer to the online multi-objective problem.

1.4 Publications and Communication

Accepted Peer Reviewed Conferences:

- I. F. de Nardin, P. Stolf and S. Caux, "Adding Battery Awareness in EASY Backfilling", 2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Porto Alegre, Brazil, 2023.
- I. F. de Nardin, P. Stolf and S. Caux, "Analyzing Power Decisions in Data Center Powered by Renewable Sources", 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Bordeaux, France, 2022, pp. 305-314;
- I. F. de Nardin, P. Stolf and S. Caux, "Evaluation of Heuristics to Manage a Data Center Under Power Constraints", 2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC), Pittsburgh, PA, USA, 2022, pp. 1-8;
- I. F. de Nardin, P. Stolf and S. Caux, "Mixing Offline and Online Electrical Decisions in Data Centers Powered by Renewable Sources", IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society, Brussels, Belgium, 2022, pp. 1-6;
- I. F. de Nardin, P. Stolf and S. Caux, "Smart Heuristics for Power Constraints in Data Centers Powered by Renewable Sources", Conférence francophone d'informatique en Parallélisme, Architecture et Système (COMPAS 2022), Jul 2022, Amiens, France. paper 7.

Others Disseminations:

- Talk: Analyzing Power Decisions in Data Center Powered by Renewable Sources, GreenDays@Lyon, March 2023.

1.5 Dissertation Outline

The remaining dissertation has the following organization:

Chapter 2 - Context and Related Works: This chapter presents the fundamentals to understand this dissertation. Considering the scope of the topic, the context consists of four parts. First, we introduce the context of global and ICT GHG emissions. Then, we describe renewable energy as an alternative to replace brown energy. After, we explain the usage of renewable to power a data center. Then, we define the uncertainties of weather and workload in a renewable-only data center. This last part also clarifies the importance of using predictions but with an online adaptation. After presenting the context, we introduce a list of works that solve part of our problem, highlighting the existing gaps in the state-of-the-art;

Chapter 3 - Modelling, Data, and Simulation: In this chapter, we describe the model to deal with several elements that compose a renewable-only data center. Datazero2 creates a division between Offline and Online decisions. We present the model to deal with offline decisions using predicted power demand and production. Then, we detail the output of Offline used by the Online. Finally, we define the Online model, which englobes the job scheduling and modifications in the Offline plan. After describing the model, we explain the source of the different data (e.g.,

workload, weather, servers) applied in the simulations. We present an explanation of the work done in the traces of the literature. Finally, we present the simulation tools used in this work;

Chapter 4 - Introducing Power Compensations: This chapter describes the proposed algorithm to react to power uncertainties. We created four heuristics to find the best place to compensate for battery changes, which aim to reduce the number of killed jobs and the distance between the battery level and the target level. The results presented are related to the publications [31] and [32];

Chapter 5 - Learning Power Compensations: This chapter presents the idea and the results of the introduction of Reinforcement Learning (RL) in the power compensation problem. We propose two RL algorithms (Q-Learning and Contextual Multi-Armed Bandit) to learn the best moment to compensate;

Chapter 6 - Adding Battery Awareness in EASY Backfilling: This chapter explains a heuristic to mix scheduling and power compensation decisions. This heuristic is based on the EASY Backfilling scheduling algorithm but considers the battery's State of Charge to make better decisions;

Chapter 7 - Conclusion and Perspectives: Finally, in this chapter, we summarize the contributions of this work, providing a discussion about future works.

Chapter 2

Context and Related Work

Contents

2.1 Global Warming and ICT Role

Global warming is one of the most critical environmental issues of our day [33]. Global warming is the effect of human activities on the climate, mainly the burning of fossil fuels (coal, oil, and gas) and large-scale deforestation [33]. Both activities have grown immensely since the industrial revolution. The burning of fossil fuels process results in greenhouse gas emissions [3]. Today, fossil fuels are one of the world's main sources of energy production, emitting more and more GHG [3]. GHG stays in the atmosphere creating a layer as a blanket over the planet's surface. Without this blanket, the Earth can balance the radiation energy from the sun and the thermal radiation from the Earth to space [33]. However, this human-generated blanket imposes a barrier to the thermal radiation from the Earth, heating the planet. All this process works as a greenhouse which is the reason for the name greenhouse gas [33].

This situation brings us to United Nations Climate Change Conference (COP21) in Paris, France, on 12 December 2015. At this conference, 196 parties signed the Paris Agreement aiming to [34]:

1. Reduce global greenhouse gas emissions substantially, limiting the global temperature increase in this century to 2°C while pursuing measures to limit the growth even further to 1.5°C;

2. Review countries' commitments every five years (through the Nationally Determined Contribution, or NDC);

3. Provide financing to developing countries to mitigate climate change, strengthen resilience, and enhance their abilities to adapt to climate impacts.

These are ambitious but necessary objectives. Since then, countries and organizations have proposed several actions and pledges. However, a recent report indicates that the actual world's effort is not enough [2]. Figure 2.1 shows GHG emission and temperature estimations. We could see that there is a small reduction in emissions increase tendency. Nevertheless, this figure estimates that real-world actions based on current policies will lead to an increase of somewhere between 2.6 and 2.9°C by 2100. Another recent report confirms the estimation of 2.8°C by 2100 [1]. This estimation is well above the 1.5°C pursued by the Paris Agreement. Considering the targets proposed by the countries through NDC, the temperature will be around 2.4°C. In a scenario based on 2030 NDC targets and submitted and binding long-term targets, the prediction is a temperature of 2°C by 2100, the limit proposed by the Paris Agreement. The report forecasts an optimistic scenario considering the effect of full implementation of all announced targets (e.g., net zero targets) in about 140 countries. Even in this optimistic scenario, the estimated temperature would be 1.8°C. The situation tends to be even worst with the gold rush for gas [35]. The report indicates that in 2022 we arrived at 1.2°C warming [2].

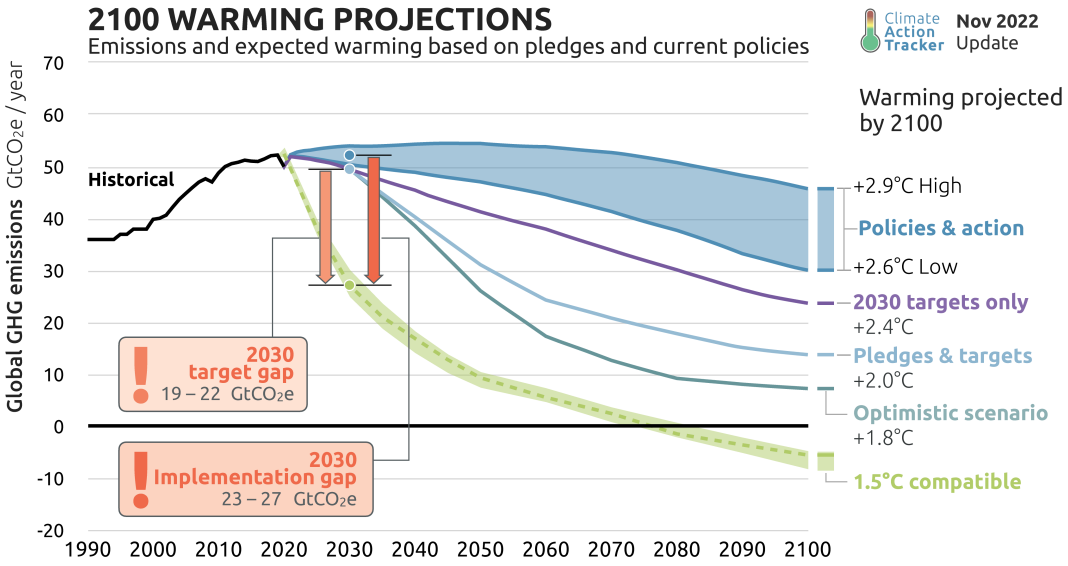

Figure 2.1: Estimated global GHG emissions [2].

We have started to feel the impacts of global warming on humanity, such as heatwaves, droughts, and floods, impacting flora and fauna directly [36, 37]. In a cascade effect, this increases food and water insecurity worldwide [37, 38]. In addition, high temperatures increase mortality, impact labor productivity, impair learning, increase adverse pregnancy outcomes possibility, increase conflict, hate speech, migration, and infectious disease spread [39]. Therefore, an increase of the temperature by 2.7°C as forecasted would impact one-third (22–39%) of the world's population by 2100 [39]. Climate change has already impacted around 9% of people (>600 million) [39]. Reducing global warming from 2.7 to 1.5°C results in a ∼5-fold decrease in the population exposed to unprecedented heat (mean annual temperature ≥29°C) [39]. Thus, all sectors must reduce their GHG emissions as much as possible.

Information and Communication Technology is one of these sectors which has accelerated growth in the last 70 years. Unesco defines ICT as [40]:

> "Information and communication technologies (ICT) is defined as a diverse set of technological tools and resources used to transmit, store, create, share or

> exchange information. These technological tools and resources include computers, the Internet (websites, blogs, and emails), live broadcasting technologies (radio, television, and webcasting), recorded broadcasting technologies (podcasting, audio and, video players, and storage devices), and telephony (fixed or mobile, satellite, visio/video-conferencing, etc.)."

Regarding the ICT role in GHG emissions, the global share is around 1.8%-2.8%, or 2.1%-3.9% considering the supply chain pathways in 2020 [4]. The situation tends to get even worst, driven by the boom in Internet-connected devices. A Cisco report indicates that the Internet had 3.9 billion users in 2018 [7]. The same report predicts an increase to 5.3 billion in 2023 (66 percent of the global population). Furthermore, they predicted 3.6 networked devices per capita in 2023, up from 2.4 networked devices per capita in 2018. However, International Telecommunication Union (ITU), a United Nations specialized agency for ICTs, indicates that we arrived at 5.3 billion connected users in 2022 due to the COVID-19 pandemic [41]. But will the growth in internet users increase GHG emissions? Andrae and Edler [42] and Belkhir and Elmeligi [43] agree that this growth could lead to an increase in GHG emissions. Figure 2.2 shows the predictions of both works.

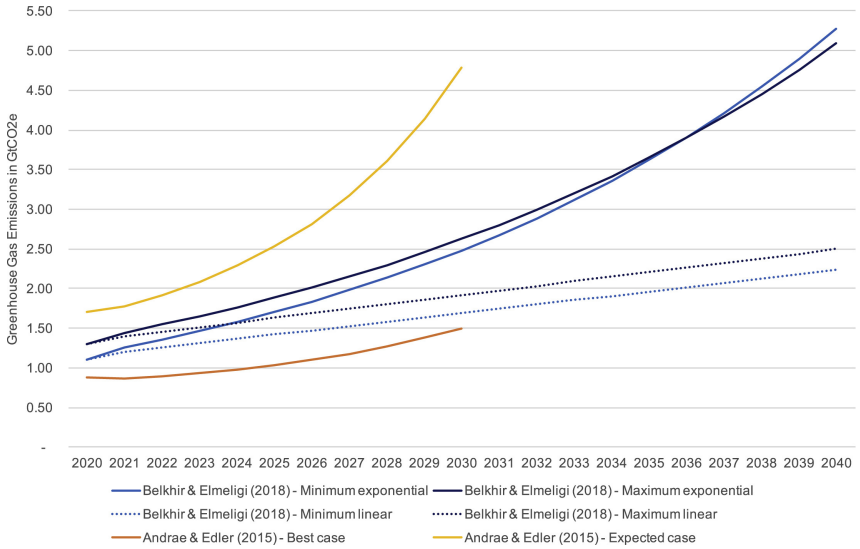

Figure 2.2: Projections of ICT's GHG emissions from 2020 [4].

This figure illustrates the contraction in the Paris Agreement targets (see Figure 2.1) and the predictions about usage in the ICT sector. In all forecasts of Figure 2.2, the tendency is the growth of emissions. However, ICT needs to reduce its emissions drastically. Figure 2.3 illustrates the carbon emission share if the ICT stays at the same level as 2020 and the other sectors decrease their emissions. Without changes, ICT would have 35.1% of global emissions in 2050. So, ICT must move towards reducing its emissions. One of the biggest GHG emitters inside the ICT sector is data centers [4]. IBM defines the data center as "a physical room, building or facility that houses IT infrastructure for building, running, and delivering applications and services, and for storing and managing the data associated with those applications and services" [44]. The International Energy Agency (IEA) defines data center as [5]:

> "Data centers are facilities used to house networked computer servers that store, process and distribute large amounts of data. They use energy to power

both the IT hardware (e.g., servers, drives, and network devices) and the supporting infrastructure (e.g., cooling equipment)."

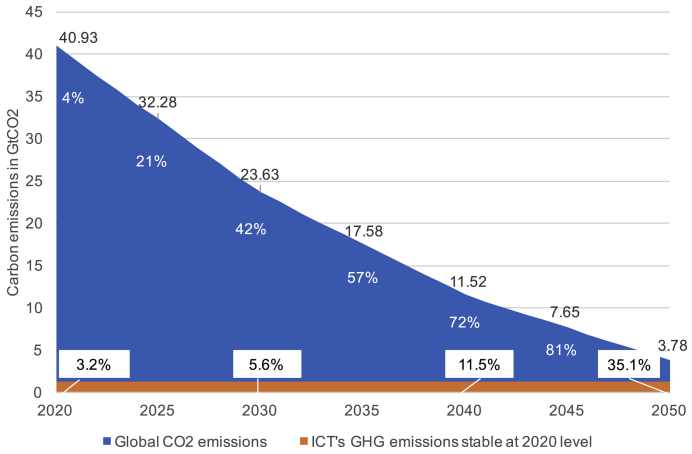

Figure 2.3: ICT's emissions, assuming the 2020 level remains stable until 2050, and global CO2 emissions reduced in line with 1.5°C [4].

Data centers are very energy consumers. IEA published an article indicating that data centers and networks were responsible for almost 1% of energy-related GHG emissions in 2020 [5]. Google data centers consumed the same amount of energy as the entire city of San Francisco in 2015 [6]. Global data center electricity usage in 2021 was 220-320 TWh, corresponding to 0.9-1.3% of the global demand [5]. For example, the domestic electricity consumption of Italy was 300 TWh in 2021 [45]. In Ireland, electricity consumed by data centers went from 5% of the total electricity consumption in 2015 to 14% in 2021 [46]. Denmark predicts tripling data center consumption, corresponding to 7% of the country's electricity use [47].

Despite the strong growth in demand, data center energy usage has only moderately grown [5]. A reason that explains it is the improvements in IT energy consumption (in hardware and software). These improvements allowed a boost in microchips' speed with a reduction in their power consumption, letting big data center companies cope with the peak in demand. Gordon Moore predicted in 1965 (Moore's law) that [48]:

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."

Even if he predicted it just until 1975, it is the case nowadays. However, the future is uncertain, and the community is divided to confirm continuous efficiency improvements [4]. While Andrae and Edler [42] and Belkhir and Elmeligi [43] expected an ending in power-consuming improvements (indicated in Figure 2.2), Malmodin and Lundén [49] are more optimistic. They suggest that ICT's carbon footprint in 2020 could halve by 2030. To achieve that, he considers two key factors. First, the improvements will continue. However, all these improvements are not enough because of the increase in usage (demonstrated in Figure 2.2). Second, the migration to renewable sources.

2.2 Renewable Energy Sources

The ICT migration to renewable energy sources (RES) is one of the factors that helped reduce the growth in GHG emissions despite the rapidly growing demand for digital services [5]. RES is one of the principal solutions to decarbonize electrical production [3, 11]. RES is also named green energy, in contrast to brown energy from fossil fuels. Basically, RES generates energy from natural sources, such as solar, wind, geothermal, hydropower, wave and tidal, and biomass [9, 10, 11, 12, 13]. These natural sources have a low impact on GHG emissions. For example, manufacturing is the stage with higher emissions for wind and solar [50]. So, these components could produce energy with no or low GHG emissions. The renewable term comes from the idea that these sources are constantly replenished. On the other hand, fossil fuels are non-renewable because they need hundreds of millions of years to develop. In the Net Zero Emissions by 2050 Scenario, RES is responsible for one-third of the reductions between 2020 and 2030 [51]. Some countries focus on nuclear power plants to produce energy. Even if nuclear power is a low carbon emissions energy source, it introduces the risk of accidents and environmental impacts of radioactive wastes [52]. It also consumes a lot of water.

The biggest challenge of implementing RES is its intermittence [11]. Since RES production comes from nature, it depends on the climate conditions. For example, there is no power production from solar during the night. There are two approaches for implementing RES production: on-site and off-site generation [53]. On-site generation uses local renewable resources, and off-site takes resources available on the grid. In an off-site generation, it is not possible to guarantee that the incoming energy is from RES since the grid mixes all types of power generation [11]. Giant cloud providers (e.g., Google, Amazon, and Facebook) invest in solar and wind power plants in an off-site approach [8, 14, 54]. So, they could say that they provide RES to the grid with the same amount that they expend. However, they transfer the RES uncertainty problem to third parties [11]. For example, in a case with a peak in demand, they will use the power from the grid, renewable or not. Therefore, they are still non-renewable-dependent.

2.3 Renewable-only Data center

Since data centers have a controlled infrastructure (e.g., the number of servers tends to stay constant, the network is stable, changes in the infrastructure are planned, etc.), they are a good target to migrate to a renewable-only environment [11]. However, creating a renewable-only data center imposes several challenges. In a data center using grid energy, a manager receives the tasks from the users and defines a placement. This manager focuses on maintaining a good QoS. However, in a renewable-only data center, the manager also needs to control the available energy usage. In this kind of data center, all the generation is on-site without backup from the grid. Nevertheless, the production and demand can not match. Figure 2.4 exemplifies the mismatch between the power demanded by a data center and power generation. This mismatch requires a production (electrical) or a load (IT) shift. This thesis focuses on the manager in a renewable-only data center. We will present both electrical and IT elements needed for this kind of data center.

2.3.1 Electrical elements

As mentioned before, different renewable sources can generate power. We focus on wind and solar since they are the most prominent in current years [51]. For wind turbines, the

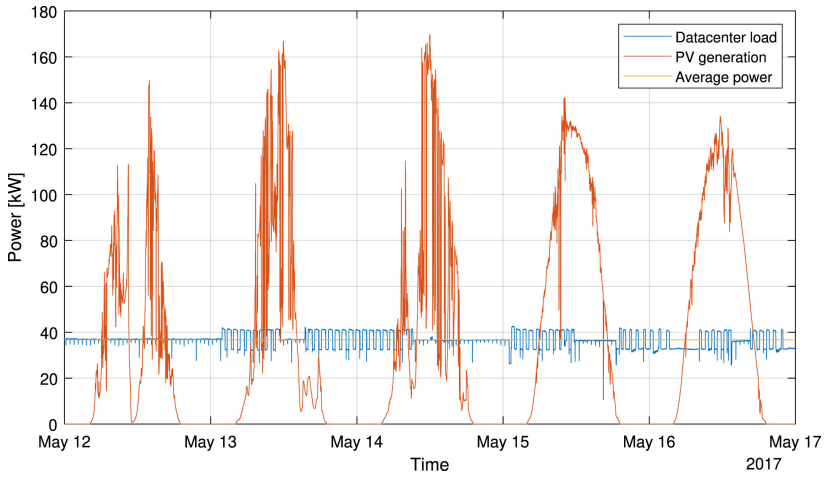

Figure 2.4: Comparison of small data center load and the generation from a theoretical photovoltaic in Belfort, France. Both load and production have the same average value [11].

wind speed is crucial. Equation 2.1 gives the power output $P_{wt}(t)$ at the moment t of a wind turbine, given the wind speed v [55, 56, 57].

$$P_{wt}(t) = \begin{cases} 0 & v \leq v_{in} \text{ or } v(t) > v_{out} \\ P_{WT,rated} \times \frac{v(t)-v_{in}}{v_{rated}-v_{in}} & v_{in} < v(t) \leq v_{rated} \\ P_{WT,rated} & v_{rated} < v(t) \leq v_{out} \end{cases} \tag{2.1}$$

Where:

- $P_{wt}(t)$: Power generated by a wind turbine (kW);
- v: Wind speed (m/s);
- v_{in}: Cut-in wind speed (m/s);
- v_{out}: Cut-out wind speed (m/s);
- v_{rated}: Speed related to wind turbine nominal power (m/s);
- $P_{WT,rated}$: Wind turbine nominal power (kW).

If the wind speed v is lesser or equal to the cut-in v_{in} or greater than the cut-out v_{out}, it does not produce power. It tests the cut-out v_{out} to protect the generator. If the speed v is greater than the cut-in v_{in} and lesser or equal to the rated speed v_{rated}, it generates proportionally to the rated power $P_{WT,rated}$ and rated speed v_{rated}. Finally, if the speed v is greater than the rated speed v_{rated} and lesser or equal to the cut-out v_{out}, it produces constant power $P_{WT,rated}$.

Regarding solar production, the photovoltaic (PV) system uses solar panels to generate power from solar irradiance. Equation 2.2 demonstrates how to calculate the output power of a solar panel $P_{pv}(t)$ [56, 57, 58].

$$P_{pv}(t) = P_{R,PV} \times (R/R_{ref}) \times \eta_{PV} \tag{2.2}$$

Where:

- $P_{pv}(t)$: Power generated by each PV panel (W);
- $P_{R,PV}$: PV panel Nominal power (kW);
- R: Solar irradiance (W/m^2);
- R_{ref}: solar irradiance at reference conditions. Usually set as 1000 (W/m^2) [56];
- η_{PV}: PV efficiency, including power eletronics and power point control [57, 58].

Some works simplify PV efficiency by applying a constant value [16, 56]. Equations 2.1 and 2.2 demonstrate that both wind turbines and solar panels depend on wind speed and solar irradiance, respectively. So, the weather conditions drive how much power both can generate.

Due to the weather intermittence, it is necessary to introduce storage elements. These storage elements allow for shifting generation and consumption over time [11]. For example, power coming from wind turbines during the night can be stored and used during the day. Big companies are investing in massive storage elements. An example is Google which is planning a 350 MW solar plant in Nevada connected to a storage system with a maximal power of 280 MW [14]. There are different types of storage with advantages and drawbacks [59]. One of them is hydropower and underground compressed air storage. However, this kind of storage is very geographical, geological, and terrain dependent, which makes it inappropriate to use in data centers [11]. Another type is the very short-term storage such as flywheels or supercapacitors. These storages can output and absorb energy over ms to minutes [59]. They are very suitable for maintaining power stability but not for storing energy for a larger time horizon (e.g., hours or days) [11]. In this thesis, we focus on the batteries and Hydrogen Storage System (HSS).

Batteries are electrochemical devices that store energy in chemical form [11, 59, 60]. They are very reactive because they do not need a warm-up to store/generate power. Batteries are good for short-term storage scenarios (e.g., several hours, day/night cycles) [11]. However, they are inappropriate for longer periods due to their self-discharge rate and low energy density [11, 60]. Historically, Uninterruptible Power Supply (UPS) added batteries to avoid the server's blackout, doing a soft shutdown that avoids several problems, such as data loss, data corruption, work loss, etc. A problem with batteries is the degradation in capacity and performance over time, requiring battery replacement [11]. A way to extend battery life is by avoiding charging/discharging too extensively [25]. There are some methods to model the energy level inside the battery, such as energy-based, Current-based, or State of Charge [11]. We focus on the State of Charge since it represents the percentage of energy inside the battery according to its capacity (e.g., 100% means battery full and 0% dry). Xu et al. present results showing that maintaining SoC at a narrow range reduces battery degradation [25]. However, using a narrow range would reduce the battery's effectiveness because it can deliver less energy to deal with intermittence. So, the battery SoC must be maintained within a range considering this trade-off. Equations 2.3 and 2.4 demonstrate how to calculate the State of Charge [16].

$$E_{bat}(t) = (E_{bat}(t-1) \times (1-\sigma)) + (P_{ch}(t-1) \times \eta_{ch} \times \Delta t) - (P_{dch}(t-1) \times \eta_{dch} \times \Delta t) \quad (2.3)$$

$$SoC(t) = \frac{E_{bat}(t)}{C_{bat}} \times 100 \quad (2.4)$$

Where:

- Δt: Duration of t (h);
- $E_{bat}(t)$: Energy in the battery at instant t (kWh);
- $P_{ch}(t)$: Charging power (kW);
- $P_{dch}(t)$: Discharging power (kW);
- σ: Battery self-discharge rate (%);
- η_{ch}: Battery charge efficiency (%);
- η_{dch}: Battery discharge efficiency (%);
- C_{bat}: Capacity of the battery (kWh);
- $SoC(t)$: State of Charge at instant t (%);

We can divide Equation 2.3 into three parts. The first part ($E_{bat}(t-1) \times (1-\sigma)$) calculates the natural self-discharge, ignoring charging or discharging the battery. The second part ($P_{ch}(t-1) \times \eta_{ch} \times \Delta t$) computes the energy stored in the battery according to the charging power. The last part ($P_{dch}(t-1) \times \eta_{dch} \times \Delta t$) is similar but for discharging. Both charging and discharging are not perfect with some losses given by η_{ch} and η_{dch}. For example, if we apply 1 kW this does not mean that, after one hour, we charged 1 kWh. We will charge $1kW \times \eta_{ch}$ (where $\eta_{ch} < 1$). Besides, we can not charge and discharge the battery simultaneously, so if $P_{ch} > 0$ then $P_{dch} = 0$, and vice-versa [16]. Equation 2.4 normalizes the SoC to percentage.

Hydrogen, differently from batteries, is more suitable for long-term storage (e.g., over seasons), mainly because it can store large amounts of energy with very low self-discharge [61]. A big limitation of this kind of storage is the lack of reactivity since it demands a longer warming-up time. Furthermore, it includes performance degradation concerns, low efficiency compared to batteries, high costs, and complicated safety measures [11]. Even with all these drawbacks, it is a good solution for storing energy during abundant periods (e.g., summer) and using it during lacking periods (e.g., winter). Three elements compose an HSS: electrolyzer, hydrogen tank, and fuel cell. The electrolyzer produces hydrogen from electricity, according to Equation 2.5 [16].

$$E_{ez} = P_{ez}(t) \times \Delta t = \frac{HH_{h_2} \times Q_{ez}(t)}{\eta_{ez}} \tag{2.5}$$

Where:

- E_{ez}: Energy put into the electrolyzer ($P_{ez}(t) \times \Delta t$);
- $P_{ez}(t)$: Power put into electrolyzer (kW);
- HH_{h_2}: H_2 higher heating value (kWh/kg);
- $Q_{ez}(t)$: Electrolyzer H_2 mass flow (kg);
- η_{ez}: Electrolyzer efficiency (%).

This equation indicates how much hydrogen is added to the tank ($Q_{ez}(t)$) according to the electrolyzer operating power ($P_{ez}(t)$). On the other hand, the fuel cell transforms hydrogen into electricity, according to Equation 2.6 [16].

$$E_{fc} = P_{fc}(t) \times \Delta t = LH_{h_2} \times Q_{fc}(t) \times \eta_{fc} \tag{2.6}$$

Where:

- E_{fc}: Energy delivered by fuel cell ($P_{fc}(t) \times \Delta t$);
- $P_{fc}(t)$: Power delivered by fuel cell (kW);
- LH_{h_2}: H_2 lower heating value (kWh/kg);
- $Q_{fc}(t)$: Fuel cell H_2 mass flow (kg);
- η_{fc}: Fuel Cell efficiency (%).

Similarly, this equation indicates how much hydrogen is removed from the tank ($Q_{fc}(t)$) according to the output power of the fuel cell ($P_{fc}(t)$). To calculate the Level of Hydrogen ($LoH(t)$ (kg)) Equation 2.7, consolidates the result of the electrolyzer and the fuel cell.

$$LoH(t) = LoH(t-1) + Q_{ez}(t-1) - Q_{fc}(t-1) \tag{2.7}$$

2.3.2 IT elements

While electrical elements are power producers (wind turbines and solar panels) or producers/consumers (batteries and hydrogen), the IT elements are entirely power consumers. IT power consumption can be divided into two parts: IT hardware (e.g., servers, data storage, and network devices) and supporting infrastructure (e.g., cooling equipment) [5, 62]. This thesis focus on computing nodes (servers) and scheduling policies on the IT side, so we do not consider data storage, network devices, and supporting infrastructure. There are several articles dealing specifically with these components [62, 63, 64, 65, 66]. The servers are powerful, high-performance machines designed to handle intensive computational tasks and ensure the efficient functioning of various applications and services. They are optimized for reliability, scalability, and performance. Even with these optimizations, they do not have a negligible power consumption [63, 67].

The server power consumption is divided into two parts: static and dynamic [63, 68]. Static power consumption is constant and given by current leakage present in any powered system. Dynamic power is not constant and depends on computing usage. There are different models to estimate power consumption, such as mathematical linear and non-linear, linear regression, lasso regression, support vector machines, etc [67]. Equation 2.8 expresses a mathematical linear representation of static and dynamic power [67, 68].

$$Pcpu(t) = P^{static} + (P^{dynamic} \times u_{cpu}) \tag{2.8}$$

Where:

- $Pcpu(t)$: Power consumption at moment t (W);
- P^{static}: Static power consumption (W);
- $P^{dynamic}$: Dynamic power consumption (W);
- u_{cpu}: CPU usage (%);

While Ismail and Materwala indicate that P^{static} can be considered as the power idle [67], Heinrich et al. demonstrate a slight difference between the power usage at fully idle and when the real P^{static} [68]. Power idle (or P_{idle} in Figure 2.5) is the CPU power usage when all its cores are idle with nothing running. The processor enters a power-saving mode, reducing power consumption. On the other hand, P^{static} is the power consumption when at least one core is running something. Then, starting with P^{static}, the relation between the number of active cores and power is linear, as presented in Figure 2.5. The work of Heinrich et al. is the base for a well-known data center simulator named Simgrid[1] and its evolutions. This article also indicates that $P^{dynamic}$ depends on the application and the server frequency. Figure 2.5 shows the linearization of the power consumption according to the frequency for the same application. Setting different frequencies is possible through the Dynamic Voltage and Frequency Scaling (DVFS) technique. Putting the server at a lower frequency reduces the server's power consumption (as illustrated in Figure 2.5). However, it also decreases the server's speed. Nevertheless, DVFS is a possible solution to reducing energy consumption in moments with lower power available.

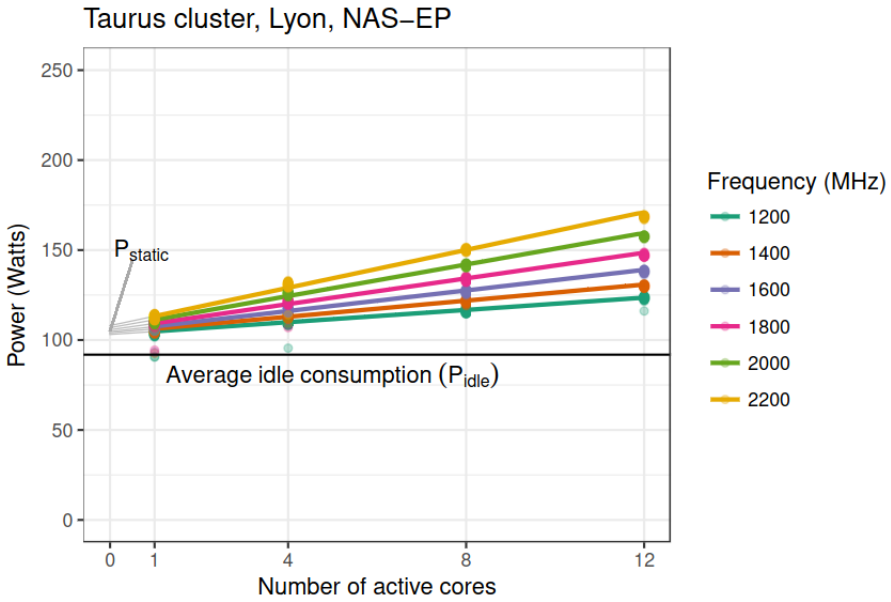

Figure 2.5: Power consumption on a GRID5000 server when running the same application, but varying the frequency and the number of active cores [68].

Another possibility, more drastic, is putting the server to sleep. In the sleep state, the server is unavailable but consuming the lowest possible. Besides being inaccessible, another consideration in the sleep state is that sleep transitions (on→off and off→on) are not instantaneous and waste energy. Raïs et al. present a Dynamic Power Management (DPM) solution [69]. This DPM estimates a T_{wait} threshold that when the server is idle for more than T_{wait} seconds, it is more energy-efficient to switch the server off. Equation 2.9 represents their idea.

$$T_{wait} = \max\left(\frac{E_{OnOff} + E_{OffOn} - (P_{Off} \times (T_{OnOff} + T_{OffOn}))}{P_{idle} - P_{Off}}, (T_{OnOff} + T_{OffOn})\right) \tag{2.9}$$

Where:

[1] https://simgrid.org/

- T_{wait}: Waiting time before putting the server to sleep (s);
- P_{idle}: Power consumption when the server is unused, but powered on (W);
- P_{Off}: Power consumption when the server is off (not null and lower than P_{idle}) (W);
- E_{OnOff}: Energy consumed during the On→Off transition (J);
- E_{OffOn}: Energy consumed during the Off→On transition (J);
- T_{OnOff}: Time spent by the server on On→Off (s);
- T_{OffOn}: Time spent by the server on Off→On (s);

A data center's main objective is to execute users' applications. Running applications in the servers variates the server's CPU usage u_{cpu}. Data centers receive plenty of different application types. We can separate these applications into two big categories: services and batch [11, 70]. Services are applications that interact with different clients. These clients make requests answered by a running service. Each request has a low processing time, but the ensemble of these requests can be very CPU-consuming [71]. In addition, the service must answer the request as soon as it arrives. On the other hand, batch applications (or parallel jobs [27]) do not run interactively. While services can run indefinitely, batches have a start and end time. Usually, these applications aim to solve complex problems, such as weather prediction, optimization problems, and simulations, being very long and CPU-consuming [71]. Batch jobs are more flexible considering the moment to execute them, allowing the batch scheduler to define the best moment in the future to run them. Both services and batches demand different approaches and algorithms to deal with them. This thesis focuses on batch high-performance computing (HPC) applications. An HPC job is composed by [11, 72, 73]:

- *Submission time*: The moment when the user sends the job;
- *Requested resources*: The resources demanded by the job, such as the number of cores, servers, memory, etc;
- *Estimated execution time or walltime*: The user indicates how long the job executes. If the real execution time is equal to the walltime, the scheduler kills the job.

One of the most important components in a data center is the scheduler. The scheduler is responsible for managing all the IT elements. The scheduler actions are:

- *Job filtering*: The scheduler must accept or reject incoming jobs from users. For example, it can reject a job because it is too big to process in this data center;
- *Waiting queue management*: All the accepted jobs wait in a queue. The scheduler uses this list to take the jobs to process. Additionally, the scheduler can sort the list by any metric, such as size, importance, arrival (e.g., First-Come, First-Served), etc.;
- *Job placement*: The scheduler selects a server to process a job from the waiting queue. The server must match the job's resource requirements (e.g., memory, processors, storage, etc.);

- *Execution management*: The scheduler can capture metrics of the execution of the jobs, such as execution time, energy usage, CPU usage, job state, etc. These metrics can help in decision-making. It can interrupt the job execution, killing, suspending (to finish later), or migrating it. In addition, it should kill the jobs with execution time higher than the walltime given by the user, avoiding a job with infinity execution using the resources;

- *Server configuration*: In a power-aware system such renewable-only data center, the scheduler must set the server's states. The server state can be sleeping or running at some speed (through DVFS). So, it can turn on servers to deal with new jobs or shutdown or change the processor's frequency to save energy;

All these responsibilities make the scheduler a crucial element of optimization in a data center. It is even more important with the introduction of renewable-only power constraints.

2.4 Sources of Uncertainty

After describing renewable-only data center elements, in this section, we detail the sources of uncertainty. First, we start presenting the uncertainty from electrical components due to weather conditions. After that, we describe the uncertainties from server power consumption and HPC jobs. Finally, we discuss the challenges in dealing with all these uncertainties.

2.4.1 Weather Uncertainties

As presented in Section 2.3.1, the objective of the electrical components (solar panels and wind turbines) is to generate power. So, they transform natural renewable resources into energy. Due to the intermittence of these renewable resources, the output power is also intermittent [74]. Regarding solar panels, the output power is calculated easily, using Equation 2.2, in a "clear-sky" condition [75]. "Clear-sky" considers an exposition total of the panels to the sun. However, solar irradiance is impacted by several weather conditions, such as clouds, aerosols, and other atmospheric constituents [75]. Besides, the panel efficiency is temperature dependent. Concerning wind turbines, the power output depends on the wind speed (see Equation 2.1). The production has lower and higher wind speed thresholds, meaning that even too slow/fast wind will not produce power.

Due to the renewable intermittence, it is crucial to forecast weather conditions to estimate future power production. Several works propose ways to predict these conditions [75, 76, 77, 78]. Two key terms are important in renewable production: Predictability and Variability [74, 78]. Predictability means the ability to anticipate the availability of a generation resource [74]. For example, solar irradiance is more predictable than wind speed because the forecast accuracy on clear days is high, and satellite data tracks precisely the direction and speed of clouds [74]. On the other hand, due to the erratic nature of the atmosphere, there is randomness in wind power production [77]. Variability indicates the variation over time in production [74]. Both wind and solar can vary. For example, the wind has high variability because it will deviate from 0%–100% over a day [74]. Another element that influences forecast accuracy is the time horizon. For example, the next five minutes are more predictable than the next three days.

2.4.2 Workload Uncertainties

Workload uncertainties come from two sources: the server's energy consumption and jobs. Estimating the real power consumption of a server is not trivial. Several works try to find a model to describe energy consumption or even apply machine learning to define it [62]. Even two machines with the same configuration can consume differently [79]. It is also true that distinct applications can have completely different energy consumption, mainly because they use the CPU differently [79]. Equation 2.8 presents a simplification of server power consumption. However, this equation is still applicable since different servers can have different dynamic ($P^{dynamic}$) and static (P^{static}) power. Considering that energy consumption is the integral of Equation 2.8, different applications can have distinct CPU usage (u_{cpu}) over time. Even if the equation is still appropriate, predicting its parameters is challenging. For example, the CPU usage (u_{cpu}) of a job can vary between executions (e.g., due to different application parameters). Furthermore, new applications do not have records to estimate their usage. Considering the static power (P^{static}), it is known that it can vary according to the processor's heat [80].

Besides impacting server consumption, jobs have their own uncertainties. A workload (ensemble of jobs) can be predicted as a load mass or resource usage (e.g., CPU usage over time) [71, 81]. These predictions help the scheduler to define how many machines are needed to cope with the demand load. However, the exact jobs' arrival is very difficult to predict and can lead to energy waste. For example, if a server is available expecting a job, but the job does not come or arrives late, this server wastes energy unnecessarily. The submission is one of the job uncertainties. The second job uncertainty is the execution time. The scheduler receives jobs with requested resources and walltime. So, the scheduler will find a placement for each job to match the requested resources during the walltime. The walltime is a user expectation of the execution time that can be overestimated [73]. An overestimated walltime reduces the effectiveness of the scheduler because it will reserve more resources than necessary for the job [72, 73].

2.4.3 Dealing with Uncertainties

After describing the uncertainties in electrical and IT elements, we present some ways to deal with them. The renewable-only data center global problem is a scheduling problem under power constraints. Therefore, the problem includes:

1. *Scheduling*: The first objective of a data center is scheduling jobs. The scheduler must choose the actions (from Section 3) to deal with the arriving jobs. The uncertainty comes from the jobs;

2. *Power*: The second objective is finding the best power decisions for the electrical part, mainly the energy storage. The scheduler can decide to use more or less energy from batteries. For example, the scheduler can use more energy from the battery to finish a job earlier. In addition, it must adapt the power consumption according to over/underproduction. Finally, it must respect the state of charge constraints, such as letting the SoC at a safe level;

3. *Server*: Finally, the scheduler must translate power consumption to server configurations. Therefore, it must decide which machine impact when there is more or less energy.

An optimization problem for a renewable-only data center must consider all these elements. We can divide the problem into offline and online. Offline optimization uses

predictions (from weather and workload) to optimize the decisions. Some methods are available to estimate power production and demand, such as Artificial Neural Networks, Support Vector Machines, Markov Chains, Regression Models, Autoregressive Models, and a combination of the methods, such as using genetic algorithms to optimize a neural network [71, 75, 77, 78, 81]. Then, this optimization finds the best approach to match production and demand (e.g., shifting the load, using more power from batteries, rejecting jobs, etc.). Finally, the offline optimization result is applied to the real scenario of production and demand. The idea is to show that even under the uncertainties, the optimized result is good enough. However, offline optimization does not react to real events. For example, it maintains the plan even in a scenario with under/overproduction. Furthermore, the power demand for the workload is treated as a mass, even if in practice a data center receives jobs. This workload simplification helps to solve the optimization problem since the scheduling problem is an NP-Complete [82, 83]. Some works propose offline scheduling, knowing all information from the jobs. However, this is unrealistic in reality [82].

On the other hand, online optimization does not know any future events (e.g., job arrival and power production), discovering them on the fly. Since online just knows actual events, it can not find the optimal global solution. So, online reacts to the incoming events optimizing the problem locally. Differently from offline, online works in real-time and can not take too long to find a solution. This real-time reaction complicates finding a near-optimal global solution. Besides, the uncertainties are reduced, since the online knows the actual events. However, future events are yet uncertain. Sometimes online optimization can introduce fast predictions, but they have a small time horizon (seconds to minutes). To sum up: offline uses predictions to optimize, but it is not reactive; online reacts to actual events, but without global optimization. Then, a third possibility emerges: A mix between offline and online. This combination allows taking the best from each side (prediction and global optimization from offline and reactivity from online).

There are several methods to optimize this problem. We can divide them into five groups: (i) exact algorithms; (ii) greedy heuristics; (iii) machine learning; (iv) metaheuristics; and (v) game theory.

The exact methods consist of creating a mathematical model of the problem. The model defines an objective function. It is possible to optimize the objective function through Linear Programming (LP). Solvers such as CPLEX[2] and Guroby[3] are used to find the optimal. The drawback of this approach is its high computation time in large problems, especially if one or more variables are integers (called Mixed Integer Linear Programming - MILP). So, it is not suitable for online optimization, but it is the best approach for offline (when the solving time is not a constraint).

A greedy heuristic is a problem-solving strategy employed in algorithm design that aims to efficiently find approximate solutions by making locally optimal choices at each step, without considering the overall global optimality. Heuristic operates by iteratively selecting the most advantageous option based on defined criteria or objective functions. Although it may not guarantee the optimal solution, the greedy heuristic's simplicity and computational efficiency make it particularly useful for tackling large-scale problems. Two examples of heuristics for job scheduling are First Come First Served (FCFS) and Easy Backfilling. Figure 2.6 demonstrates the differences between both algorithms. In FCFS, the jobs are placed in the order they arrive. The Easy Backfilling approach tries to fill the hole in scheduling with small jobs (J4 in the figure). Easy Backfilling is highly dependent

[2]https://www.ibm.com/fr-fr/analytics/cplex-optimizer
[3]https://www.gurobi.com/

on walltime estimation in this backfilling step, highlighting the impact of the uncertainties [72, 73]. Walltime is the maximum execution time for the job given by the user. The user can overestimate this walltime, difficulting the scheduling [73].

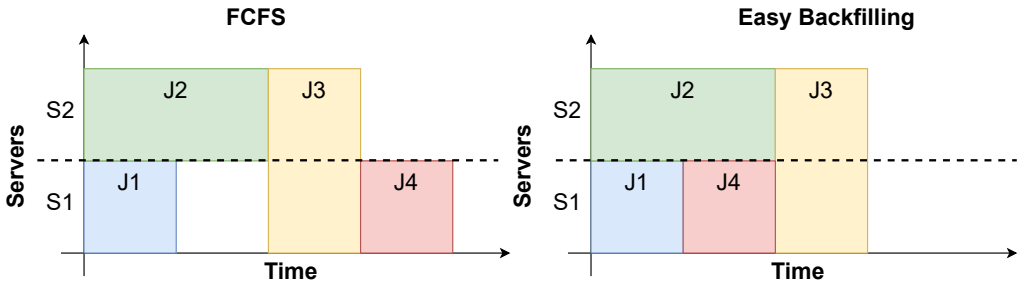

Figure 2.6: Comparison between FCFS and EASY Backfilling scheduling heuristics.

Machine learning is a subfield of artificial intelligence that contains algorithms capable of automatically learning from data and improving performance on specific tasks. In some cases, they emulated the process of human learning. For example, Artificial neural networks simulate the neural network from the human brain. Another example is Reinforcement Learning (RL), which considers the trial-and-error approach, where an agent explores an environment, takes actions, and receives feedback [84]. Three components compose an RL model: **state**, **action**, and **reward**. Let's exemplify it by using RL to define the website content. A website can apply RL to define which content to display for the user. The idea is to discover the user's subject preferences (e.g., sports, politics, technology, etc.) The **state** is the information about the user, such as age, previous subjects read, etc. Using this **state**, the RL evaluates it and chooses the articles to show to the user. The chosen articles' subjects are the **actions**. Finally, the **reward** can be 1 if the user clicked on the article and 0 if the user did not click on it. RL algorithm tries to maximize (in this case) the **reward**, increasing the number of clicks. Since RL does not know the user's behavior, it tries some articles. The clicked articles reinforce the user's subject preferences. Then, the following process is performed at each decision moment (see Figure 2.7):

1. RL receives a state from the environment;
2. RL verifies which action to take for this state;
3. RL applies the action in the environment;
4. The environment returns a reward for this action;
5. RL uses the reward to calculate the relation between state and action;
6. The environment goes to a new state, restarting the process.

The RL interacts with the environment in this way several times. RL uses the feedback (reward) to learn which are the best actions for the states. Another important aspect is the exploration-exploitation. Since the RL agent does not know the environment in the early interactions, it starts exploring the different actions in the state. After some interactions, the agent stops the exploration and starts to exploit the actions with higher rewards in the past. Different approaches can be used to model the exploration-exploitation transition, which also depends on the RL algorithm.

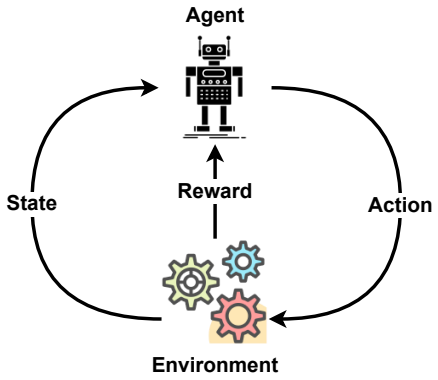

Figure 2.7: Agent learning process in an environment. At each step, the agent verifies the actual state and chooses an action. The environment executes the action and returns a reward. The agent learns the reward obtained in that state for that action.

Metaheuristics are another kind of algorithm to solve hard optimization problems. The "meta" term indicates that they are "higher level" heuristics, different from the problem-specific heuristics [85]. They are nature-inspired (based on some principles from physics, biology, or ethology). An example is the Genetic algorithm which simulates the evolution and mutation process from biology. Another example is Swarm algorithms inspired by the collective behavior of social insect colonies and other animal societies. Generally, metaheuristics are used to solve problems with no satisfactory problem-specific algorithm [85].

Finally, Myerson defines game theory as “the study of mathematical models of conflict and cooperation between intelligent rational decision-makers”. Game theory uses mathematical techniques to make decisions in situations with two or more individuals and where the decisions impact the welfare of each individual. Even having the word "game" in its name, it is not only related to recreative activities, and it could be applied to different situations [86, 87]. Therefore, like in a game, a set of actions is given to each individual, that needs to make their actions using their interests. The individual’s actions return gains or losses (depending on the model) for all "players". This kind of algorithm is known for solving conflict problems.

In this thesis, we apply exact algorithms, greedy heuristics, and machine learning directly in this thesis. A game theory can be applied between offline and online parts (the following section explains its usage). In the offline part, we applied exact algorithms to find optimal solutions. For online, we propose heuristics to solve the specific problem. We also attempted to introduce RL to learn the environment’s behavior. Since the online problem needs a fast solution for a specific problem, metaheuristics were not studied.

2.5 Datazero2 Project

The Datazero2 project aims to integrate all IT and electrical elements in a feasible architecture [24]. This integration englobes the sizing of all elements (e.g., number of serves, model of wind turbines and solar panels, battery and hydrogen capacity, etc.), the interconnections between electrical and IT devices, power and workload predictions, message format among all elements, and decision processes. Figure 2.8 presents the architecture of the project. The figure’s bottom part illustrates the IT (nodes/servers) and power (battery, hydrogen, wind turbines, and solar panels) elements. All the components (servers,

electrical elements, decision modules, etc.) communicate through a Message BUS. On the decision side, it is possible to divide the decision into two main parts: offline and online. Offline consists of IT Decision Module (ITDM), Negotiation Module (NM), and Power Decision Module (PDM). Negotiation is a crucial step in Datazero2 architecture. A renewable-only data center introduces several constraints and decision variables. On the PDM side, it must approximate demand and production while considering long-term storage elements. For example, PDM can provide more power from hydrogen in a case with low renewable generation. However, PDM must evaluate the impact of its actions since the energy of the storage is finite. On the other hand, ITDM maximizes the Quality of Service. Thus, it demands more energy to run more servers at faster speeds. Both PDM and ITDM make their decisions using predictions (weather prediction for PDM and workload prediction for ITDM).

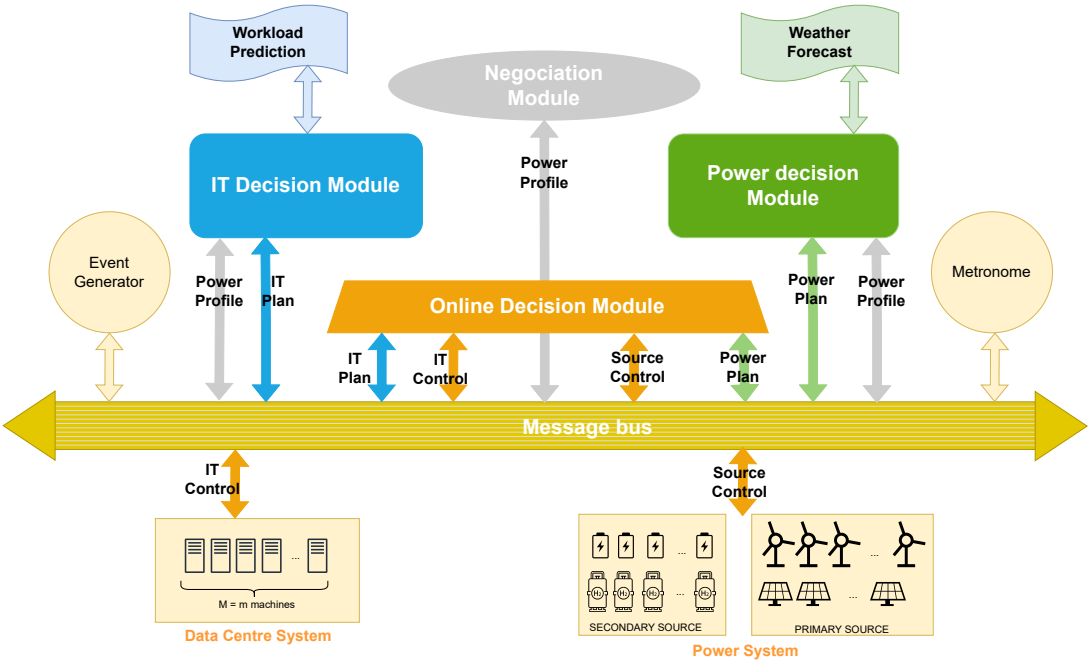

Figure 2.8: Datazero2 architecture [24].

NM is between PDM and ITDM, trying to find an agreement. NM works iteratively in several rounds. On each round, both PDM and ITDM propose a power envelope to NM, considering the objective of each module. A power envelope is a time series of the power production from the sources in a time window. While PDM tries to reduce the power envelope to control the storage, ITDM increases it to run more jobs faster. NM compares both envelope propositions and returns a new one. Then, each module verifies if they can use the proposed power envelope. They run several rounds until both agree or reach a timeout. At the end of the negotiation game, each module creates a plan for the IT and electrical elements, respecting the power profile. ITDM produces the IT Plan and PDM creates the Power Plan. This step ends the offline part.

The only decision module that works in online mode is Online Decision Module (ODM). This module is the focus of this thesis. It receives both plans from the offline modules. ODM can control all the IT and electrical elements. Besides the plans, ODM also receives through the message BUS the real events, such as power production and job arrival. Therefore, ODM applies the plan, making modifications according to the real events. For example, in a lower power production situation, ODM can reduce IT power usage to match

production and usage. The following chapters will explore the algorithms to implement ODM.

2.6 Literature Review

This section presents works to solve the issues related to a renewable-only data center. We verify the decisions on both offline and online levels. Some articles are not renewable only but introduce renewable in the decision process. We divided the articles into three groups: only offline, only online, and mixed decisions. The works are presented in chronological order inside each group.

2.6.1 Offline Decisions Only

Gu et al. [88] proposed an Integer Linear Programming for minimizing the carbon emissions of a data center while meeting the scheduling QoS (response time), and considering distributed data centers. Their LP finds the optimal solution to meet a response time constraint, the minimal number of servers, and the best moments to make the server available (e.g., when there is more renewable production). They used an M/M/n model to schedule service requests. They inserted an electricity budget as a constraint. Kassab et al., in [89], [90], and [91], defined an offline scheduling model to minimize the makespan under power constraints (renewable-only data center). The works were developed in the context of the Datazero project. They proposed heuristics and metaheuristics to find the optimal solution to the NP-Hard scheduling problem. The works [89] and [90] focus only on schedule, maintaining the server availability static (e.g., the server state is previously), while in [91], the authors take a step further, adding server decisions (machines on/off). The server decisions are to turn on servers when needed (and there is power available in the power envelope) and shut down idle servers. All three works ignore power decisions, such as using more or less power from batteries, or online uncertainty.

In [92], the authors created a weighted average scheduling algorithm (WALECC). This algorithm receives a DAG of a parallel application and defines the job placement. This article is unrelated to renewable energy. However, the problem of a system with energy constraints is similar to a renewable-only data center. WALECC mixes heterogeneous processors with DVFS decisions to find the optimal placement (considering makespan) respecting the energy constraints. WALECC works offline because it knows all the jobs in advance, having the information of each job execution time on each processor/speed. Since it receives the relationship between the jobs through a DAG, it respects the jobs' precedence. Lu et al. [93] presented a robust optimization for scheduling and power decisions aiming to minimize energy costs. Their scheduling model knows the execution time of each job at different nodes (servers). The authors introduced renewable production uncertainty in the model. Since the uncertainty introduces a random variable, they created a threshold-based algorithm to choose the solution considering the uncertainty distribution. The cost minimization is solved using Linear programming on MATLAB's MOSEK optimization toolbox.

Caux et al. [94] introduced RECO, a Genetic based algorithm. RECO aims to minimize jobs' due date violations in a renewable-only data center. In an offline way, RECO defines the optimal DVFS frequency to run batch jobs and the server states (on/off). Therefore, it works on both scheduling and server configuration. RECO applies a genetic algorithm, creating several scheduling possibilities (pair of job and server) as chromosomes. It applies crossover and mutation, selecting the best-fitted genes. They proposed

two fitness functions (to choose the genes). First, a function to reduce the number of due date violations, and the second one uses a weight-based approach, mixing due date and power consumption. The exact processor, frequency, and starting time are assigned using a greedy heuristic. [21] is an evolution of [94], where Caux et al. proposed a new heuristic named MinCCMaxE and a heuristic for degradation. The objective is to maximize the profit from the batches and services execution. Both services and batches are composed of different phases. Services run all the time window duration. On the other hand, MinCCMaxE must find the best placement for batches. MinCCMaxE cross-correlates task and processor loads (both as time series). If it does not find placement (due to power constraints), it delays or degrades the jobs. The degradation step considers the impact on the profit. Both works ([94] and [21]) were developed in the context of the Datazero project.

Haddad et al. [16] modeled a Constraint Satisfaction Problem to define power decisions in a renewable-only data center. This work considers wind and solar as renewable production and battery and hydrogen as energy storage. The objective is to find the power decisions (e.g., battery charge, hydrogen discharge, etc.) that approximate the power produced and demanded. They defined target levels of battery and hydrogen hard constraints. So, the decision variables are battery and hydrogen usage and a relax factor applied to the relationship between produced and demanded. The model considers all losses in power generation, such as battery discharge rate, battery charge/discharge efficiency, hydrogen charge/discharge efficiency, etc. This work is also in the context of the Datazero project. In [95], the authors proposed a system for matching renewable production and demand. They predicted renewable generation and energy demand using Long Short-Term Memory (LSTM). Renewable production comes from wind turbines and solar panels, using brown energy as a backup. The model uses the result of the predictions to map renewable sources to physical machines. They solved the problem using integer linear programming and Deep Q-Network. Deep Q-Network is an extension of the Reinforcement Learning algorithm Q-Learning.

Wiesner et al. [15] created Cucumber, an admission control policy. The authors use probabilistic forecasts to predict power demand (from workload) and production (from renewable sources). Using both predictions, they calculate how much energy is free to use (named freep). The freep is a time series of renewable production minus power demand. Since probabilistic forecasts results in several predictions, they introduced a parameter to tune the forecasts' optimism. Then, they evaluate the freep time series to accept or reject new jobs in an FCFS fashion. They verify if placing a new job using freep capacity would violate the deadline from the other jobs. If so, they reject the new job. The idea is to maximize the peak of renewable production without adding batteries. Yuan et al. [66] proposed an optimization problem to minimize the operating cost of the entire data center. The data center is powered by renewable energy coming from wind turbines and solar panels, energy storage, and the grid. Their model considers the possibility of selling and buying energy to the grid. For solar and wind turbines, the authors take into account the operation and maintenance costs. They modeled the IT part considering network equipment and cooling system. Similar to other works, they consider batches and services, considering their delay characteristics.

2.6.2 Online Decisions Only

Aksanli et al. [96] presented an online heuristic that uses short-term (30 min) forecasts for green energy in the scheduling process. The green energy comes from wind and solar. The scheduler has two queues, one for services and another for batch. Services will run independently of the green energy available, using brown if needed. Then, they predict the

green energy available in the next period. Using this prediction, they estimate the number of slots available to run batch jobs. If the number is greater than the currently available slots, they schedule new jobs and spread the remainder to the running ones. If this number is smaller, the scheduler deallocates some jobs (reducing slots, killing, suspending them, or using brown energy). The authors compared their predictive heuristic with a reactive scheduler that allocates servers according to the energy available. Their solution achieves 3 times better green energy usage and reduces the number of canceled tasks up to 7.7 times.

The authors in [22] presented Blink, a renewable-only manager which fastly changes the state (on and off) to meet a power constraint. For example, if the power supply drops by 10%, blink deactivates the servers 10% of the time. They propose this manager completely off-grid. However, the applications must have the ability to stop and resume their execution instantaneously, which is not the case in the majority of the applications. They focused mainly on *memcached* applications, a general-purpose distributed memory-caching system. These applications manage a distributed database, answering other client applications. *memcached* applications have very low execution time since they only have to answer the requests with the demanded data. Applying this policy in applications with higher execution times can lead to execution interruption. They focused in maximize the percentage of answered requests (hit rates).

In [18], the authors proposed a parallel job scheduler, named GreenSlot, for data centers powered by solar energy but using the grid as backup. Their scheduler uses predictions of solar generation to place jobs in the moments with higher production. If it is impossible to place all the jobs in these moments, GreenSlot finds the moments where the grid energy price is lower. GreenSlot saves energy by deactivating idle servers. It creates several slots with the cost. GreenSlot calculates the cost assuming zero for green and the grid price for brown. It assigns penalty costs on slots that cause the job's deadline, avoiding them. The authors indicated some limitations in their work, such as high job rejection or missed deadlines in data centers with high utilization.

Li et al. [97] created a framework named GreenWorks to manage power and server decisions in a data center powered by renewable energy and using battery and grid as backup. They defined a heuristic to manage power generation in four stages. Stage I is when renewable generation is enough to ensure full-speed server operation. In this stage, the renewable production excess is stored in the batteries. Stage II is active when the production is inadequate to provide the power demanded. Here, the heuristic balance between discharging the battery and impacting the jobs (through DVFS). If this stage is not enough to handle the power mismatch, the system enters stage III. In this stage, GreenWorks tries to decrease load power more, use UPS energy if it has power, and, the last resource, it use the grid. They only use power from the grid for the same amount of energy that they exported previously. So, they accumulate a budget of net green energy exported. If it is not enough, stage IV shuts down the servers. GreenWorks does not use predictions and relies on the first-come-first-serve (FCFS) scheduler.

The authors in [98] created an opportunistic scheduling heuristic. This heuristic tries to minimize brown energy usage by mixing batteries and solar production. The heuristic takes into account services and batches. When the energy consumption is higher than the solar supply, they suspend batch jobs and consolidate the VMS, switching off the servers. The scheduler takes energy from the battery before going to the grid. Just after the batteries dry, it starts to consume brown energy. They implemented a First Fit Decreasing (FFD) scheduling algorithm. Grange et al. [99] proposed an algorithm named Attractiveness-Based Blind Scheduling Heuristic (ABBSH). ABBSH introduces a

negotiation model for electrical and IT systems, where both know only their own model. Negotiation helps to deal with different objectives. For example, the objective of the electrical system is to reduce brown energy usage, while IT is to respect the System Level Agreement (SLA) criteria. Both calculate a normalized metric named attractiveness. This metric represents the quality of a given proposal. So, for each job, it calculates the attractiveness of its placement in the IT and electrical context. Then, a function defines, among all possible placements, which one has the best attractiveness. The authors select the best attractiveness using a simple weighted sum of both (electrical and IT), a weighted sum of the hyperbolic sinus of both, or a fuzzy-based one. In this work, the authors consider the electrical part as a black box without making power decisions.

Haghshenas et al. [100] developed a heuristic aiming to minimize energy costs. The energy cost (from the grid) is the IT and cooling usage minus solar generation. This heuristic considers services and batches. It always schedules the services in the FIFO approach. It finds the best moment to place batches using best-fit and considering solar production and energy price. Even online, the authors assume knowing all the jobs for the next period. They used solar production predictions to make decisions for the next time slot. They consider the possibility of selling solar production to the grid. They do not consider power on/off transitions, assuming the IT energy consumption is zero when the servers are not running jobs. Finally, the authors updated their algorithm, adding a simplified battery model. The battery will store the surplus solar generation to use later, reducing the energy cost. [101] proposed an online heuristic to schedule jobs in servers. The main objectives are to minimize the makespan, energy cost, and overall cost and maximize renewable usage. The authors separated the heuristic into three phases. First, they estimated the completion time and cost for the execution of a user request on each data center. The cost considers if the data center is powered by renewable or non-renewable. The second phase calculates the fitness value, normalizing completion time and cost. Finally, the last phase takes the data center with higher fitness to place the user request. The work considers that all the data centers are available all the time (using renewable or not).

In [20], He et al. created an online scheduling heuristic to minimize the energy cost of a data center called ODGWS (Online workload Scheduling algorithm with Delay Guarantee). The authors take into account solar, wind, and grid energy. They simplified the job description to be the power requested at each time step by services and batches. The scheduler must deliver the services' power requested in the same step that they arrive. On the other hand, the scheduler can delay the batches' power demand until a fixed time horizon. So, even if the authors named their algorithm workload scheduling, the decisions are which source to use to provide the energy to the jobs. They do not consider the placement problem (e.g., which server will receive the jobs). Besides, they assume that the servers are available all the time. Their problem is a constrained stochastic problem solved by dynamic programming, but they translated it into an online heuristic, which can work without knowing future events.

Peng et al. designed REDUX3 an energy management system with a renewable-aware scheduler. REDUX3 uses energy from different sources, such as wind turbines, solar panels, the grid, diesel generators, and batteries. The system focuses on batch tasks, allowing them to postpone jobs to match production and demand. They added the grid to deal with uncertainties. They created three energy states: Outage Case, Stable Case, and Fluctuate Case. An Outage Case is when the renewable supply is at the minimum, a Stable Case is when the renewable supply exceeds the maximum level, and a Fluctuate Case is when the previous energy state was Outage Case but is stable now. In addition, they

introduced three key components. First, the scheduling window module defines the number of available processors according to the energy case. Second, the scheduling algorithm uses a backfilling algorithm to place the jobs in the available processors. This scheduler receives a job priority, considering it in the decision process. Finally, they do a job power profiling, providing data to create a job power model.

In [19], the authors defined an online energy-aware scheduling algorithm using deep reinforcement learning. The main idea is to use the grid power when the carbon emission rate is lower. They used a DAG to define the different tasks of a batch job. The DAG indicates when a task can start (e.g., task 2 can run after task 1 is finished). A typical reinforcement learning algorithm has three key elements: state, actions, and reward. The state in this article is given by server usage, task queue, electricity price, and emission rate. The task queue contains only the tasks ready to run, considering the job's DAG. The actions are the tuple of task and server, considering the feasibility of this tuple (if a server can not receive the task, it is not a feasible action). Finally, the reward is carbon, cost, and QoS. They do not introduce server decisions, so the servers are always available.

2.6.3 Mixed decisions

The only work that mixed offline and online decisions is [23]. In the offline part, the authors predict renewable energy production. They use these predictions, energy storage, and brown energy to fix the number of resources. After that, online makes the scheduling decisions considering the offline server configuration. They created a deep reinforcement learning algorithm to define the jobs to run. The state is composed of tuples containing both the resource availability and the array of job metadata. The job metadata includes the price that the user is willing to pay, QoS, expected finish time, duration, and resource requirements. The actions are which job run, suspend a job, or do nothing. So, they can suspend a job, placing a job with a higher price first. This suspended job will run later. Finally, the reward is the total value obtained by running the jobs respecting the QoS. The authors indicate that power decisions would transform their problem into a multi-criteria optimization problem. These power decisions include changing battery usage and selecting power sources. They claimed that this multi-criteria optimization is future work. They evaluated the impact of the power supply intermittence on the algorithm, but DRL does not consider it in the model.

2.6.4 Discussion and Classification of the Literature

Table 2.1 presents all the works presented in the previous section. We classified them considering 5 criteria:

- *Objective*: It describes what each article aims to optimize;
- *Electrical infrastructure*: It synthesizes the power sources, such as solar panels, wind turbines, batteries, hydrogen, diesel generator, and the grid;
- *Offline decision*: The type of decisions in offline way. We classify these decisions into scheduling, server, and power decisions. Scheduling decisions focus on placing the jobs, deciding the order, rejecting jobs, selecting the servers, etc. Server decisions are the decisions about the availability of the servers, such as server state (on/off) or speed (DVFS). power decisions define power usage (e.g., charging/discharging batteries);
- *Online decisions*: Same as offline decisions, but in online mode;

- *Method*: The method used to solve their problem (e.g., Heuristic, Exact algorithm, etc);

It is possible to notice that several works introduce renewable energy in data centers environment. As mentioned before, renewable sources provide clean but intermittent power. The majority of works add a connection to the grid, aiming to reduce the impact of the intermittence. However, these approaches maintain the brown energy dependency. Some of the works with the grid have as their objective to minimize energy costs, adding the possibility of selling energy to the grid. This could lead to a solution where selling energy is more attractive than running jobs due to price fluctuations. As presented in this chapter, migrating entirely to renewable sources is mandatory. Several works without grid connections are from the Datazero project. They are pertinent works and are the base of this thesis. The only outside Datazero no-grid works are from Wiesner et al. and Sharma et al.. The first one focuses on using the peak of renewable sources to schedule new jobs. In our work, and other works from the literature [16, 23, 66, 97, 98, 100, 101, 102], these peaks are stored in energy storage. These energy storages (e.g., batteries or hydrogen) help to smooth the RES intermittence but introduce another decision level. The work from Sharma et al. deals with web service applications by blinking (deactivating/activating) the servers. This behavior demands small applications (low execution time) or applications with the possibility of fastly stopping and resuming their execution.

Usually, the works with energy storage only apply it as an energy cache. So, they can dry or overflow these resources, without good management. Maintaining the state of charge of batteries between a narrow range can increase the battery's life. Only Li et al. consider good battery decisions to maximize the battery life span. However, the authors balance using the battery or buying energy from the grid. Another important aspect of battery management is related to mixing short, and long-term decisions. In short-term decisions (online), it could dry the battery to maximize the QoS. However, the battery is finite and this behavior could lead to future QoS degradation (e.g., drying the battery in the short-term would reduce battery usage in the future, decreasing the power available to IT servers).

Table 2.1 also highlights that the majority of the previous works only focus on one side of the decision process. Sometimes they make offline decisions considering knowing all the future events or using predictions. These works focus on finding an optimal solution for the problem. However, the uncertainties of a renewable-only data center demand a reaction when the solution is not feasible in reality. Another possibility is working online entirely, reacting to events or using fast predictions. In this case, we lose the long-term aspects. For example, it is simple to maximize the number of jobs running in a renewable-only data center if the system dries the energy storage. However, in the big picture, this behavior leads to problems in the following days. [23] is the only work aggregating decisions on two levels, but with a simple offline server configuration. They create this configuration using power from the grid, and they do not change battery usage to react to actual production.

This thesis aims to provide algorithms to integrate both offline and online decisions. While the offline creates a plan considering long-term, online reacts and adapts this plan. To the best of our knowledge, there are still no approaches integrating both sides to make better decisions. Furthermore, we introduce three levels of decisions, considering server states, job scheduling, and power decisions. Mixing all these aspects allows us to follow the long-term plan while improving QoS. Finally, we focus on HPC batch jobs. The services appear in some works as a constraint, since they do not make big decisions about them (e.g., some authors use brown to meet service QoS).

Table 2.1: Summary of characteristics for existing renewable data center scheduling works.

Article	Year	Objective	Electrical infrastructure	Offline decisions	Online decisions	Method
Aksanli et al. [96]	2011	Maximize green energy usage	Solar panels, wind turbines, and grid	-	Server and Scheduling	Heuristic
Sharma et al. [22]	2011	Minimize performance degradation	Solar panels, wind turbines, and batteries	-	Server	Heuristic
Goiri et al. [18]	2015	Maximize green energy usage and reduce grid energy cost	Solar panels and grid	-	Server and Scheduling	Heuristic
Gu et al. [88]	2015	Minimize carbon emissions	Solar panels, wind turbines, and grid	Server, Scheduling, and Power	-	Exact algorithm
Li et al. [97]	2016	Balance QoS, battery life span, and average backup time	Wind turbines, batteries, and grid	-	Server and Power	Heuristic
Kassab et al. [89]	2017	Minimize makespan and flowtime	Solar panels and wind turbines	Scheduling	-	Heuristic
Li et al. [98]	2017	Maximize green energy usage	Solar, batteries, and grid	-	Server, Scheduling, and Power	Heuristic
Kassab et al. [90]	2018	Minimize makespan and flowtime	Solar panels and wind turbines	Scheduling	-	Metaheuristic
Grange et al. [99]	2018	Minimize grid energy and respect QoS	Solar and grid	-	Server and Scheduling	Heuristic

Article	Year	Objective	Electrical infrastructure	Offline decisions	Online decisions	Method
Hu et al. [92]	2018	Minimize makespan under energy constraints	-	Scheduling	-	Heuristic
Lu et al. [93]	2018	Minimize energy cost	Solar panels and grid	Scheduling and Power	-	Exact algorithm
Caux et al. [94]	2018	Maximize QoS under power constraints	Solar panels and wind turbines	Server and Scheduling	-	Metaheuristic and heuristic
Caux et al. [21]	2019	Maximize profit under power constraints	Solar panels and wind turbines	Server and Scheduling	-	Heuristic
Haddad et al. [16]	2019	Match power demand and production	Solar panels, wind turbines, batteries, and hydrogen	Power	-	Exact algorithm
Gao et al. [95]	2020	Match power demand and production minimizing QoS violations	Solar panels, wind turbines, and grid	Power	-	Exact algorithm and machine learning
Haghshenas et al. [100]	2020	Minimize energy cost	Solar panels, batteries, diesel generator, and grid	-	Scheduling and Power	Heuristic
Nayak et al. [101]	2021	Minimize makespan, energy consumption, and overall cost, and maximize renewable usage	Not specified (renewable and non-renewable without battery)	-	Scheduling	Heuristic

Article	Year	Objective	Electrical infrastructure	Offline decisions	Online decisions	Method
He et al. [20]	2021	Minimize energy cost	Solar panels, wind turbines, and grid	-	Power	Heuristic
Peng et al. [102]	2022	Minimize energy cost	Solar panels, wind turbines, batteries, diesel generator, and grid	-	Server, Scheduling, and Power	Heuristic
Wiesner et al. [15]	2022	Maximize renewable excess energy usage	Solar panels and wind turbines	Scheduling	-	Heuristic
Yuan et al. [66]	2022	Minimize energy cost	Solar panels, wind turbines, batteries, and grid	Power	-	Exact algorithm
Liu et al. [19]	2023	Minimize the energy consumption cost and carbon footprint	Grid	-	Scheduling	Machine learning
Venkataswamy et al. [23]	2023	Maximize job value (revenue)	Solar panels, wind turbines, batteries, and grid	Server	Scheduling	Machine learning

2.7 Conclusion

This chapter focused on presenting the motivations and fundamentals for understanding this work. We started explaining the context, going from global warming to the role of renewable-only data centers on climate change. Then, we detailed the several elements that compose a renewable-only data center. Finally, we described the different sources of uncertainty, presenting some classes of algorithms to deal with them. In the second part of this chapter, we showed different works on applying renewable sources in data centers. We clarify the different approaches, highlighting the gaps in the state-of-the-art.

Chapter 3

Modelling, Data, and Simulation

Contents

3.1 Introduction

After describing the state-of-the-art, this chapter presents the models, data, and simulation tools used in this thesis. First, we focus on the model describing the offline and online scheduling problem. We explain what kind of information is exchanged between offline and online. After the model, we introduce the traces used in the experiments. These traces emulate a real environment regarding workload, weather, and platform. Finally, we detail the simulation tools, explaining the modifications needed to create the experimental environment.

3.2 Model

As presented in Chapter 2, a gap in the state-of-the-art is the mix of offline and online. Figure 2.8 illustrates the architecture proposed by the Datazero2 project, mixing both decision levels [24]. Since this work is part of this project, we use the same architecture. There are four main modules: IT Decision Module (ITDM), Power Decision Module (PDM), Negotiation Module (NM), and Online Decision Module (ODM). ITDM, PDM, and NM are responsible for the offline decisions, and ODM manages the online actions. This thesis focuses on the Online Decision Module. However, we present in the following sections the optimizations made in offline modules to provide the data needed by ODM since we propose a mix between offline and online decisions.

Besides these four modules, Datazero2 also includes an event generator and a metronome. Both components are essential for the simulations. Event generator simulates the real events of a data center, such as job submissions, weather conditions, etc. It simply reads a file and sends the data to the bus. The metronome synchronizes the simulation messages. So, every component waits for the time evolution from the metronome. This thesis does not detail these components, concentrating on the decision modules and their interactions.

Table 3.1: General notations.

Notation	Description
t	Time step (int)
T	Last time step (int)
Δt	Time step length (s)
T_w	Time window length (s)
P_{load}	Estimated power demand (kW)
u_{load}	Uncertainty of P_{load} (See Section 3.2.2) (float)
P_{prod}	Power production by all sources (kW)
P_{renew}	Power delivered by renewable sources (kW)
u_{renew}	Uncertainty of P_{renew} (See Section 3.2.2) (float)

Table 3.1 presents the general notations and each following section introduces its own notations. Both offline and online use the time division from Figure 3.1. The time window is the horizon of the offline plan. Offline considers the time window to define how far to predict weather and workload. In addition, it uses the time window to determine the planned actions. Our model divides the time window into several time steps, as represented in Figure 3.1 by the different t. The actions for power and server are constant inside the time step. For example, if a server is at some state in step $t = 0$, it will remain at this state during the step duration.

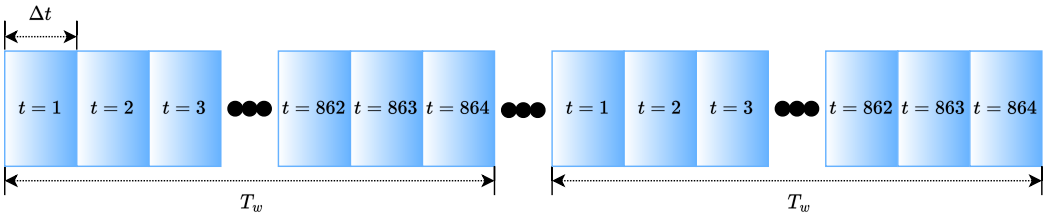

Figure 3.1: Time window definition. It gives an example for a time window of 3 days.

3.2.1 Offline Decision Modules

This section starts presenting the offline decision modules. First, we demonstrate how PDM and ITDM agree on a power envelope through NM. Then, we explain the power decisions from PDM, resulting in a power plan. Finally, we detail the ITDM, which defines the IT plan.

Negotiation (NM)

The negotiation Module is between PDM and ITDM, trying to find an agreement about the power profile. Since this thesis focuses on the online part, we simplify this process. We implemented the negotiation in three steps. First, ITDM proposes a power envelope P_{load} based on the energy demanded to run a predicted workload. P_{load} is a prediction from ITDM. We do not include it in this model, but we present in the following chapters how we generated it. Then, PDM takes this envelope and runs its optimization. It can degrade the power envelope to meet its objectives, resulting in a new power envelope P_{prod}. Finally, ITDM takes the PDM power production and finds the best server configuration that meets it. The following sections present the PDM and ITDM optimizations.

Power Decision Module (PDM)

Table 3.2 gives the notations for PDM. PDM plans the renewable source engagement to provide the energy needed to maintain the IT elements running. A renewable-only data center introduces several constraints in power generation. Therefore, PDM must approximate the demanded power while considering long-term storage elements. For example, it can use more energy coming from hydrogen during the winter, which has lower power production, compensating for this usage in the summer, which has higher power generation. On the other hand, PDM can degrade the provided energy due to a lack of energy from storage and estimated renewable. In the context of Datazero, Haddad et al. created the first model to solve this problem [16]. This thesis uses a similar model to PDM. Equation 3.1 gives the power production from all renewable sources. Equation 3.2 indicates that P_{renew} comes from wind and solar production. $P_{wt}(t)$, $P_{pv}(t)$, $P_{fc}(t)$, $P_{ez}(t)$, $P_{dch}(t)$, and $P_{ch}(t)$ are calculated using Equations 2.1, 2.2, 2.6, 2.5 and 2.3 from Section 2.3.1. Batteries are reversible, which means that they can charge ($P_{ch}(t)$) and discharge ($P_{dch}(t)$) energy. Similarly, hydrogen can be considered reversible using an electrolyzer to charge ($P_{ez}(t)$) and a fuel cell to discharge ($P_{fc}(t)$).

Table 3.2: Notations for PDM.

Notation	Description
P_{wt}	Power delivered by wind turbines (kW)
P_{pv}	Power delivered by solar panels (kW)
P_{fc}	Power delivered by fuel cell (kW)
P_{ez}	Power put into electrolyzer to generate hydrogen (kW)
P_{dch}	Battery discharging power (kW)
P_{ch}	Battery charging power (kW)
η_{ch}	Battery charge efficiency (%)
η_{dch}	Battery discharge efficiency (%)
SoC	State of Charge (%)
LoH	Level of Hydrogen (kg)
SoC_{max}	Maximal battery State of Charge (%)
SoC_{min}	Minimal battery State of Charge (%)
C_{bat}	Capacity of the battery (kWh)
LoH_{max}	H2 tank limit (kg)
$P_{dch_{max}}$	Battery maximum discharging power (kW)
$P_{ch_{max}}$	Battery maximum charging power (kW)
$P_{fc_{max}}$	Fuel cell maximum charging power (kW)
$P_{ez_{min}}$	Electrolyzer minimum charging power (kW)
$P_{ez_{max}}$	Electrolyzer maximum charging power (kW)
SoC_{target}	Target State of Charge at the end of the time window (%)
LoH_{target}	Target Level of Hydrogen at the end of the time window (kg)
rf	Relax factor. The values is between 0 and 1 (float)
P_{load}^{real}	Real power demand (kW)
P_{renew}^{real}	Real power production (kW)

$$P_{prod}(t) = P_{renew}(t) + (P_{fc}(t) + P_{dch}(t) - P_{ez}(t) - P_{ch}(t)), \quad \forall 0 \leq t \leq T \tag{3.1}$$

$$P_{renew}(t) = P_{wt}(t) + P_{pv}(t), \quad \forall 0 \leq t \leq T \tag{3.2}$$

$P_{ch}(t)$, $P_{dch}(t)$, $P_{ez}(t)$, and $P_{fc}(t)$ are the decision variables in Equation 3.1, since $P_{wt}(t)$ and $P_{pv}(t)$ come from wind speed and solar irradiance. As presented in Section 2.3.1, $SoC(t)$ depends on the charge $P_{ch}(t)$ and discharge $P_{dch}(t)$ (see Equation 2.3), and $LoH(t)$ depends on the power of the electrolyzer $P_{ez}(t)$ and fuel cells $P_{fc}(t)$ (see Equation 2.7). Regarding $SoC(t)$, the state of charge must be between the boundaries SoC_{min} and SoC_{max}, as written in Equation 3.3. These boundaries help to extend the battery lifespan [25].

$$SoC_{min} \leq SoC(t) \leq SoC_{max}, \quad \forall 0 \leq t \leq T \tag{3.3}$$

On the other hand, hydrogen only has the tank size as a boundary. So, Equation 3.4 presents the level of hydrogen constraint.

$$0 \leq LoH(t) \leq LoH_{max}, \quad \forall 0 \leq t \leq T \tag{3.4}$$

Considering the power to charge/discharge the batteries, both have upper limits. These boundaries avoid destroying the battery. So, we introduce constraints 3.5 and 3.6.

$$0 \leq P_{dch}(t) \leq P_{dch_{max}}, \quad \forall 0 \leq t \leq T \tag{3.5}$$

$$0 \leq P_{ch}(t) \leq P_{ch_{max}}, \quad \forall 0 \leq t \leq T \tag{3.6}$$

Fuel cells and electrolyzers also have boundaries. While fuel cells have only a maximum limit, electrolyzers have an operating range. So, Equations 3.7 and 3.8 present them.

$$0 \leq P_{fc}(t) \leq P_{fc_{max}}, \quad \forall 0 \leq t \leq T \tag{3.7}$$

$$P_{ez_{min}} \leq P_{ez}(t) \leq P_{ez_{max}}, \quad \forall 0 \leq t \leq T \tag{3.8}$$

Another important constraint is the target hydrogen and battery level at the end of the time window ($SoC(T)$). Using only the previous constraints, the model can use all the power available in the energy storages, drying them but providing a high quality of service. However, Figure 3.1 shows that the time windows are chained. So, the next time window will not have energy in the storage. Therefore, we introduce these targets. So, the state of charge and level of hydrogen in the last step of the time window must respect Equations 3.9 and 3.10. These targets can be the subject of another optimization or indicated by hand by the data center manager. Furthermore, the targets must consider the long-term perspective, such as seasons with lower/higher production, the peak of demand over an external event, etc.

$$SoC(T) \geq SoC_{target} \tag{3.9}$$

$$LoH(T) \geq LoH_{target} \tag{3.10}$$

Finally, the objective is to approximate the power production to the power demand. So, Equation 3.11 shows the relation between demand (P_{load}) and generation (P_{prod}). The optimization finds a solution where the production is higher or equal to the demand. However, it can not match both in every case. Therefore, the model introduces a demand degradation using a relax factor (rf). With the relax factor equal to 0, it matches demand and production. Increasing the relax factor would reduce the power given to IT, impacting the QoS. Thus, the objective is reducing the relax factor, as presented in Equation 3.12.

$$P_{prod}(t) \geq (1 - rf) \times P_{load}, \quad \forall 0 \leq t \leq T \tag{3.11}$$

$$\textbf{minimize } rf \tag{3.12}$$

IT Decision Module (ITDM)

ITDM aims to maximize QoS, translating $P_{prod}(t)$ into server configuration. Table 3.3 introduces the notations for ITDM. Server configuration means the CPU P-state of the servers. For each P-state, the server has a speed (in flops [103]) and power consumption (in W). Table 3.4 exemplifies this relation. The CPU frequency range is discrete, although some works define it to be continuous [104]. ITDM must find the best combination of servers off and on at some speed that uses equal or less energy than the power envelope $P_{prod}(t)$. Thus, given a data center with S servers, each server s has a list of states D_s. Each state d in D_s has a speed $F_{s,d}$ and a power $P_{s,d}$. $D_{s,d}(t)$ is the boolean decision variable that indicates that the server s is at state d at step t. The sleep state has a different state $Dsl_s(t)$ which helps to identify the transition between sleeping and running. The transition between on→off is called sedating and off→on is waking.

Table 3.3: Notations for ITDM.

Notation	Description
S	Servers (list)
N_S	Number of servers (int)
s	Server index (int)
D_s	States of server s (list)
N_{D_s}	Number of states of server s (int)
d	State index (int)
$F_{s,d}$	Speed of server s at state d (Flops)
$P_{s,d}$	Power of server s at state d (W)
$D_{s,d}$	Indicates that the server s is at state d (boolean)
Dsl_s	Indicates that the server s is sleeping (boolean)
wa_s	Indicates if the server is waking, transiting from off→on (boolean)
T_{wa_s}	Transition time from off→on (s)
se_s	Indicates if the server is being sedate, transiting from on→off (boolean)
T_{se_s}	Transition time from on→off (s)
E_{tot}	Energy total spent by the servers (J)
E_{run}	Energy spent by the running servers (J)
E_{wak}	Energy spent by waking the servers (J)
E_{sed}	Energy spent by sedating the servers (J)
E_{sle}	Energy spent by the sleeping servers (J)

First, Equation 3.13 ensures only one state per time t. The state can be anyone from $D_{s,d}$ to indicate a P-state, or $Dsl_s(t)$ to specify the sleep state. Since both variables are booleans (accepting only 0 or 1 values), summing them must be equal to 1 (at least one must be true). Both $D_{s,d}$ and $Dsl_s(t)$ are the only decision variables in the ITDM model.

$$Dsl_s(t) + \sum_{d=0}^{N_{D_s}} D_{s,d}(t) = 1, \quad \forall 0 \leq t \leq T, \forall 0 \leq s \leq N_S \tag{3.13}$$

Table 3.4: Server definition example. The power is for all server's processors busy. The values are from Grid5000's Parasilo server [105, 106].

State	Power (W)	Speed (Gflops)
0	221.77	38.4
1	216.77	37.78
2	213.58	36.93
3	208.90	36.01
4	204.45	34.72
5	200.62	33.90
6	197.28	32.84
7	192.49	31.72
8	184.26	30.63
9	182.04	29.25
10	179.75	27.93
11	176.70	26.37
12	175.53	25.01
13 (sleep)	4.5	0

Then, we must model the sedating and waking transitions. These transitions take time and spend energy. During these transitions, the servers are unavailable to run jobs. So, Equations 3.14 and 3.15 model the waking transition (off→on) and the sedating transition (on→off), respectively. For example, Equation 3.14 verifies if the previous state is sleeping ($Dsl_s(t-1) = 1$) and now is not sleeping ($Dsl_s(t) = 0$). So the result of $Dsl_s(t-1) - Dsl_s(t)$ will be 1, which indicates that the server s is waking. If both are 0 or 1, the result is 0, implying that the server is not transiting. If the previous state is not sleeping ($Dsl_s(t-1) = 0$) and now is sleeping ($Dsl_s(t) = 1$), the result will be -1. However, the max(0) function will put 0 as $wa_s(t)$. Equation 3.15 does the same but inverts the order of states.

$$wa_s(t) = \max(0, (Dsl_s(t-1) - Dsl_s(t))), \quad \forall 0 \leq t \leq T, \forall 0 \leq s \leq N_S \tag{3.14}$$

$$se_s(t) = \max(0, (Dsl_s(t) - Dsl_s(t-1))), \quad \forall 0 \leq t \leq T, \forall 0 \leq s \leq N_S \tag{3.15}$$

Equation 3.16 introduces the power constraint. We transform the power into energy (multiplying $P_{prod}(t)$ by Δt) because we are dealing with the transitions. Thus, the power from a server is not constant inside a time step (e.g., in the waking transition, first a server spent energy turning on and just after running jobs). Since $E_{tot}(t)$ is in J and the energy generated ($P_{prod}(t) \times \Delta t$) is in kJ, we transformed the generation into J by multiplying by 1000. $E_{tot}(t)$ is the total energy spent by the servers calculated using Equation 3.17. This equation sums the expended energy by running, waking, sedating, and sleeping states.

$$P_{prod}(t) \times \Delta t \times 1000 \geq E_{tot}(t), \quad \forall 0 \leq t \leq T \tag{3.16}$$

$$E_{tot}(t) = E_{run}(t) + E_{wak}(t) + E_{sed}(t) + E_{sle}(t), \quad \forall 0 \leq t \leq T \tag{3.17}$$

Equations 3.18, 3.19, 3.21, and 3.20 demonstrate the energy of each state. Equation 3.18 verifies if the server is in one of the possible P-states $D_{s,d}$. If so, it multiplies the

power by the time in this state. For calculating the time in the state, the equation verifies if the server is waking, removing the transition time if so. Equation 3.19 does the same for the sleeping state, considering the sedating transition. Equations 3.20 and 3.21 are simpler, just multiplying the power usage in the transition state by the time, if the server is in the state.

$$E_{run}(t) = \sum_{s=0}^{N_S} \sum_{d=0}^{N_{D_s}} D_{s,d}(t) \times P_{s,d} \times (\Delta t - (wa_s(t) \times T_{wa_s})), \quad \forall 0 \leq t \leq T \tag{3.18}$$

$$E_{sle}(t) = \sum_{s=0}^{N_S} Dsl_s(t) \times P_{sl_s} \times (\Delta t - (se_s(t) \times T_{se_s})), \quad \forall 0 \leq t \leq T \tag{3.19}$$

$$E_{wak}(t) = \sum_{s=0}^{N_S} wa_s(t) \times T_{wa_s} \times P_{wa_s}, \quad \forall 0 \leq t \leq T \tag{3.20}$$

$$E_{sed}(t) = \sum_{s=0}^{N_S} se_s(t) \times T_{se_s} \times P_{se_s}, \quad \forall 0 \leq t \leq T \tag{3.21}$$

Finally, Equation 3.22 demonstrates the objective function of ITDM. The objective is to maximize the total flops executed by the server. We do not consider idle servers here, since we do not make offline scheduling. So, when the server is running, we consider that it is providing all the flops possible at state $D_{s,d}$. Like in the energy consumption in the running state (Equation 3.18), the objective function also considers the transition state for reducing the total flops delivered by the server.

$$\textbf{maximize} \sum_{t=0}^{T} \sum_{s=0}^{N_S} \sum_{d=0}^{N_{D_s}} D_{s,d}(t) \times F_{s,d} \times (\Delta t - (wa_s(t) \times T_{wa_s})) \tag{3.22}$$

3.2.2 Offline Plan

The PDM and ITDM optimizations result in a plan for the next time window, providing two time series: The power plan (PDM) and the IT plan (ITDM), as follows:

- Power plan (PDM)
  - Time step (t);
  - For battery (for every t):
    - Power usage ($P_{dch}(t)$ and $P_{ch}(t)$);
    - Expected storage level ($SoC(t)$);
  - For hydrogen (for every t):
    - Power usage ($P_{fc}(t)$ and $P_{ez}(t)$);
    - Expected storage level ($LoH(t)$);
  - For solar panels and wind turbines (for every t):
    - Estimated renewable power production ($P_{renew}(t)$);
    - Power production uncertainty ($u_{renew}(t)$);
- IT plan (ITDM)

- Time step (t);
- For each server (for every t):
  - P-state ($D_{s,d}(t)$ and $Dsl_s(t)$);
- Estimated power demand ($P_{load}(t)$) (for every t);
- Power demand uncertainty ($u_{load}(t)$) (for every t);

u_{renew} and u_{load} indicate confidence in the prediction. They give the range that the actual values can be. For example, let's say $P_{load}(t) = 500$ and $u_{load}(t) = 200$. So, the real value of $P_{load}(t)$ ($P^{real}_{load}(t)$) is between 300 and 700 ($P^{real}_{load}(t) = P_{load}(t) \pm u_{load}(t)$). Equations 3.23 and 3.24 present the possible actual values interval.

$$P_{renew}(t) \times -u_{renew}(t) \leq P^{real}_{renew}(t) \leq P_{renew}((t)) \times u_{renew}(t), \quad \forall 0 \leq t \leq T \tag{3.23}$$

$$P_{load}(t) \times -u_{load}(t) \leq P^{real}_{load}(t) \leq P_{load}((t)) \times u_{load}(t), \quad \forall 0 \leq t \leq T \tag{3.24}$$

So, PDM and ITDM send this plan to ODM, which uses it as a guide for real-time decisions. However, the following sections present the reasons for changing this plan.

3.2.3 Online Decision Module (ODM)

ODM is a power-aware scheduler. A power-aware scheduler means having all responsibilities described in Section 2.3.2 while managing electrical elements. In this section, we present the job scheduler and the modifications in the offline plan. Table 3.5 introduces the notations for ODM.

Table 3.5: Notations for online scheduling and adaptations.

Notation	Description
Q	Waiting queue of jobs (list)
j	Job index (int)
$Wall_j$	Walltime of job j (s)
$Wait_j$	Waiting time of job j (s)
Sb_j	Submission time of job j (s)
Ex_j	Execution time of job j. It is how many seconds the job is executing from the beginning to now (s)
Dfl_j	Demanded flops of job j. This is unknown in advance (flop)
Dfl'_j	Estimated demanded flops of job j. The scheduler estimates it using Equation 3.29 (flop)
R_j	Number of servers requested by job j (int)
$Size_j$	Estimated job j size. It is the walltime multiplied by the number of servers requested (float)
$bsld_j$	Bounded Slowdown of job j (float)
Gfl_j	Flops processed by the job j (flops)
F'_s	Server speed for job size estimation (flops)
ϵ_u	User execution time estimation error (float)
ΔE_{bat}	Difference of target and calculated battery energy at the end of the time window (kWh)
E_{comp}	Energy to compensate (kWh)

Job scheduler

Since the offline in the model does not know the jobs exactly, it can not execute the job scheduling. Therefore, ODM must define the scheduling. The job scheduler's objective is to maximize the number of finished jobs, considering the power constraints. It does not know exactly when the jobs will arrive. As soon as a job arrives, the scheduler places it in a waiting queue Q. Each job j in the waiting queue Q is composed by submission time (Sb_j), walltime ($Wall_j$), and the number of resources requested (R_j). Walltime is the maximum execution time allowed for the job. The job executes flops (Dfl_j), but it is discovered only after the job execution.

After placing the job in the waiting queue, the scheduler must select which job will start. For selecting the next job to place, the scheduler must sort the queue Q. This sort puts the more important jobs in front, according to a specified rule. One rule can consider the jobs' size, using Equation 3.25. Since the scheduler does not know the real size, it estimates using the walltime and number of resources requested.

$$Size_j = Wall_j \times R_j \tag{3.25}$$

Another way to sort the waiting queue Q is using Bounded Slowdown. Equation 3.26 demonstrates how to calculate the Bounded Slowdown. It estimates the ratio between the total time a job stays in the system and its actual processing time. This order helps to let a job wait proportionately to its size. τ is a constant to avoid smaller jobs from reaching a very high Bounded Slowdown. Since the scheduler does not know the actual size, Equation 3.26 uses the walltime as size.

$$bsld_j = \max(\frac{Wait_j + Wall_j}{\max(Wall_j, \tau)}, 1) \tag{3.26}$$

After sorting the queue, it places the front job in R_j servers. Due to a possible heterogeneous data center, the scheduler takes first the servers with higher speed. After starting, $D_{s,d}$ controls how fast the job will finish. The scheduler considers a job as a mass (unknown) flops Dfl_j. So, increasing the speed $D_{s,d}$ increases the number of flops processed by the server and reduces the execution time Ex_j. On the other hand, reducing the speed reduces the server's flops and increases the execution time. Constraint 3.27 verifies if the execution time is lower than the walltime $Wall_j$. The moment when the execution time becomes equal to or greater than the walltime, the scheduler kills the job. The job finishes correctly when it executes all its flops Dfl_j before the walltime $Wall_j$.

$$Ex_j < Wall_j \tag{3.27}$$

Modifying Offline Plan

An important part of ODM is adapting the offline plan. The offline plan gives a guide in a predicted scenario, but the reality can be different. The main reason for adaptations is due to power production and demand variations. There are three sources of variation: IT consumption, renewable production, and scheduling adaptations.

The IT consumption is the difference between the ITDM planned usage and the real usage. Since ITDM does not know the real scheduling, it considers the worst-case where the server usage is the higher possible. Equation 2.8 demonstrates the relation of the power consumption at an instant according to the CPU. Inside a time step t, the server's CPU usage can vary according to the jobs. Figure 3.2 illustrates this difference due to the

job variance. Since we consider the worst-case in ITDM, the real energy usage is always lower or equal to the predicted.

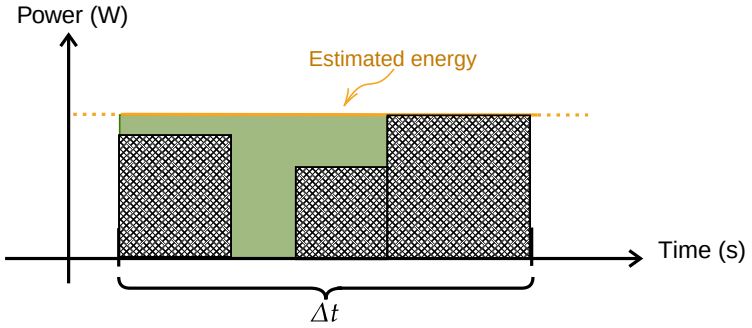

Figure 3.2: Energy consumption comparison between predicted and real. The black boxes are jobs. The green area is the energy predicted, but not used.

The renewable production variation comes from the wind and solar variations. These uncertainties can increase or decrease power generation P_{renew}. Finally, the scheduler adapts the server configuration to improve the QoS. Since the objective of the scheduling is to maximize the finished jobs, it must modify the offline plan. For example, if a job is running on a server, but the IT offline plan indicates putting this server to sleep, it would kill the job. So, ODM can change this decision, using more energy now and maintaining the job running. Another case is considering the speed given to the job. Letting a job in a server with a slower P-state can increase the execution time (Ex_j), violating Constraint 3.27 and killing the job. A way to reduce this possibility is by using Equation 3.28. This equation verifies the minimum speed to complete the remaining job's flops.

$$(Wall_j - Ex_j) \times D_{s,d} \times F_{s,d} \geq Dfl'_j - Gfl_j \tag{3.28}$$

Since ODM does not know the job's flops exactly, it can estimate using the walltime. Equation 3.29 shows a way for estimating Dfl'_j, similar to [73]. F'_s is a fixed speed applied to the job. ϵ_u is a user error because the user can overestimate the execution time. We do not consider the walltime underestimation, because this always leads to killing the job (respecting Equation 3.27). Takizawa and Takano calculate ϵ_u for each user, using previous users' requests. Even if Equation 3.28 does not exactly use the real size, it is a good way to balance the states of a job.

$$Dfl'_j = wall_j \times F'_s \times \epsilon_u \tag{3.29}$$

The batteries smooth these variations, providing the power needed in case of under-production or absorbing the generation excess. At the end of time step t, the $SoC(t)$ can be different from the prediction. Therefore, ODM recalculates all future SoC using Equations 2.3 and 2.4. With the $SoC(T)$ updated, Equation 3.30 can estimate how far the SoC at the end of the time window will be from the target. This equation gives a value of energy (in kWh). A positive ΔE_{bat} means that the battery will have more energy than predicted, allowing the scheduler to use this excess to run more jobs or speed up the servers. On the other hand, a negative ΔE_{bat} means that the battery will have less energy than predicted. In this case, ODM must reduce future usage to approximate $SoC(T)$ to SoC_{target}.

$$\Delta E_{bat} = \frac{SoC(T) - SoC_{target}}{100} \times C_{bat} \tag{3.30}$$

Then, ODM calculates how much energy it can compensate, using Equation 3.31. This equation considers the loss in the process of charge/discharge.

$$E_{comp} = \begin{cases} \frac{\Delta E_{bat}}{\eta_{ch}} & \Delta E_{bat} > 0 \\ \frac{\Delta E_{bat}}{\eta_{dch}} & \Delta E_{bat} < 0 \end{cases} \tag{3.31}$$

Finally, ODM must modify futures P_{dch} and P_{ch} to use E_{comp}. We focus on battery modifications since it is faster to make online decisions on it. Hydrogen is not too reactive. We proposed some ways to deal with it in the following chapters. Since ODM modifies P_{dch} and P_{ch}, P_{prod} will also change (see Equation 3.1). So, ODM must adapt the IT plan (servers' states $D_{s,d}$) to meet the new P_{prod}. The algorithm to modify $D_{s,d}$ on-the-fly is also presented in the following chapters. This modification must respect the Constraint 3.16.

3.3 Data

After describing the models, we describe the data used to simulate our environment. Our simulators expect three data: workload, weather, and platform. We explain the source of each one in the following sections.

3.3.1 Workload Trace

Workload trace is a log of job submissions in a resource (servers) provider. Some trace examples are Microsoft Azure [107], Google [26], and Alibaba [28]. Regarding HPC, Feitelson et al. proposed the SWF format, which allows the data center providers to distribute logs to the research community [27]. In SWF, each line is a job with the fields separated by whitespace. Each line contains the following fields [27]:

- *Job Number*: The job ID starting with 1;
- *Submit Time*: The submission time. The first job has 0 as submit time, and the following jobs use the first job as reference (in seconds);
- *Wait Time*: How long the job waited in the queue (in seconds);
- *Run Time*: The execution time in the data center (in seconds);
- *Number of Allocated Processors*: The number of processors allocated to the job (integer);
- *Average CPU Time Used*: The time that the job used the CPU. It is the average from all processors (integer);
- *Used Memory*: The used memory, also average from all processors (kilobytes);
- *Requested Number of Processors*: The number of processors requested by the user (integer);
- *Requested Time*: This is the walltime (seconds);
- *Requested Memory*: Requested memory per processor (kilobytes);

- *Status*: 1 if the job was completed, 0 if it failed, and 5 if canceled;
- *User ID*: The user ID. Can be used to identify different jobs from the same user (integer);
- *Group ID*: A group ID (some systems control the group and not the user ID) (integer);
- *Executable (Application) Number*: Can link different jobs from the same application (integer);
- *Queue Number*: Indicates in which queue the job was allocated (integer);
- *Partition Number*: Indicates in which partition the job was allocated. For example, it is possible to use partition numbers to identify which machine in a cluster was used (integer);
- *Preceding Job Number*: Indicates the number of a previous job in the workload, such that the current job can only start after the termination of this preceding job (integer);
- *Think Time from Preceding Job*: The time between this job and the Preceding Job (seconds);

It is not mandatory to insert all information in a SWF file. Currently, Parallel Workloads Archive[1] has 40 traces in SWF format. We chose the MetaCentrum2 workload trace from the Czech National Grid Organization [108]. MetaCentrum is a grid with resources in several cities in the Czech Republic. They have 19 clusters with 495 nodes and 8412 cores in total. Nodes are individual machines or servers within a cluster. Each node can have its own set of resources such as CPU, memory, storage, and network connectivity. A core refers to a processing unit within a CPU (Central Processing Unit). Modern CPUs often have multiple cores, allowing them to perform multiple tasks simultaneously. The trace has 5,731,100 jobs from January 2013 to April 2015. Metacentrum does not provide the *Average CPU Time Used*, *Used memory*, *Status*, *Group ID*, *Executable (Application) Number*, *Preceding Job Number*, and *Think Time from Preceding Job* fields. It has different queues according to the job type. The Partition number indicates which cluster executed the job. We do not have more information about the servers, just the number of nodes, cores, and memory.

Figure 3.3 illustrates the inter-arrival and execution time distribution for MetaCentrum2. Inter-arrival is the time between job submissions, and execution time is the real runtime. Applying the Kolmogorov-Smirnov test to these data, we observed that both follow a log-normal distribution with p-value = 0.2174 for inter-arrival and p-value = 0.1802 for execution time (excluding 0.003838705% of jobs as outliers). In the Kolmogorov-Smirnov test, if the p-value is smaller than the chosen significance level (we chose a confidence level of 95%, so smaller than 0.05), it indicates that there is strong evidence to suggest that the data doesn't follow the specified distribution (both values are higher than the significance level).

Another aspect of the MetaCentrum2 workload is the *Requested Time* field. This field could be considered as walltime. Nevertheless, MetaCentrum2 defines this field according to the job queue. Consequently, it does not come from the user. We recalculate the walltime using the proposition from Takizawa and Takano [73]. Thus, we equally divided the jobs into five groups. Each group estimates the walltime as follows:

[1] https://www.cs.huji.ac.il/labs/parallel/workload/index.html

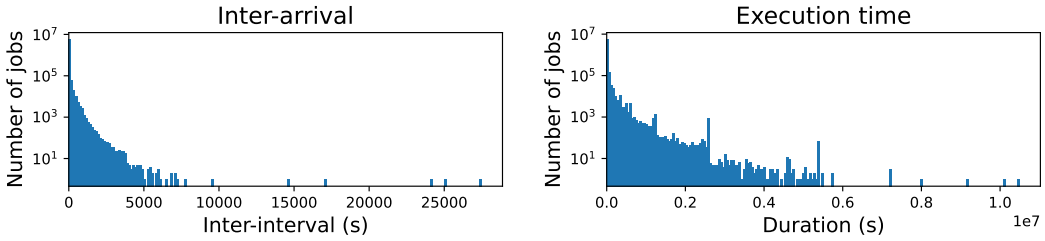

Figure 3.3: Inter arrival and execution time distribution for MetaCentrum2 workload trace.

1. walltime = real execution time × 5
2. walltime = real execution time × 3.333333333
3. walltime = real execution time × 2
4. walltime = real execution time × 1.428571429
5. walltime = real execution time × 1.111111111

This uncertainty complicates the scheduling decisions but it is more realistic. We created two scripts for translating the SWF format to the simulator chosen (see Section 3.4.1) and Datazero2 Middleware formats. We simulated several possibilities of workload, and each chapter will describe the selection process.

3.3.2 Weather Trace

Regarding weather, we are interested in the traces of solar irradiance and wind speed, which allow us to estimate power generation using Equations 2.2 and 2.1. NASA's Modern-Era Retrospective Analysis for Research and Applications (MERRA) provides a dataset of solar irradiance and wind speed from any place in the world [109]. Regarding wind speed, we get the data directly from the MERRA site[2]. Renewable Ninja[3] is a tool that transforms the weather from MERRA to electricity production [30, 110]. Renewable Ninja uses some solar panel models to estimate power generation. We use Renewable Ninja to obtain ground-level solar irradiance because the authors consider cloud cover and aerosols' impact on the irradiance. We chose Toulouse, France as the reference point, getting the data from 2020. After exporting the data, we also translate the output CSV to the simulator chosen (see Section 3.4.1) and Datazero2 Middleware formats. We took different days from 2020 for our simulations. The following chapters will present the selection process.

3.3.3 Platform Configuration

Finally, the last data is the hardware specification. This configuration simulates the real behavior of the different components of a data center, such as servers, storage, network, etc. In this thesis, we focused on the server specification. For simulating a DVFS-enabled server, the following information is necessary:

[2]https://power.larc.nasa.gov/data-access-viewer/
[3]https://www.renewables.ninja/

- Sleeping power: Power used to maintain a server sleeping;
- Time for on→off transition: Time to turn off a server;
- Power for on→off transition: Power to turn off a server;
- Time for off→on transition: Time to turn on a server;
- Power for off→on transition: Power to turn on a server;
- Power idle: Power used when the server is idle;
- DVFS states: A list of states containing:
  - Power: The power used in the state with all processors busy;
  - Speed: The speed in the state.

We chose to model a data center using the specification from GRID5000 servers. da Costa performed experiments in GRID5000 to obtain the data for the DVFS states [105, 106]. Some other authors also simulated GRID5000 machines, giving information about their simulation [21, 69, 94, 111]. We consolidate these works to create a platform for both the simulator chosen (see Section 3.4.1) and Datazero2 Middleware. Table 3.6 exemplifies the consolidated data for GRID5000's Gros server used in the experiments.

Table 3.6: Gros definition. The power is for all server's processors busy. The values are from Grid5000's Gros server.

Parameter	Value
Sleeping power	4.5 W
Time on→off	6 s
Power on→off	76.5 W
Time off→on	164 s
Power off→on	110.52 W
Power idle	62 W

State	Power (W)	Speed per core (Gflops)
0	143.45	35.2
1	123.57	33.59
2	122.34	31.98
3	121.68	30.36
4	118.49	28.79
5	115.8	27.14
6	114.58	25.57
7	110.89	23.85
8	108.06	22.38
9	106.81	20.78
10	104.13	19.18
11	102.83	17.59
12	100.78	15.99

3.4 Simulation

After describing the source of data, we detail our simulation environment. We explain two simulators and the work done to adapt our data for these simulators. We define the different metrics used to evaluate the algorithms in the following chapters.

3.4.1 Batsim Simulator

Batsim is an infrastructure simulator that enables the study of resource management policies [112]. This simulator is based on the well-known Simgrid [113]. The Batsim protocol helps to develop scheduling and resource management algorithms without Simgrid knowledge. The protocol works via a socket and includes the following events:

- Bidirectional (Batsim→Scheduler and Scheduler→Batsim):
  - *QUERY*: It has two queries. First, in the *consumed_energy* query, the scheduler asks Batsim about the total consumed energy (from time 0 to now). On the other hand, in the *estimate_waiting_time*, Batsim demands the scheduler what would be the waiting time of a potential job;
  - *ANSWER*: The *QUERY* answer;
  - *NOTIFY*: This event allows a peer to notify something to its counterpart. This message can be a specific message from an external event.
- Batsim→Scheduler:
  - *JOB_SUBMITTED*: A new job was submitted;
  - *JOB_COMPLETED*: The job has completed its execution;
  - *JOB_KILLED*: The job was killed;
  - *RESOURCE_STATE_CHANGED*: The state of a resource (server) changed;
- Scheduler→Batsim:
  - *EXECUTE_JOB*: Execute the job in the servers;
  - *KILL_JOB*: Kill the running job;
  - *SET_RESOURCE_STATE*: Set the state of a resource (server);

These are some messages of the protocol (the entire protocol is in Batsim documentation[4]). The messages *QUERY* and *ANSWER* enable the scheduler to receive IT energy consumption. The scheduler uses the *JOB_SUBMITTED*, *JOB_COMPLETED*, *JOB_KILLED*, *EXECUTE_JOB*, and *KILL_JOB* messages to control the jobs. Finally, the *SET_RESOURCE_STATE* and *RESOURCE_STATE_CHANGED* allow server configuration. However, Batsim does not provide the electrical management of renewable sources and energy storage. Considering renewable sources, the main objective is receiving power production. Therefore, we applied Equations 2.1 and 2.2 on the data from Section 3.3.2, resulting in a time series of power production. Then, we create a file with several JSON lines:

```
1  {"type": "user_specific_type", "timestamp": 0, "power_available": 0.0}
2  {"type": "user_specific_type", "timestamp": 300, "power_available": 460.4}
3  {"type": "user_specific_type", "timestamp": 600, "power_available": 5172.26}
```

Batsim reads this file and sends at each "timestamp" a *NOTIFY* message. Then, we parse the JSON in the message, taking the power available. Δt defines the interval between messages. The scheduler does not know all the events in this JSON file, receiving them just when the simulation arrives at the "timestamp". Regarding energy storage, we implemented two classes to simulate them. The first class is for the battery and uses Equations 2.3 and 2.4 to estimate the state of charge. The second class is for hydrogen

[4]https://batsim.readthedocs.io/en/latest/protocol.html

and implements Equations 2.5, 2.6, and 2.7. The energy storage implementation must respect the following constraint:

$$((P_{renew} + P_{dch} + P_{fc} - P_{ez} - P_{ch}) \times \Delta t \times 1000) - E_{tot} = 0 \tag{3.32}$$

Constraint 3.32 indicates that the difference between energy production ($(P_{renew} + P_{dch} + P_{fc} - P_{ez} - P_{ch}) \times \Delta t \times 1000$) and the energy expended by the IT (E_{tot}) must be equal to zero. P_{renew} comes from the JSON input with no modifications. E_{tot} is calculated using *QUERY* and *ANSWER* messages. Since it returns the total energy expended (from 0 to now), we calculate the difference between two *QUERY* calls. So, the simulator balances P_{dch}, P_{fc}, P_{ez}, and P_{ch}. In the default implementation, the simulator first adapts the battery (P_{dch} and P_{ch}), and, just if necessary, it changes the hydrogen (P_{fc} and P_{ez}). However, it depends on the implementation (e.g., if the offline plan indicates something different). This simulation respects the boundaries 3.5, 3.6, 3.7, and 3.8.

Another input for Batsim is the platform file. Batsim uses the Simgrid platform format[5]. Using the results from Section 3.3.3, we created a script to generate this file with all DVFS states. This script receives the server parameters (e.g., Table 3.6) and the number of nodes from this server. It can create homogeneous and heterogeneous data centers according to the user specification. Finally, the last input for Batsim is the workload file. Batsim simplifies the workload definition of HPC jobs. We focused on the Homogeneous Parallel Task kind of application in the experiments. In this application type, we define the amount of floating-point operations (flops) to execute on each machine. Here, we create another script to translate the SWF format (from Section 3.3.1) to Batsim JSON format. An example of a workload file is:

```
{
    "jobs": [
        {
            "id": "0",
            "profile": "profile_1",
            "res": 1,
            "subtime": 773,
            "walltime": 9112
        }
    ],
    "profiles": {
        "profile_1": {
            "cpu": 163137000000000,
            "type": "parallel_homogeneous"
        }
    },
}
```

In this example, we have only one job with the id equal to 0. This job is submitted after 773 seconds from the simulation beginning (*"subtime": 773*), with a walltime of 9112 seconds (*"walltime": 9112*), and requests one machine (*"res": 1*). The scheduler only knows these fields. The profile field indicates the number of flops to execute. Therefore, the job must calculate 163137000000000 flops. The real execution time is *cpu* divided by the speed of the server which runs this job. The script estimates the field *cpu* by the execution time in SWF multiplied by a server's average speed. We defined the average speed considering the servers in our data center. We took a median DVFS state's speed, avoiding over/underestimating. This estimation is necessary because SWF does not provide any metric of the real computation. All experiments in this thesis use Batsim as the simulator. In parallel, we implement our algorithms in Datazero2 Middleware.

[5]https://simgrid.org/doc/latest/Platform.html

3.4.2 Datazero2 Middleware

The Datazero2 project proposes a middleware to integrate all components from Figure 2.8. This middleware has all electrical and IT components, external event integrations, and decision modules. It works on two coding languages: Python and C++. It depends on different frameworks, such as:

- ActiveMQ 5.14.4;
- C++:
  - Protobuf 3.21.8;
  - Apache APR 1.7.4;
  - ActiveMQ C++ library 3.9.5;
  - Simgrid 3.27;
- Python:
  - Stomp 8.1
  - Protobuf 3.20.3
  - Pandas
  - Pulp

Some modules work in Python, and some in C++. The messages from the modules are exchanged via ActiveMQ, using Protobuf as protocol. Simgrid is used to simulate the IT side. The electrical side is implemented using the same rules presented in this thesis. This middleware is still in development by different actors in the project. During this thesis, we worked on the ODM implementation. The results from the Batsim simulations are being implemented in the middleware.

However, we helped with all these dependencies management, allowing every project's partner to execute the middleware on their computers. Therefore, we create a docker version of the middleware. We create three containers: ActiveMQ, C++, and Python. The ActiveMQ provides the message BUS for all the modules. C++ implements the Simgrid and some modules (e.g., ODM). Python has the electrical control and other modules (e.g., ITDM and PDM). We create a docker-compose script that installs and compiles everything. We included some advanced parameters for developers for coding inside docker without the need to rebuild everything.

Regarding the middleware's simulation input, some changes were done. The weather data is passed directly to the middleware, which applies the equations to transform weather into power. Even if it uses Simgrid (so, same platform file format), the middleware divides this file into two files: Model and Machines. The Model file defines the DVFS states, including a model id. The Machine file indicates all the servers, linking each one to a model id. This division simplifies the modifications in the model. So, we create a new version of the platform script, allowing choose between Batsim or Datazero2 middleware. Finally, the workload file is not JSON but XML. We also create a new version of our script to migrate from SWF to Datazero2 middleware XML.

3.4.3 Metrics

In this section, we present the metrics used in this thesis. We evaluate four aspects in this thesis: Jobs finished, storage state at the end of the time window, wasted energy,

and bounded slowdown. The first objective is to increase the number of finished jobs and reduce the number of killed jobs. It is possible to have a high number of finished jobs and a high number of killed jobs in aggressive scheduling, where the scheduler starts jobs even if it is not possible to finish them. Each job can finish in one of five states:

1. Finished: Jobs that finished their computation before the walltime;
2. Postponed: Jobs postponed to the next time window;
3. Reached walltime: The jobs that reached the walltime because they do not finish all the computation due to the servers speed;
4. Not completely finished: The jobs that were not finished completely because they are still running at the end of the time window;
5. Killed: The killed jobs.

Therefore, we present the absolute number of jobs in each state at the end of the simulation. In addition, we introduce this metric considering the size of the job ($Dfl_j \times R_j$). The size of the job shows the impact of the decisions in bigger and small jobs. The second aspect is the algorithms must end the time window with the storage levels as close as possible to the planned. This means the algorithms can not "cheat" by using more storage than planned. As explained before, finishing close to the target levels helps to plan the next time window. We present a metric of the distance from the target. This distance can be positive (save more energy) or negative (use more energy). For battery, we present it as the % difference of target SoC, and for hydrogen, we present it as the kg difference of target LoH.

The third metric is wasted energy. We consider wasted energy the energy expended not computing finished jobs, englobing, for example, the energy used in killed jobs, turning on/off servers, and letting servers idle. This metric is present as kWh. Finally, the fourth metric is the bounded slowdown. This metric is the same as presented in Equation 3.26. The bounded slowdown is a difficult metric to compare in executions without the same number of jobs finished. For example, starting and killing a job in the sequence will result in a very low slowdown. On the other hand, maintaining jobs running will make the jobs in the queue wait for more time, which impacts directly on the slowdown. So, we will analyze this metric with precaution.

3.5 Conclusion

This chapter focused on presenting the model, data, and simulation utilized in the remainder of this thesis. The aim is to provide the basis to understand the following chapters. The model introduces all aspects of offline decisions. This offline module builds a plan for the online module. The online model considers the offline as a guide but improves this plan according to the real events. Moreover, the online model considers job scheduling in the decision process. Then, we describe the work done in the data for the simulations. This work includes workload, weather, and platform information. Finally, we explained simulation tools. We detailed the modifications in Batsim to use renewable energy and energy storage. Moreover, we present the metrics used in this thesis to compare the different approaches.

The next chapter introduces the first heuristic to deal with power compensations, trying to improve QoS. After that, Chapter 5 proposes a model for learning the best

power compensations. Finally, Chapter 6 presents our final heuristic, using predictions to make better decisions.

Chapter 4

Introducing Power Compensations

Contents

4.1 Introduction

In this chapter, we propose some heuristics to solve the problem from the model presented in Section 3.2.3. Due to the uncertainties, online scheduling needs to adapt the offline plan. To address this, we propose two contributions in this chapter. First, we schedule the jobs according to the power envelope. The second contribution is to adapt the power envelope, using power compensations. Power compensations are modifications in the power envelope to approximate the battery SoC at the end of the time window to the offline plan. First, we describe the heuristics to solve it, linking with the model from Chapter 3. This heuristic includes scheduling and power decisions. Then, we explain the experimental environment used in the experiments. Finally, we present and discuss the results, highlighting the impact of the power constraints.

4.2 Proposed Approach

We divided the heuristic into two parts. First, we detail the scheduling decisions. The scheduler defines the job priority and placement. The main objective of the scheduling is to finish the jobs, avoiding killing them. To do so, the scheduler can demand more power, adapting the power envelope. Then, we describe the power compensation heuristics. Power compensations are modifications in the power envelope (given by the offline side). These modifications are needed because the real power production and demand can vary from the offline, due to the uncertainties. The power compensations' objective is to approximate the battery's state of charge from the target level at the end of the time window. Finally, the scheduler must translate the power modifications into server configurations. We explain our heuristic to do it.

4.2.1 Scheduling

Job scheduling is a well-known problem NP-complete problem [82, 83]. Several heuristics are proposed to solve this problem in an acceptable time. We implemented and adapted a well-known algorithm named EASY Backfilling in this chapter for the job placement [114]. This heuristic is known for its simple and robust implementation [72]. Furthermore, it maximizes server utilization [72]. EASY backfilling focuses on job placement, but the scheduler also adapts the power envelope to avoid killing jobs. The placement heuristic runs on every job arrival, job finished, and new time step. On the other hand, the power envelope adaptation runs at each new time step only. First, we present the placement algorithm. This algorithm englobes queue sorting and placement. First, it places priority jobs in the available servers. Then, it fills the scheduling holes with small jobs (see Figure 2.6). Algorithm 1 denotes a pseudo-code of EASY Backfilling, presented by Lelong et al. as EASY-P_R-P_B policy [115].

Algorithm 1: EASY-P_R-P_B scheduling [115].

```
   input : Queue Q of waiting jobs, P_R as priority order, and P_B as backfilling order.
   output: None (calls to Start())
 1 begin
 2     Sort Q according to P_R;
 3     for job j in Q do
 4         Pop j from Q;
 5         S_j ← select_servers(j);
 6         if j can be started in servers S_j then
 7             Start(j, S_j);
 8         else
 9             Reserve j at the earliest time possible according to the walltime of the currently
                running jobs;
10             Sort Q according to P_B;
11             for job j' in Q do
12                 S_j ← select_servers(j');
13                 if j' can be started in servers S_j without delaying the reservation on j then
14                     Start(j', S_j);
15                 end
16             end
17             break;
18         end
19     end
20 end
```

First, this heuristic sorts the jobs in the queue in a priority order P_R (line 2). Then, it selects the servers to run this job (line 5). If the job j can start in the servers S_j (line 6), it places the job in the servers (line 7). When it finds a job that can not start now, the algorithm starts to backfill (lines 9-17). Then, it reserves the first moment to run this job (named priority job) in the future (line 9). So, it re-sorts the queue using P_B (line 10), placing the other jobs in the servers (lines 11-16) without delaying the (future) priority job execution (line 13). As presented, the algorithm sorts the queue using P_R (in the priority moment) and P_B (in the backfilling moment). However, the typical implementation uses the same sort for both [115]. Our first implementation uses the typical algorithm (same sorting policy). Thus, P_R and P_B apply the Descending Bounded Slowdown (higher Bounded Slowdown first) policy from Equation 3.26. This order helps to let a job wait proportionately to its size.

Besides the placement, the scheduler makes power envelope adaptations to maintain

the jobs running at least at a "minimal speed". These decisions occur at each new time step. This minimal speed is given by Equation 3.28. The idea is to keep the jobs' servers at the speed $D_{s,d}$ (it is the DVFS speed). For example, if $D_{s,d}$ is too slow, the job can reach the walltime without finishing all its computing. Besides, if a server goes to sleep, its running jobs are killed. So, every time step t, the scheduler verifies the energy needed to maintain the jobs running at least at minimal speed. If it is necessary to use more energy now (changing the plan), it verifies if it is possible to migrate power from the future to now. This verification consists of two tests:

1. *Will battery boundaries be violated?* Equation 3.3 must be respected. So, migration can not violate the SoC boundaries;

2. *Is it possible to compensate for this change?* Section 4.2.2 explained this verification;

If possible, the scheduler modifies the offline IT plan to let the servers run at P-state $D_{s,d}$. If not, the scheduler does not change the offline plan.

4.2.2 Power compensations

After describing the scheduling algorithm, this section explains the heuristic to compensate for power fluctuations presented in Section 3.2.3. The power compensations' objective is to ensure that SoC and LoH are close to the offline plan at the end of the time window. Therefore, every time step t, ODM calculates E_{comp} using Equation 3.31. As explained in Chapter 3, we can have two kinds of compensations: positive and negative. So, E_{comp} can be positive or negative, according to battery usage or renewable production. A positive compensation indicates that the scheduler can use more energy to run jobs. On the other hand, negative compensation demands a reduction in usage to finish the time window close to the battery target level. Thus, the problem is to define the future time step to use energy E_{comp}. Let t' be the future time step. Then, ODM must change $P_{dch}(t')$ and $P_{ch}(t')$. Of course, P_{dch} and P_{ch} are power, and E_{comp} is energy, so we adapt to work only with energy in this process. Furthermore, the changes must consider the boundaries from Equations 3.5 and 3.6. In addition, the power usage in the sleep state is not zero, demanding minimal power production. Let P_{min} be the minimal power possible. Then, $P_{prod}(t') \geq P_{min}$, considering that P_{prod} comes from Equation 3.1.

ODM spreads the energy over different t' until all modifications compensate for E_{comp}. The power compensations happen in two cases:

Case 1 Every time step, ODM recalculates $SoC(T)$ using Equation 2.3 from the actual step until T. Then, it compares $SoC(T)$ and SoC_{target}, using Equation 3.30. Finally, ODM compensates E_{comp} (from Equation 3.31), approximating $SoC(T)$ to SoC_{target} (respecting Equation 3.9);

Case 2 When the scheduler demands more power, as presented in Section 4.2.1, it tries to maintain the servers with jobs running. So, it increases the power usage at the step. This increase will change the $SoC(T)$, demanding compensation. This compensation is always negative (if ODM uses more power now, it reduces the usage in the future).

If ODM can not entirely compensate E_{comp}, it has two possible actions. If the request for compensation comes from the scheduler (case 2), it does not make the modifications demanded by the scheduler, impacting the jobs. On the other hand, in case 1, ODM will modify as much as possible. On Datazero2, the impossibility of compensating for case

1 can start a new negotiation from the offline part to deal with this event. However, in this thesis, we let the simulation run to see the impact of these cases, resulting in a SoC difference at the end of the time window.

The question now is: in which future time step t' can we compensate? To answer this question, we propose four policies: *Peak*, *Next*, *Last*, and *Load*. Figure 4.1 illustrates an example of these policies. The *Next* and *Last* policies execute the same search independently of the type of compensation (positive or negative). While *Next* takes the $t+1$, *Last* takes T. On the other hand, *Peak* and *Load* take different steps according to the compensation (positive or negative). *Peak* policy finds the higher power production (P_{prod}) peak in negative compensation and the lower peak in positive. This policy will "shave" the peaks, tending to a flat curve. *Load* policy considers the difference between demand (P_{load}) and production (P_{prod}). In positive compensation, this policy takes the higher difference $P_{load} - P_{prod}$, while in negative it takes the smaller one. In Figure 4.1, if the scheduler saves energy at step 1, it could compensate for it at step 3 (*Next*), step 12 (*Peak*), step 15 (*Last*), or step 14 (*Load*). Even if Figure 4.1 compensates only one time, the algorithm can select more than one, compensating E_{comp} entirely and following its policy (e.g., *Next* will take steps 2, 3, 4, etc.).

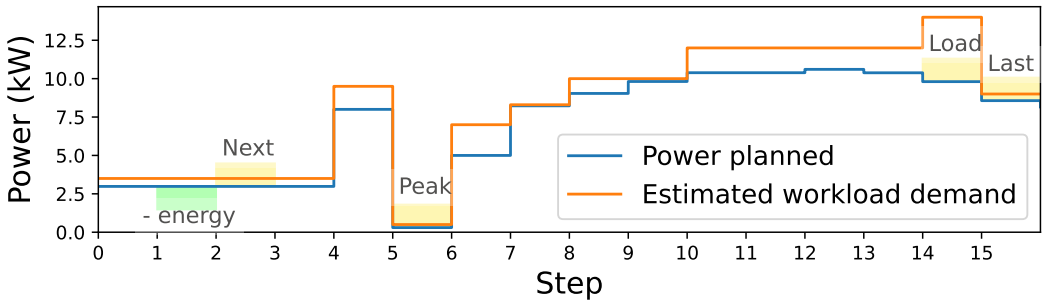

Figure 4.1: Compensation policies. This is an example of positive compensation (it uses less energy at step, so we need to increase the usage in the future). The blue curve is the offline usage plan (P_{prod}) and the orange line is the estimated demand (P_{load}). In this example, it saves some energy in time step 1 (see the green square). So, it can reintroduce this energy in future time steps (see the yellow squares).

4.2.3 Server configuration

Finally, the last part of the heuristic is transforming the power modifications into server configuration. Since the IT offline plan gives the P-state of the servers for each step t, we must adapt this plan for the new power. We propose a heuristic to find quick solutions. The four policies use the same heuristic to distribute the compensation. This heuristic has a list of all power and speed differences between two P-states, so it can fastly decide which one will impact the most on data center speed. First, it calculates the difference between the power usage in the offline plan and the power to use. If this difference is positive, it can improve the server configuration, speeding up or turning on servers. It searches on the list for the highest flops improvement below or equal to the power increment. It does it in the following order:

1. Find the highest improvement possible for the servers running some job;
2. Find the highest improvement possible for the servers not running jobs.

Taking Table 3.6 as an example, let's say that we have two servers running jobs: one on state 5 and another on state 10. The system has 30 W to increase. So, it will increase first the server on state 10 to state 1, because it will increase 14.41 Gflops (against 6.45 Gflops from state 5 to state 1). If a server is sleeping, it also considers the power needed to turn the server on. When the difference between offline and online power is negative, the algorithm must reduce the speed of the servers. So, it does the following:

1. Reduce the speed of servers not running jobs;

2. We calculate $(wall_j - elapTime_j)$ for each job running. Then, we sort the servers by this value in decreasing order. Finally, we reduce the speed of these servers following this order.

It stops after the first modification that lets the power usage lower or equal to the power production $P_{prod}(t)$. The second step will reduce servers' speed with more time to compensate for this reduction in the future. It is better to maintain jobs closer to finish with the maximum speed, granting that they will be complete.

4.3 Experimental environment

In this section, we present the environment used to run our simulations. This environment englobes IT servers definition, electrical elements, workload trace, and weather trace. We focus on simulating a time window of three days ($T_w = 259200s$), divided into time steps of 5 minutes ($\Delta t = 300$). We chose time steps of 5 minutes because it is neither too long to react to events nor too short for server transitions. For example, a time step of 1 minute is a good choice for adapting the battery usage, but it is not enough to turn on a server (Table 3.6 shows that a Gros server takes 164 seconds to finish the off→on transition). A longer time step can be too late to adapt battery usage. Concerning the server specification, we simulated a homogeneous data center using the GRID5000's Gros server. Table 3.6 presents their parameters. We demonstrate the experiments in a homogeneous data center to ease the understanding. However, we proposed different experiments in a heterogeneous data center in [32]. Our data center is composed of 400 servers with Gros specifications. We ignored other aspects, such as network and memory.

Considering the electrical components, the data center is composed of solar panels, wind turbines, batteries, and one hydrogen tank. The total size of the batteries is 800 kWh ($C_{bat} = 800kWh$). The efficiency of charging is 0.82 ($\eta_{ch} = 0.82$), and discharging is 0.82 ($\eta_{dch} = 0.82$). We defined the thresholds SoC_{min} and SoC_{max} as 20% and 90%. So, the battery energy range maintains the whole data center for 9.76 hours. We set the max charge and discharge power as 80% of the battery size ($P_{ch_{max}} = 640kW$ and $P_{dch_{max}} = 640kW$). The hydrogen tank has 20000 kg ($LoH_{max} = 20000$). At the beginning of the experiment, the battery starts half charged ($SoC(0) = 50\%$) and the hydrogen with 300 kg ($LoH(0) = 300kg$). We also specified that they should return at least to the same value as they started at the end of the time window ($SoC_{target} = 50\%$ and $LoH_{target} = 300kg$).

We have divided the experiments into two parts that use different weather and workload traces. The first part analyzes the decisions in critical cases (named *critical scenarios*), and the second part focuses on the random cases (named *random scenarios*).

4.3.1 Critical Scenarios

In the first part, we have taken two workloads from the Metracentrum dataset with the size of three days. The first workload has jobs arriving mainly on the first day and the second one on the third day. Figure 4.2 illustrates the demanded power of both workloads. We calculated this power using 400 Gros servers without power constraints (e.g., the servers are always available). The first workload has 3729 jobs and the second workload has 3158 jobs. Even if the second one has fewer jobs, it is more complicated to manage since the jobs arrive on the last day.

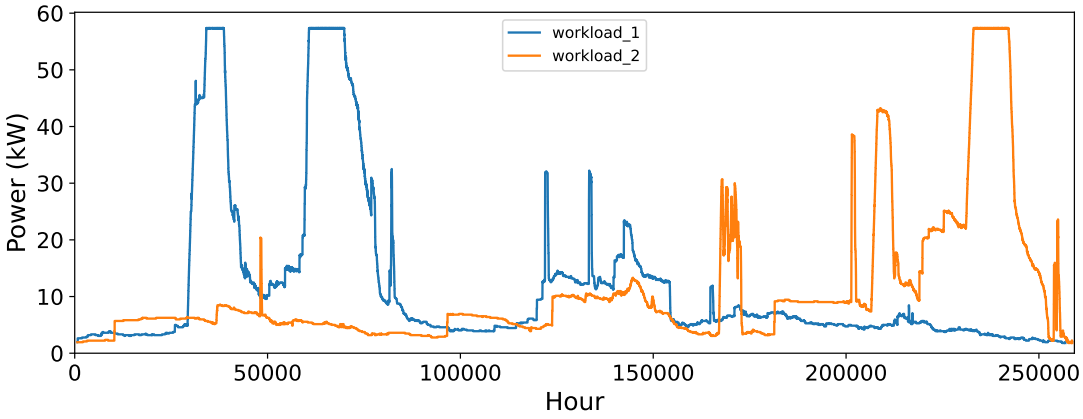

Figure 4.2: The power demanded for the critical scenarios.

We generated the three-day weather data using MERRA and renewable ninja for the critical scenarios [30, 109, 110]. Figure 4.3 shows the power production generated by the weather using the electrical components of our data center.

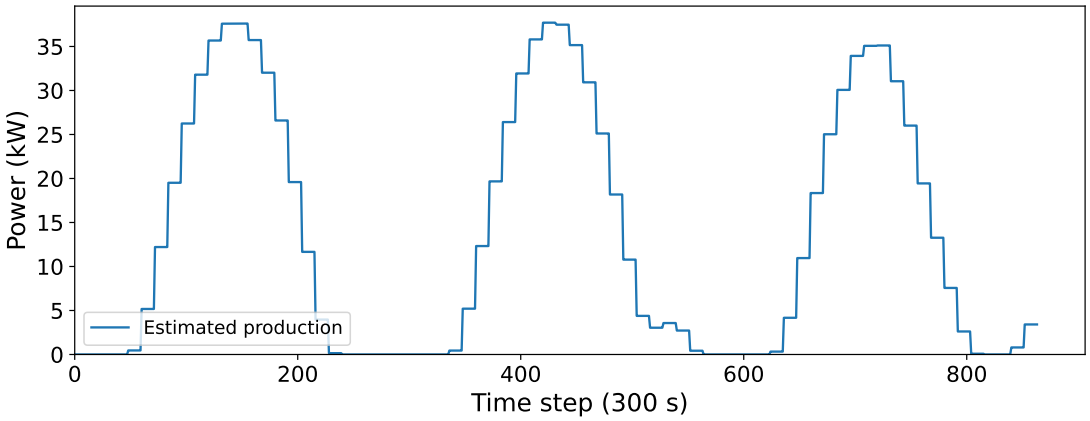

Figure 4.3: The power production for the critical scenarios.

We consider both workload and weather as predictions. Then, we use them to create the offline plan from Section 3.2.2. First, we solved the PDM model from Section 3.2.1 using the linearization proposed by Haddad et al. [16]. Then, we took the production (P_{prod}) and solved the ITDM model. The ITDM model presented in Section 3.2.1 is too complex to solve a three-day time window with 400 servers. So, we simplified the problem, ignoring the transitions on→off and off→on. This new model returns the number of servers

at each P-state at each time step and can be solved faster. After that, we have the offline plan.

The next step is to introduce noise in the predictions to emulate the variations in reality. Every prediction provides a range of where the forecast is likely to be. So, we generated noises inside this range. In the critical scenarios, we considered that the range is ±20%. Then, we created two new power productions: a worst-case and a best-case. In the worst-case scenario, all the productions are 20% lower than predicted. On the other hand, in the best-case power production, all productions are 20% higher. Figure 4.4 illustrates the power production. Therefore, we simulated the worst and best cases of production (the reason for the name critical scenarios).

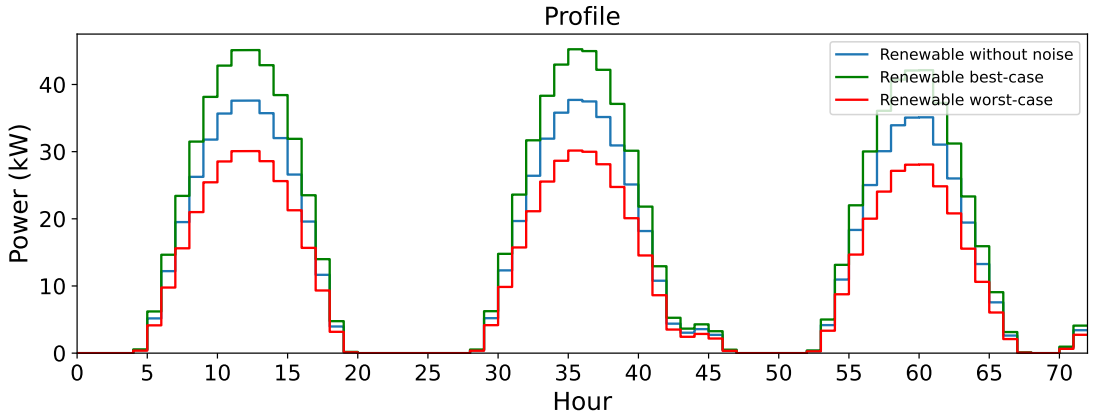

Figure 4.4: The power production with noise for the critical scenarios.

Regarding the workload, we introduced Gaussian noises in interarrival and job duration with a standard deviation of 20% of the predicted value. The relation between these workload noises and the power demand is more complicated than the power production since the power demand depends on the scheduling algorithm and queue sort. Figure 4.5 shows the impact of these noises on the power demand. Even if the impact on the workload power demand is not huge, these variances impact the offline IT plan since offline can indicate the wrong moment to turn on/off the servers. The workload's criticality comes from when the jobs arrive (in the end or beginning).

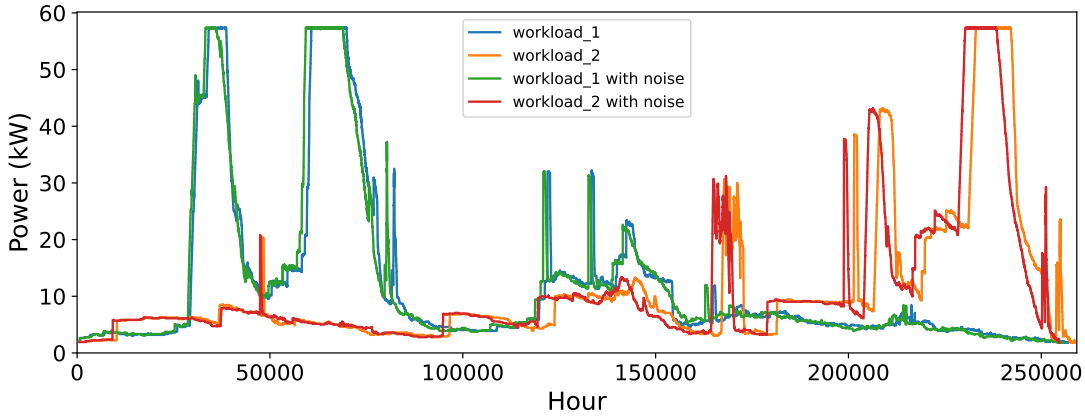

Figure 4.5: The power demand with noise for the critical scenarios.

Finally, we mixed the two productions (worst and best-case) with the two workloads (in the end and beginning), creating four scenarios:

1. *Critical 1*: Profile best-case and workload in the beginning;
2. *Critical 2*: Profile best-case and workload in the end;
3. *Critical 3*: Profile worst-case and workload in the beginning;
4. *Critical 4*: Profile worst-case and workload in the end;

The first scenario is the best possible, with more energy and having all the jobs in the beginning. Therefore, the scheduler has the energy and time to decide when to place the jobs. However, the last scenario is more complicated, with lower production and receiving the jobs on the third day.

4.3.2 Random Scenarios

In the *Random Scenarios*, we generated ten workload and weather traces. Figures 4.6 and 4.7 show the power production and demand, respectively. Then, we created ten offline plans, one for each pair of workload and profile. For example, workload 1 receives the production of profile 1, workload 2 receives profile 2, etc. We created the offline plans in the same way as the critical cases for each combination.

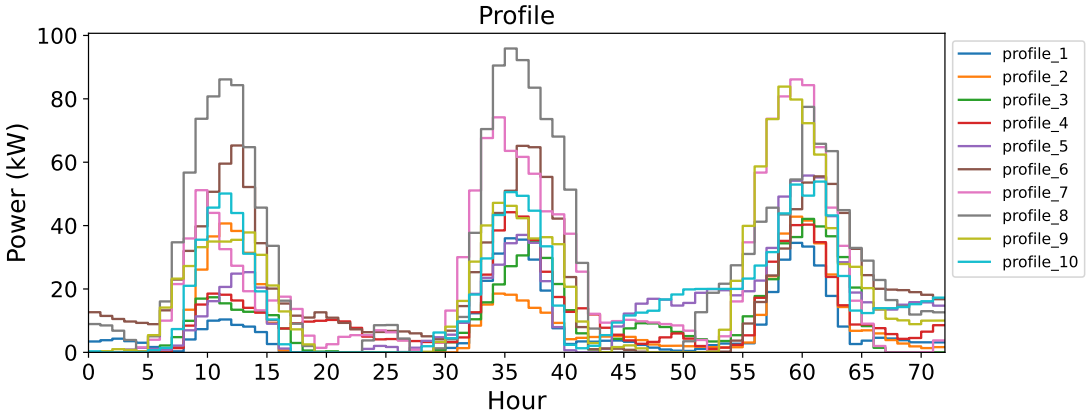

Figure 4.6: The power production for the random scenarios.

After creating the plans, we introduce noise in the values. Here, we applied Gaussian noises in power production and jobs interarrival and size. We increase the standard deviation to 50%, giving a wider range to generate values. Therefore, we have higher uncertainty. For each pair (workload + profile), we produced ten new workloads and profiles with noise, resulting in 100 combinations. We called these scenarios *Random Scenarios* because they are neither worst case nor best case.

In both parts (critical and random), besides the production and jobs noises, we also have the uncertainty from the walltime, presented in Section 3.3.1. So, our experiments introduce several uncertainties in different aspects of workload and power production. Furthermore, these uncertainties are cumulated from the different steps.

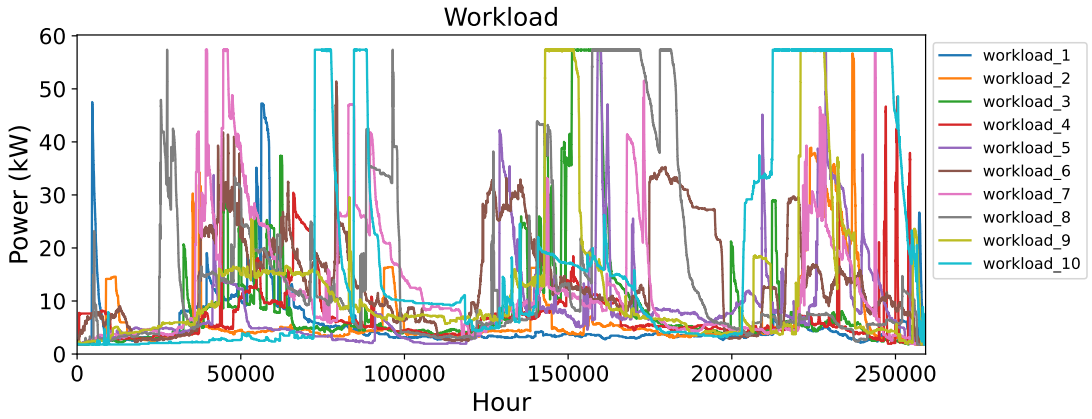

Figure 4.7: The power production for the random scenarios.

4.3.3 Baselines

We created three baselines to compare the results from the four policies. The baselines are *Follow plan*, *Power reactive*, and *Workload reactive*. *Follow plan* is an algorithm that applies the offline plan without changing it. This algorithm emulates the execution of only the offline side. *Power reactive* changes the server state according to the renewable power incoming. So, at each time step, it takes the power coming from renewable and calculates how many servers are possible to maintain running. It uses power from the batteries to maintain jobs running, if the renewable is not enough. *Power reactive* uses the server configuration heuristic presented in Section 4.2.3. *Workload reactive* turns on the servers according to the job's arrival. It starts with all servers off. For each new job submitted, it turns on the needed server at maximum speed if there is no server idle. After finishing a job, the server waits for T_{wait} seconds (using the DPM technique from Equation 2.9). The scheduler sedates the server if it stays idle for T_{wait} seconds. In all cases (baselines and compensation policies), if the battery's state of charge arrives at less than 20% (defined SoC_{min}), the scheduler kills the jobs and sedates all servers.

4.4 Results Evaluation

After describing the experimental environment, this section presents the results of the experiments. We detail the critical and random scenarios. After that, we discuss the results globally.

4.4.1 Critical cases

Scenario Critical 1

Scenario Critical 1 (Profile best-case and workload in the beginning) has the jobs arriving at the beginning, and the production is higher than expected. However, this scenario is tricky. Since the battery starts with $SoC = 50\%$, if the scheduler starts too many jobs in the beginning, this can lead to a very low SoC on the first day. This is exactly what happened with *Workload reactive* in this scenario. Figure 4.8 shows the evolution of the state of the charge in the *Workload reactive* execution. At step 264 (79200 seconds after the simulation begins), *Workload reactive* has less than 20% of SoC. So, the scheduler kills several jobs.

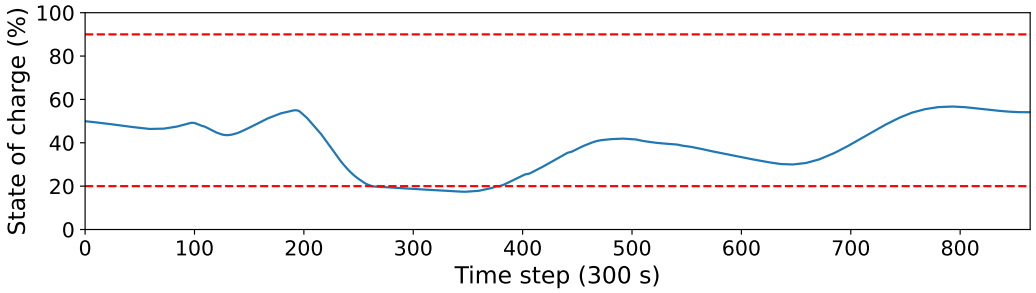

Figure 4.8: State of Charge for *Workload reactive*.

Figures 4.9 and 4.10 give the battery level at the end of the time window and jobs states, respectively. Figure 4.10 shows two graphs. The first one considers only the number of jobs, ignoring their size. Therefore, all jobs are equal. In the second graph, we consider the size of the job. So, bigger jobs have more importance than smaller ones. The second graph illustrates the mass of work to do.

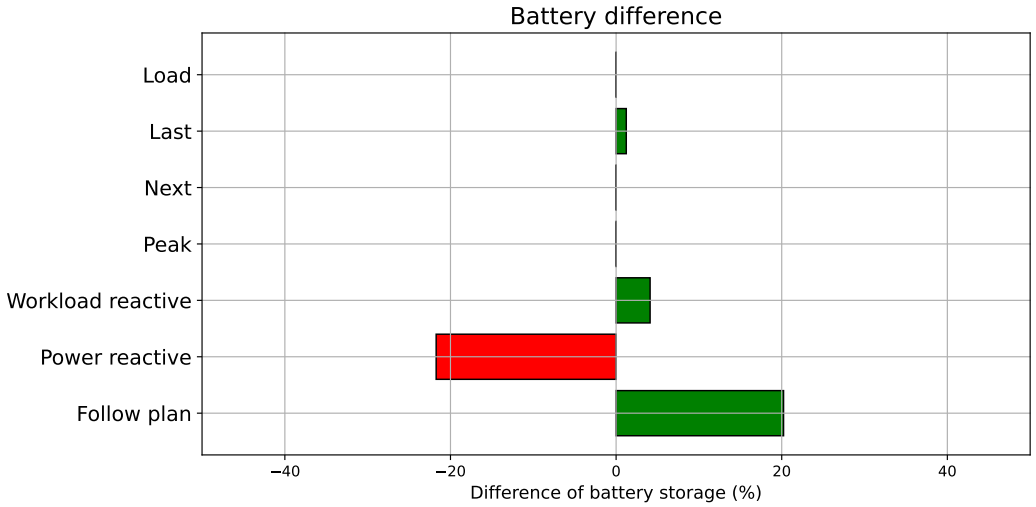

Figure 4.9: Difference between the battery target level (50%) and the real battery level at the end of the time window for scenario critical 1.

It is important to analyze the battery level and finished jobs together. For example, *Follow plan* saves energy, finishing with 20% more battery than the target. This result seems very good, but analyzing Figure 4.10, *Follow plan* has the lower finished jobs and higher killed jobs. Besides, it kills a lot of big jobs, resulting in only 55.72% of finished jobs considering the size. This result illustrates the importance of reacting to real events and using the saved energy to improve QoS. *Follow plan* tends to kill bigger jobs since it does not adapt the plan to maintain them running.

Power reactive has the worst battery level, with an SoC of around 28% (difference of -21.729269%). This algorithm allocates all incoming power from renewable sources to servers, not recharging the battery. The battery slightly recharges due to power fluctuations (e.g., the server is idle, so the incoming renewable power recharges the battery instead of going to the server). So, *Power reactive* uses the battery to avoid killing jobs, but it does not compensate for this change. *Power reactive* is the second-worst finished

Figure 4.10: Jobs states at scenario critical 1. The first graph (above) considers only the number of jobs, ignoring their size. In the second graph (below), the jobs' size is considered.

jobs metric (ignoring or considering the size), with some jobs reaching the walltime. Since it only uses the battery to avoid killing jobs, sometimes it let the servers at slower speeds, increasing the possibility of reaching walltime. It also kills some jobs because it also reaches the SoC_{min}.

As mentioned at the beginning of this section, *Workload reactive* can not manage well the state of charge, killing some jobs. Even so, it finishes with a good battery level. The DPM technique helps to save some energy, letting the incoming renewable to recharge the battery. Since it always put the servers at maximum speed, no job reached the walltime. It has the third-worst finished and killed jobs (ignoring the size). All the policies are very close to the target level. Just *Last* saved more than the target because it puts all the compensations in the end. Therefore, *Last* can not use all the power before the end. Considering the metrics about the job, all policies finished more jobs and killed less than the three baselines. The best one is *Last* with 98.87% of the finished jobs considering the number of jobs and 94.88% considering the size. In addition, it has the lowest killed jobs. *Load* places the compensations in the steps where it expects high demand. In this case, it is worth it with the second-best results. We will see in future cases that this behavior is dangerous.

Next has the worst job result among the policies. It also kills more big jobs than the other policies and *Workload reactive*. This result can be explained by comparing *Next* and *Last*. For example, let's say we saved energy in step 0. So, *Last* increase the usage in the last possible step. If at any moment between step 0 and the last step, it is necessary to

increase the usage, *Last* can migrate the energy from the end and use it now. On the other hand, *Next* expended this energy as soon as possible. Therefore, *Next* can arrive in some moments without energy to avoid killing jobs. Finally, *Peak* has the third-best result, with some jobs reaching the walltime. Since it shaves the power usage (negative/positive) peaks, it can reduce speed in critical moments (e.g., with several jobs running). However, it can maintain the big jobs running. Even if *Peak* finishes fewer jobs than *Load* (a difference of 1.66 percentage points), *Peak* approximates the finished jobs considering the size (a difference of 0.98 percentage points).

The second analysis is regarding wasted energy. This metric is the energy expended not computing finished jobs, englobing, for example, the energy used in killed jobs, turning on/off servers, and letting servers idle. Figure 4.11 shows the results. *Workload reactive* turns on resources on demand and does not let idle servers available too much. Therefore, it has the best wasted energy metric compared with the other algorithms. The worst case is *Power reactive* which turns on servers according to the power available, not the demand. *Follow plan* and the four policies are guided by the offline plan. Therefore, the noise introduced in the real workload can lead to some mismatch between demand and production. Since *Follow plan* kills more jobs than the policies, this metric is higher for this algorithm. The energy expended in killed jobs is wasted since the result of the killed job is useless. *Last* has the second-best wasted energy since it runs more jobs than the other algorithms.

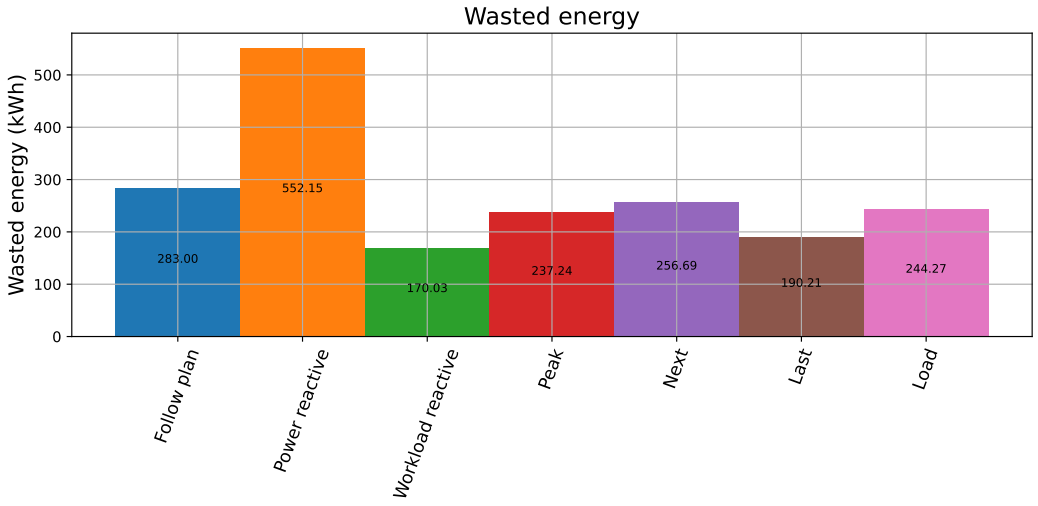

Figure 4.11: Wasted energy at scenario critical 1.

Finally, Figure 4.12 demonstrates the slowdown of the finished jobs. As mentioned before, this metric is complicated to compare with different numbers of finished jobs. We can see that *Workload reactive* has some jobs with a very high slowdown. It happens due to the long period with the SoC below SoC_{min} (as illustrated in Figure 4.8). So, it has a long period without servers running. *Load* has a good mean and median. In this case, *Load* can place the compensations close to the jobs' arrival.

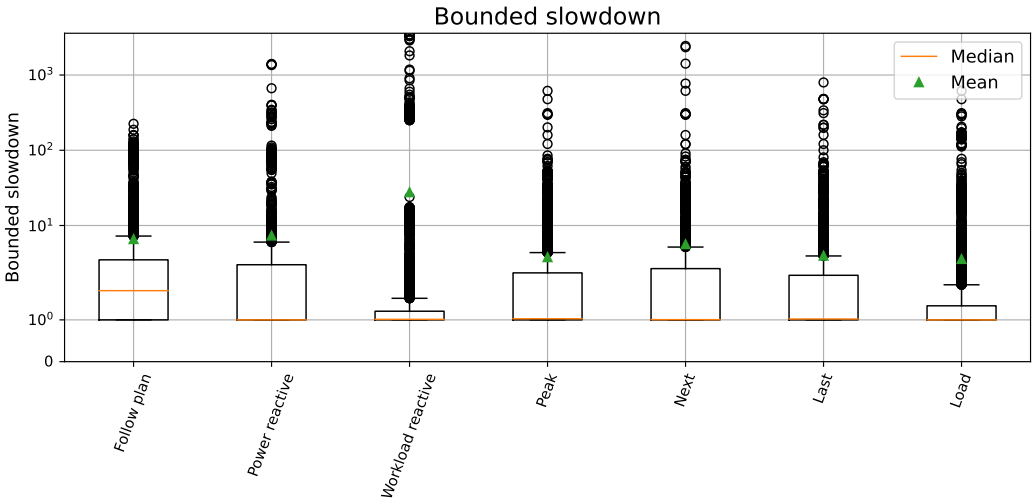

Figure 4.12: Bounded slowdown at scenario critical 1.

Scenario Critical 2

Scenario Critical 2 has more energy (Profile best-case) and the jobs arriving on the third day. The policies do not have a long time to compensate since the load comes on the last day. Figures 4.13 and 4.14 demonstrate the battery level and finished jobs.

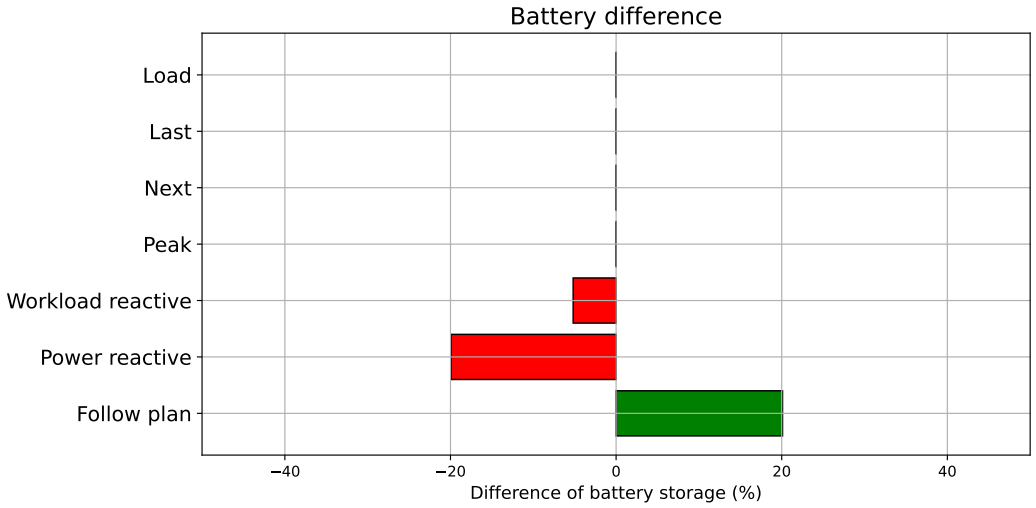

Figure 4.13: Difference between the battery target level (50%) and the real battery level at the end of the time window for scenario critical 2.

Workload reactive finished all jobs. This scenario is the best one for this algorithm since it can fulfill the battery in the first two days, consuming all in the last day. Even so, it finishes with a battery deficit (-5.186277%). *Follow plan* has the second-worst finished jobs and the worst killed jobs (in number) but finishes with a battery surplus of around 20%. So, it misses the opportunity of using this surplus to avoid killing jobs. Considering the size, it kills more than 40%. *Power reactive* has the third-best finished jobs (in number), but with a good part not completely finished. These jobs are still running at the end of the time window, so we do not know if they would finish or not. However, it has the third-worst finished jobs considering the size of the jobs. Like in Scenario Critical 1, *Power reactive* has a large battery deficit, with around -20%. All policies have a perfect battery level, finishing with the SoC at around 50% (the target level). All policies kill fewer jobs than the *Follow plan*, using the higher production to do so. However, they kill jobs to arrive at the battery target level, avoiding using more battery than predicted.

Just *Next* policy finishes fewer jobs than *Follow plan* in number, but more considering the size. This policy consumes all incoming energy as soon as possible, not having too much to use on the last day. *Peak* has the third-worst finished jobs metric considering the number, worst than *Power reactive*. This policy puts the surplus energy in the moment with less energy (negative peak). This behavior creates a more constant number of servers available during the time window. However, this case has a load in the end. So, reintroducing the energy in the *Peak* approach does not help to finish more jobs. On the other hand, *Peak* finishes bigger jobs than *Power reactive*. It can finish bigger jobs due to its constant number of servers behavior. However, since the majority of jobs arrive at the end, it is better to place the positive compensations at the end (like *Load* and *Last*).

We can see a small difference in jobs finished (in number) between *Peak* and *Load*, where *Load* puts the energy in the moments with a higher difference between demand and production. It helps to increase almost 1% of finished jobs and decreases around 2% of

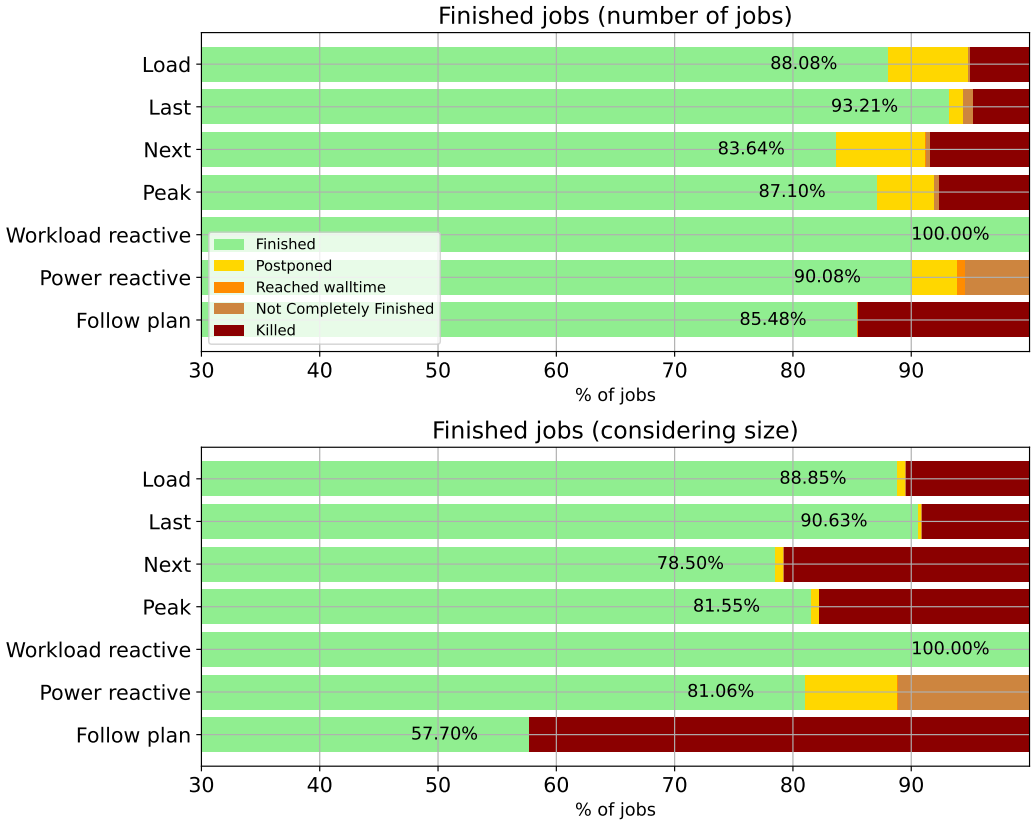

Figure 4.14: Jobs state at scenario Critical 2. The first graph (above) considers only the number of jobs, ignoring their size. In the second graph (below), the jobs' size is considered.

killed jobs. Besides, *Load* can increase the number of bigger jobs finished compared to the *Peak* policy. Nevertheless, the best results come from *Last*. This policy stocks all surplus at the last moment, allowing to execute more jobs. It can finish 3.13% (number of jobs) more than the third-best, the only above 90% finished jobs considering the size (besides *Workload reactive*), and having a perfect level of battery at the end of the time window.

Considering the wasted energy, Figure 4.15 illustrates the results in this scenario. Again, *Workload reactive* has the lowest wasted energy due to its workload reactiveness approach. The worst one is *Power reactive*, for the same reason as scenario Critical 1. It turns on and increases the speed according to the incoming power, not the demand. *Follow plan* has not-so-bad wasted energy, even with a high killed jobs. *Peak* and *Next* wasted more than the *Follow plan*. Both policies place the energy at the wrong moments. For example, if *Next* has a positive compensation (increase the usage) in step 1, it will increase in step 2. However, the demand is in the end. Then, the energy is wasted on idle servers. *Peak* policy has a similar problem. With the load in the beginning, it was not a problem for *Peak* (it can delay the jobs for later execution). Nevertheless, in this scenario, it puts positive compensations in the moments with lower usage, leading to more moments with idle servers. On the other hand, *Last* and *Load* have better energy usage than the *Follow plan*. Both policies put the energy in the right moment (last steps).

Regarding the bounded slowdown, Figure 4.16 shows the results. *Workload reactive* has the best results due to its workload reactiveness. Without the SoC problem from the previous scenario, it can execute all jobs as soon as they arrive. The policies present a

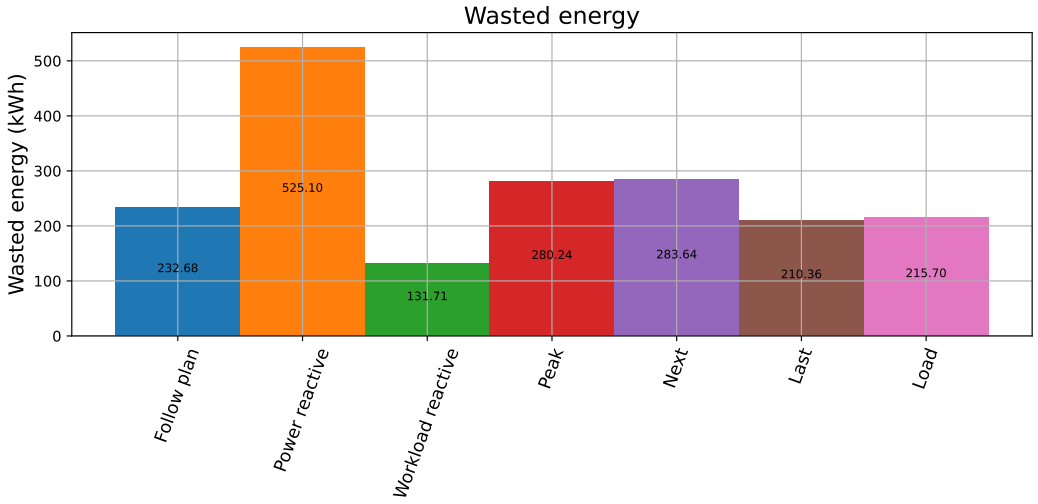

Figure 4.15: Wasted energy at scenario Critical 2.

better median than *Follow plan* but a higher mean, due to several jobs with bounded slowdown higher than 100. Yet, since *Last*, *Peak*, and *Load* finish more jobs than *Follow plan*, it is complicated to compare them.

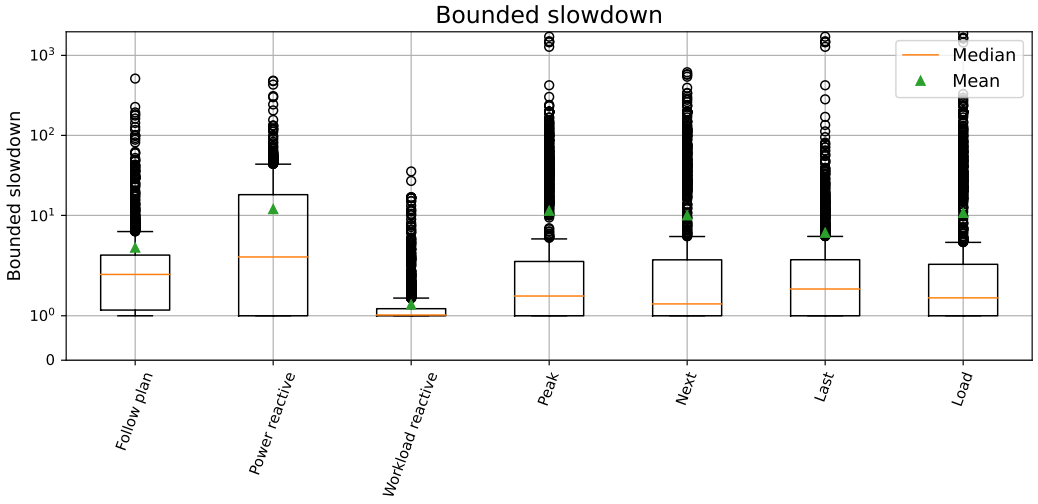

Figure 4.16: Bounded slowdown at scenario Critical 2.

Scenario Critical 3

Scenario 3 introduces less energy than predicted with the jobs arriving on the first day. Figure 4.17 gives the final battery level in this scenario. This figure demonstrates that the policies finished closer to the target level than the baselines, with -6.177244% in the worst case (*Load*). This scenario is particularly difficult to finish at 50% since the policies migrate power to the first day expecting to compensate for it on the second and third days. However, it receives less energy than expected, being a little far from the target level. Nevertheless, the policies are way better compared to the *Workload reactive*, *Power reactive*, and *Follow plan*.

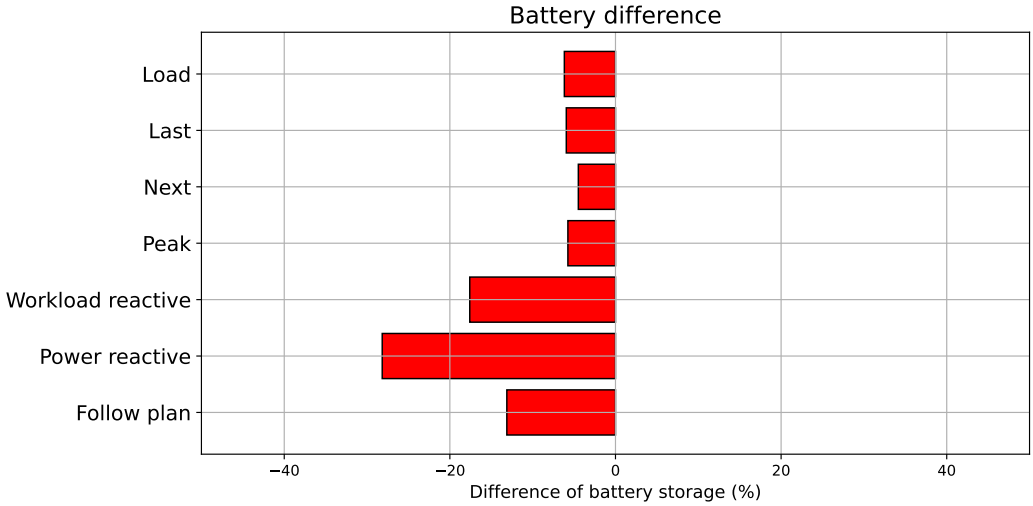

Figure 4.17: Difference between the battery target level (50%) and the real battery level at the end of the time window for scenario critical 3.

Concerning finished jobs, Figure 4.18 illustrates the results. As presented in scenario 1, *Workload reactive* has a problem with the load on the first day because it will place all jobs as soon as they arrive, drying the battery. We can see that scenario 3 is even worst, having more than 20% killed jobs in number and more than 40% in size. *Power reactive* finishes more jobs than any other algorithm, with 85.60% (in number) and 61.43% (in size). This algorithm follows the real production to set the servers' speeds. This behavior helps to start several jobs, but it can not maintain the jobs running when the battery arrives at SoC_{min}. So, it has more than 10% of killed jobs considering the number and almost 40% considering the size. *Follow plan* finishes more jobs than the policies in both number and size. However, it kills several jobs, with more than 20% in number and almost 50% in size (the worst result).

Among the policies, *Load* has the worst result of finished and killed jobs in both number and size, compared to the other policies. In this scenario, *Load* puts any positive compensation on the first day, approximating too fast to SoC_{min}. Then, it has to kill several jobs to avoid this boundary. This result shows that *Load* aggressivity does not always worth it. On the other hand, *Next* stays close to the SoC planned. In this case, it can manage the SoC better, finishing more than 80% of jobs in number. Among the policies, *Next* is only worst than *Peak*. As mentioned before, *Peak* can smooth the peaks maintaining more servers available constantly. We can see that it finishes fewer jobs in number than *Next* but more in size. *Peak* has a similar result to *Follow plan*, using less battery and killing fewer jobs, highlighting the need for reactivity.

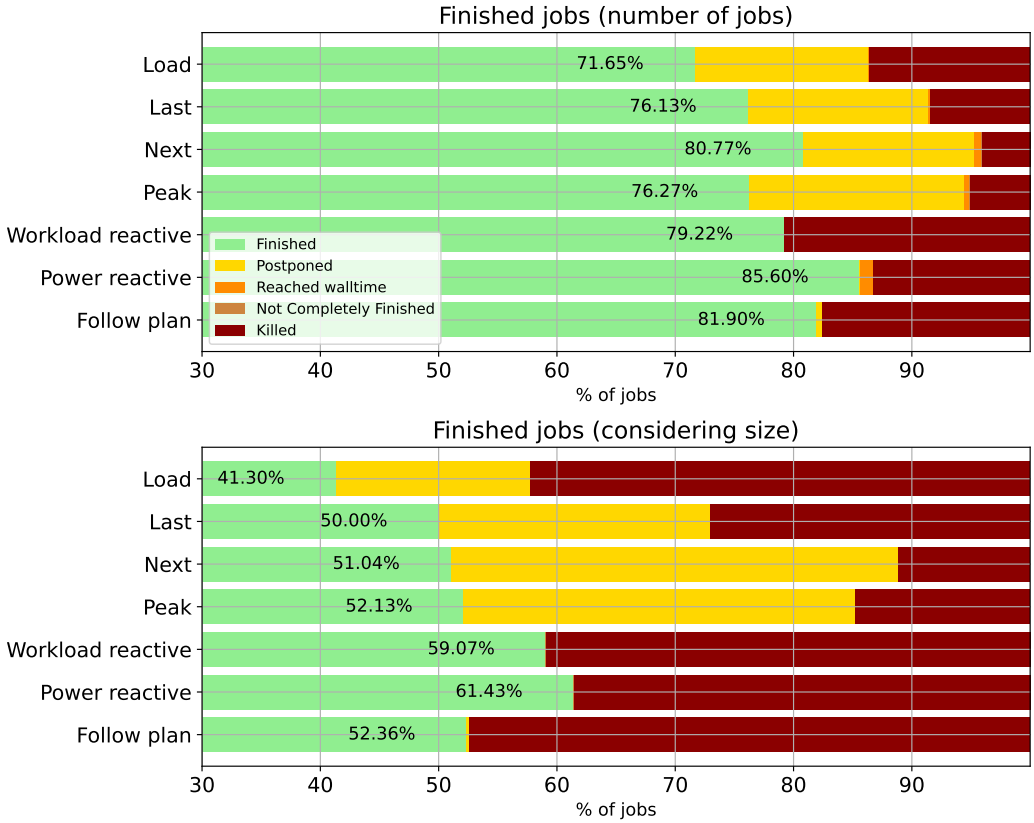

Figure 4.18: Jobs state at scenario critical 3. The first graph (above) considers only the number of jobs, ignoring their size. In the second graph (below), the jobs' size is considered.

Figure 4.19 shows the wasted energy in this scenario. We can see that all policies present lower wasted energy than the baselines, a crucial result in a scenario with less energy available. *Next* policy wasted 45.96% less energy than *Workload reactive* (the best baseline). *Load* has the worst wasted energy among the policies due to the number of killed jobs. Even so, *Power reactive* has lower killed jobs than *Load*, but it has higher wasted energy. *Power reactive* follows the power available, turning on servers according to the power available. Therefore, it expends more energy than the other algorithms in transition states.

Finally, Figure 4.20 shows the bounded slowdown. *Workload reactive* has a similar result to Scenario 1. When it arrives at SoC_{min}, it has a long time without servers available, increasing the waiting time and slowdown for some jobs. However, it still has a very low median, showing that the majority has a small slowdown. In this scenario, the policies let the jobs wait longer, delaying the starting time for the moment with enough energy to finish them. So, the policies have higher mean and median slowdowns.

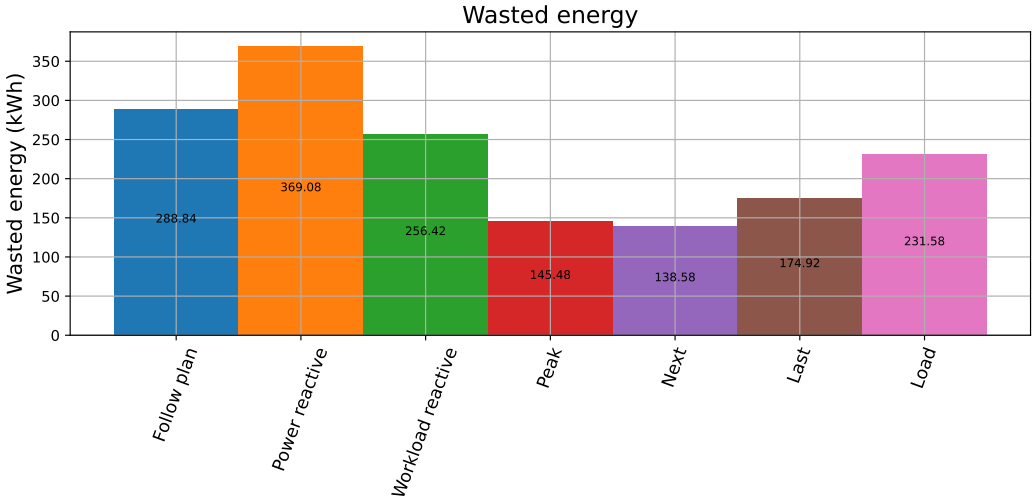

Figure 4.19: Wasted energy at scenario critical 3.

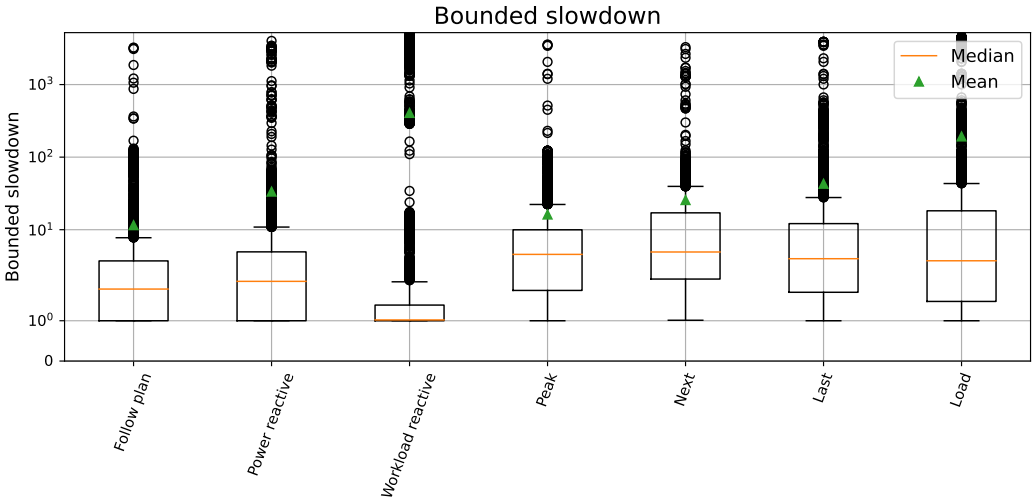

Figure 4.20: Bounded slowdown at scenario critical 3.

Scenario Critical 4

The last critical scenario is the harder one, with less energy and jobs arriving on the last day. Figure 4.21 shows the impact on the battery level. Both *Workload reactive* and *Power reactive* finished with a deficit higher than 30%, with 31.45% and 30.78%. Both ended the battery with less than the SoC_{min} of 20% (18.55% and 19.22%). *Follow plan* finished with the SoC at 35.72% (-14.28%), far from the target level. Nevertheless, the policies finished very close to 50%. Since the first two days provide less energy, the policies can adapt the usage of the last day to approximate the target level. However, Figure 4.22 shows the impact on QoS.

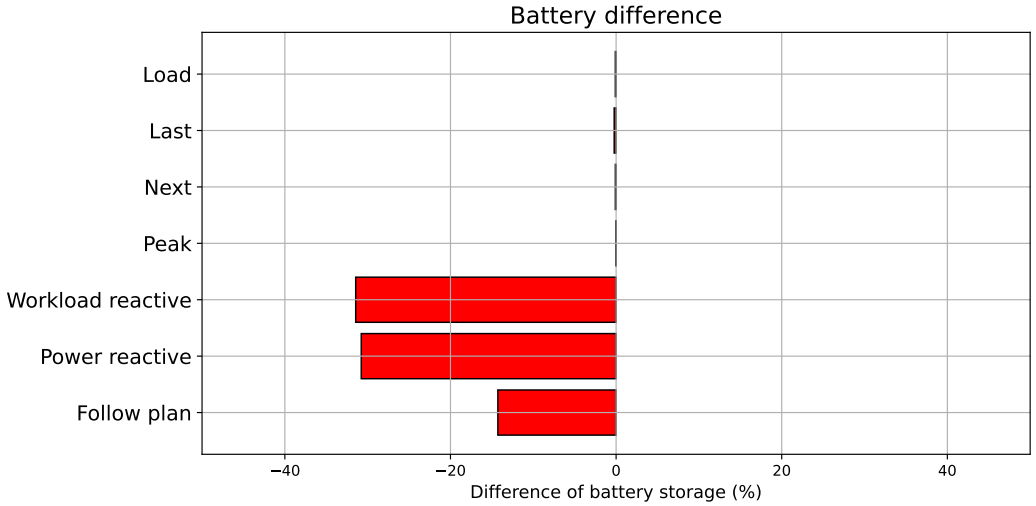

Figure 4.21: Difference between the battery target level (50%) and the real battery level at the end of the time window for scenario critical 4.

Figure 4.22 demonstrates that the policies finished fewer jobs than the baselines in number and size. Considering the number of killed jobs, only *Peak* kills fewer jobs than the other algorithms, but very close to the *Power reactive*. In a scenario with less energy, the policies will impact the QoS, approximating the battery's SoC to the target level. *Follow plan* has the best number of finished jobs, but it finishes mainly small jobs. It kills more than 40% of the jobs considering the size (the worst result). *Workload reactive* has the second-best finished jobs in number and the best in size. However, *Workload reactive* kills more jobs in number than *Power reactive*, *Peak*, *Last*, and *Load*. The battery goes below 20% on the last day, killing several jobs. The same happens to *Power reactive*. Even with several jobs killed, Figure 4.23 indicates that all policies wasted less energy than the baselines. The policies adapt the plan to maximize energy usage. In a scenario with low energy production, it is essential to improve energy usage. Figure 4.24 shows that the policies also impact the Slowdown, having worst results than the baselines.

The results in this scenario indicate that the policies degrade the QoS (finished jobs and slowdown) to approximate the target level. The policies' compensation imposes the adaptations on power usage and server configuration. In this scenario, these adaptations demand an available servers reduction or reduce their speed.

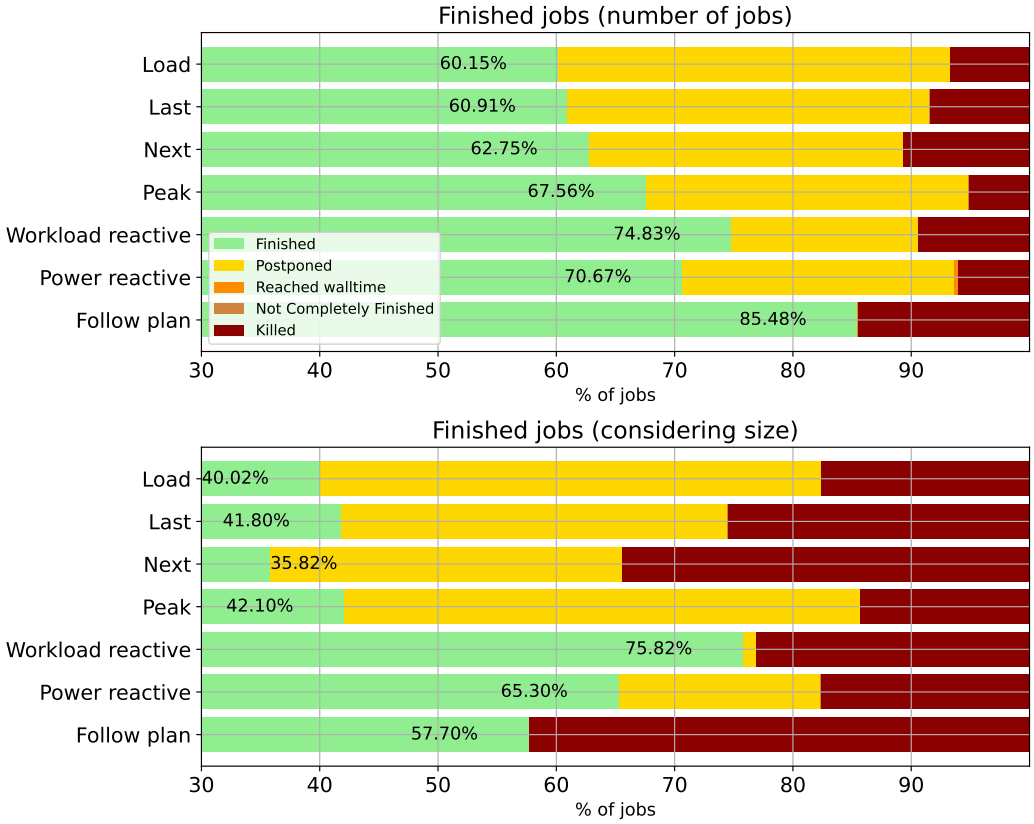

Figure 4.22: Jobs state at scenario critical 4. The first graph (above) considers only the number of jobs, ignoring their size. In the second graph (below), the jobs' size is considered.

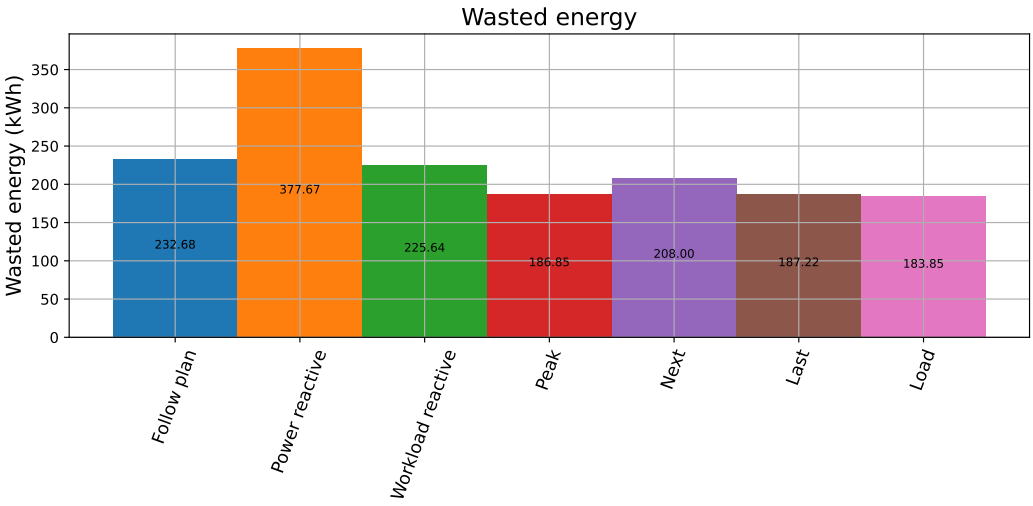

Figure 4.23: Wasted energy at scenario critical 4.

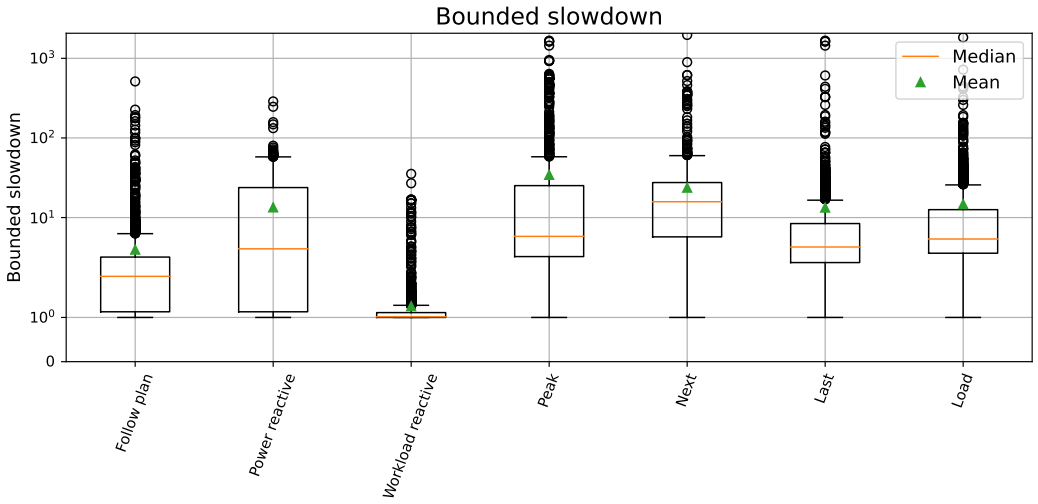

Figure 4.24: Bounded slowdown at scenario critical 4.

4.4.2 Random cases

After presenting the critical scenarios, this section demonstrates the results of 100 random cases. These random cases vary the power production and workload randomly, following a Gaussian noise. So, they are not always the worst or best, like in critical scenarios. We do not present the slowdown in this scenario. The workload and power productions vary a lot between executions. So, it is inappropriate to compare them. First, Figure 4.25 illustrates the state of charge difference from the target level. The red line indicates the perfect SoC at the end of the time window (0% of difference from the target level). We can see that the three baselines (*Workload reactive*, *Power reactive*, and *Follow plan*) have a large variance in the results. An important thing to notice is that even in non-critical scenarios, the minimal values of *Workload reactive* and *Power reactive* can arrive at -30%. This value means they finished with the battery level at 20%, which is the SoC_{min}. *Follow plan* finishes with a higher Median and Mean, indicating that it saves more energy than the other algorithms. However, it also has a large variance.

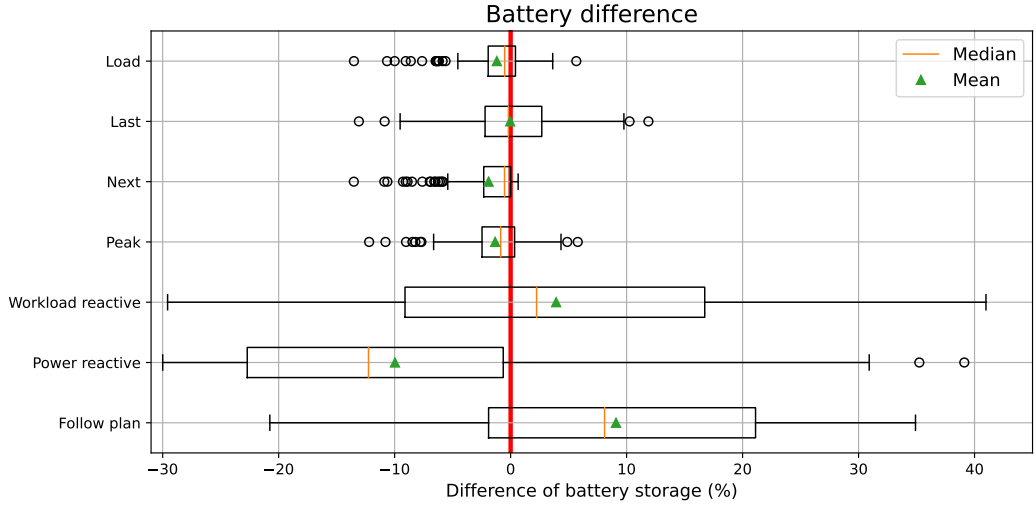

Figure 4.25: Difference between the battery target level (50%) and the real battery level at the end of the time window at 100 random cases. The line shows the standard deviation.

The policies maintain the SoC between -10% and 10%, with some outliers below/above these values. The median and mean are very close to the target level, showing they can approximate the target level which is their main objective. *Last* has the larger variance among the policies, but with median and mean almost perfect. This heuristic compensates in the last moments. Sometimes, this behavior can reduce the possibility to use the energy since the heuristic will not have enough time.

Regarding the QoS, Figures 4.26 and 4.27 illustrate the finished and killed jobs. *Workload reactive* has the best percentage of finished jobs (in number and size). This result is expected since this algorithm dries the battery to maximize the number of finished jobs. *Workload reactive* also has low median and mean killed jobs. However, concerning the number of killed jobs, it has some results higher than 15% (with two higher than 20%). *Power reactive* has good finished jobs but never finishes more than 97.90%. This algorithm also has some executions with more than 15% of killed jobs in number. The worst execution is *Follow plan*, with high mean and median killed jobs (in number and size) and low finished jobs in size. Combining the state of charge from Figure 4.25 with this result, we can notice that *Follow plan* misses opportunities of using more power to execute more

jobs or maintain big jobs running.

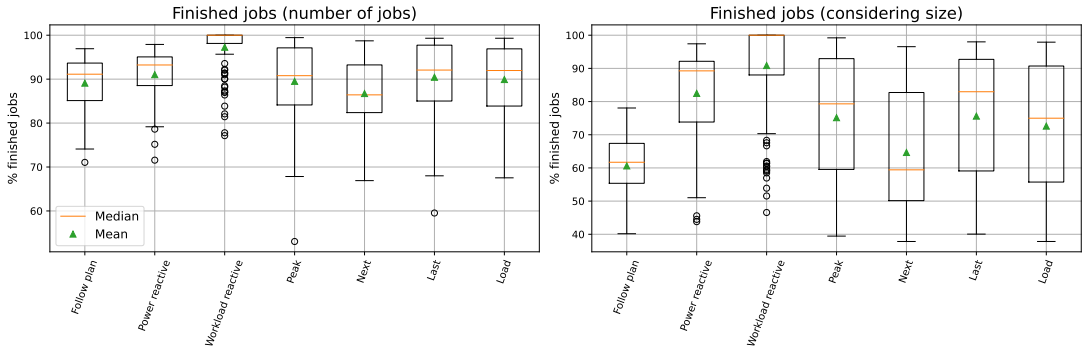

Figure 4.26: Finished jobs at 100 random cases.

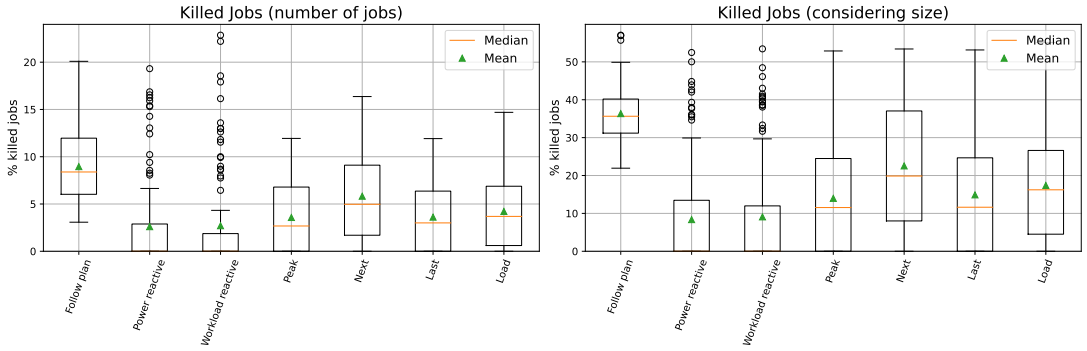

Figure 4.27: Killed jobs at 100 random cases.

The policies have good finished jobs (in number), with *Last* being a little below the *Power reactive*. They also have some executions with almost 100% of finished jobs (above 99%). Considering the number of jobs killed, *Last* and *Peak* maintain the results in control, with no value above 15%. *Load* has the worst case close to 15% and *Next* with 16.37%. Considering the size, the policies have worst results than *Power reactive* and *Workload reactive*. These jobs are harder to maintain running because they demand more time and power. So, the policies can kill them to guarantee the battery target level.

Finally, Figure 4.28 illustrates the wasted energy of over 100 executions. The best execution is *Workload reactive*. As mentioned before, this heuristic focus on running jobs, letting the servers down until they are needed. Besides, the DPM technique reduces wasted energy. The worst algorithm is *Power reactive*. This algorithm turns on servers even if they are not necessary. *Follow plan* is better than three policies (*Peak*, *Next*, and *Load*). The best policy is *Last*, mainly because it runs more jobs than the other policies and *Follow plan*. While *Follow plan* uses only the energy planned, the policies can reintroduce the energy from power variations. However, these policies change the usage using simplified heuristics. So, they can place the energy in moments without jobs to run. *Last* is the policy that suffers the least from this problem because it places the compensations in the end. Therefore, it can re-migrate this energy at any moment before the end. Section 4.4.3 presents an overview of the advantages and problems of these policies.

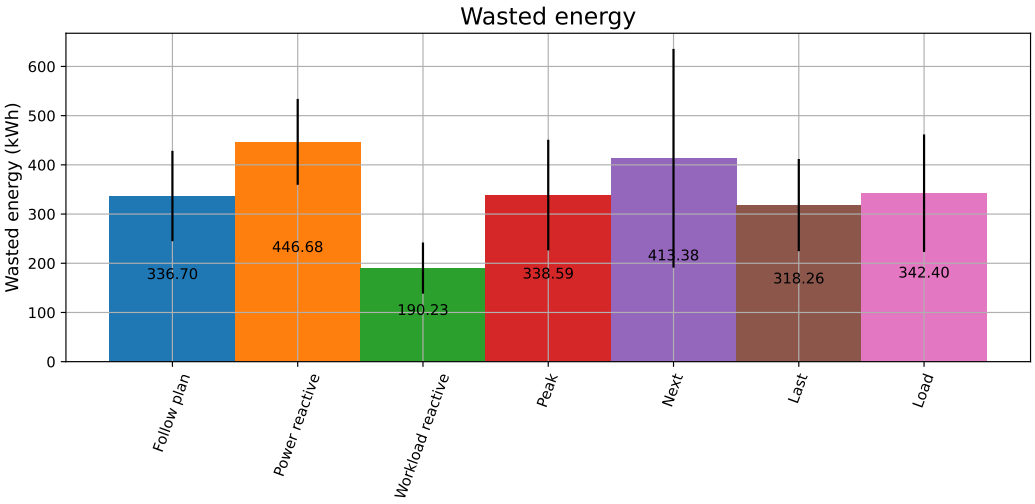

Figure 4.28: Wasted energy at 100 random cases.

4.4.3 Discussion

After presenting all results, this section discusses them generally. We highlight the advantages and disadvantages of all algorithms. Finally, we detail the gaps in the four policies that the following sections explore. The first algorithm is *Follow plan*. This algorithm does not react to real events, applying the offline plan with no modifications. The results show that this approach tends to kill big jobs. Since the workload is composed of several small jobs (see Figure 3.3), it can finish several jobs by number but not so much considering their size. Figure 4.22 illustrates this case, where the algorithm finishes 85.48% of jobs considering the number but only 57.70% considering the size. This behavior also happens in random cases, where it finishes more than 90% of the jobs in number but just more than 60% in size (considering the median). Besides, *Follow plan* does not use the surplus energy to run more jobs in scenarios 1 and 2. So, it saves energy but kills several jobs. This behavior can lead to highly wasted energy, like in critical scenario 1 (see Figure 4.11). Since *Follow plan* does not adjust the plan, it finishes with higher SoC in cases with higher energy and lower SoC with less energy. The 100 random cases show that the minimal value is close to -20% and the maximum value is above 30%. These results highlight the importance of plan adaptations.

Then, a question comes up: Why not use fully reactive algorithms? Then, we presented *Power reactive* and *Workload reactive*. *Power reactive* adapts the servers' speed according to the power available from renewable. It uses the battery to maintain a server available if the incoming renewable production is not enough. *Power reactive* finishes with a very low battery level, even in scenarios with more energy incoming from renewable. This algorithm only recharges the battery if the servers do not use the incoming renewable (e.g., a server stays idle). However, it is not enough. Another problem in *Power reactive* is wasted energy. It has the worst results in every scenario. Since it does not use an offline plan, it wastes a lot of energy in transition states. For example, if renewable production is high, it will turn on several servers. Nevertheless, if the production drops in the next step, it turns them off. So, it wasted energy just reacting to the power available.

Workload reactive is a very good algorithm from the QoS point of view. Considering the slowdown, for example, jobs do not wait too long to be placed. In addition, it can finish more jobs than the other algorithms. However, it can be too aggressive. It kills several jobs due to poor battery management in critical scenarios 1 and 3. Figure 4.8 illustrates that *Workload reactive* places all the incoming jobs on the first day, dropping the battery level too fast. When SoC arrives at 20%, *Workload reactive* turns off all servers, killing several jobs and increasing the slowdown. This also happens in critical scenario 3. In critical scenarios 2 and 4, it finishes with less battery level than the target. In random scenarios, the final battery level of *Workload reactive* varies a lot, going from -30% to more than 40%. So, this algorithm has poor battery management, like *Power reactive*. Even if it seems appropriate to use the battery to maximize the number of finished jobs, the next time window will not have energy in the batteries to use. Therefore, both reactive algorithms are not viable for a renewable-only data center.

So, we proposed four policies to adapt power usage, mixing the offline plan with reactiveness. *Peak*, *Next*, *Last*, and *Load* fulfill their main objective, approximating the battery level to the target level. In the profile worst-case scenarios (3 and 4), they degrade the QoS to approximate the battery level to the target. They do it by reducing power usage before the end of the time window. On the other hand, they use the surplus energy from scenarios with profile best-case (1 and 2) to run more jobs. In scenario 1, they are even better than the *Workload reactive*. The policies finished very close to the target level in the random cases. To do so, they impact the total finished jobs but have a more controlled

number of killed jobs than the baselines. For example, *Last* never killed more than 12% (in number). Therefore, these experiments show the impact on QoS of respecting the battery level.

It is hard to indicate the best policy. *Last* policy is the best one in critical scenarios 1 and 2. It finishes more jobs in number and size in critical scenario 1 and the second best in critical scenario 2. These scenarios have more energy, so *Last* puts the positive compensations in the last step. When it needs more energy to maintain servers running, it takes the energy from the same step. So, it is more likely for *Last* to use the surplus energy. The drawback of the *Last* policy is that the last step can be too late to use the energy (see Figure 4.9), and it can miss opportunities to turn on servers to run more jobs. For example, *Load* has a better slowdown than *Last* in critical scenario 1 (see Figure 4.12). *Load* turns on servers in the moments where they are needed. So, when the jobs arrive, servers are waiting for them. *Load* is the second best in scenarios 1 and 2. With more energy, it puts energy in the steps with a higher deficit between production and demand. However, the prediction is not perfect. So, *Load* can improve the wrong step. *Next* has the worst results in these scenarios because it uses the surplus as soon as possible. Therefore, *Next* makes it difficult to find energy in future steps to avoid killing jobs. Finally, *Peak* has a balanced result in scenarios 1 and 2. It tends to maintain a more constant number of servers available over the steps.

Regarding critical scenario 3, *Next* and *Peak* are good. This scenario has less energy available. Therefore, *Next* adapts as soon as possible the usage. *Peak* reduces the usage peak, maintaining a more stable usage. The behavior of both policies helps to avoid the lower battery boundary without starting too many jobs that can not be finished. *Load* is too aggressive here, dropping the SoC too fast and killing several jobs. Here, *Last* is not so good. It does not adapt the usage on the first day, arriving faster at SoC lower boundary. In the last critical scenario, *Peak* is the best among the policies. It has the lowest killed jobs (in number and size). *Next* has a good number of finished jobs, but a high number of killed jobs. Considering the size, it has the worst percentage of finished jobs. *Load* is still aggressive, finishing fewer jobs (in number). *Last* is a little better in this critical scenario, compared to the previous one. However, it still kills several jobs (second-worst among the policies).

Finally, considering the 100 random cases, *Last* finishes more jobs in number and size. *Peak* and *Load* are close, with *Next* being the worst one. Comparing killed jobs, *Last* and *Peak* killed fewer jobs than *Next* and *Load*. *Load* is too aggressive, and *Next* compensates too soon.

Considering all these results, we can say that:

1. *Last* policy is the best one when we have more power arriving. Letting the power in the end, helps to migrate a second time when needed. However, it can miss the opportunities to turn on more servers in demand peak;

2. *Peak* policy has a good overall result, independent of the power profile;

3. *Next* policy is good in a context where the production is lower, and the demand peak is on the first day. In this scenario, the SoC drops too fast, and *Next* adapts the usage sooner than the other policies;

4. *Load* policy is aggressive, which improves slowdown but can lead to several killed jobs.

Regarding wasted energy, *Next* is more likely to waste more than the other policies. It reintroduces the surplus as soon as possible, even if it is not necessary. *Last* uses the

energy wisely, since it places at the end, using before if necessary. *Peak* and *Load* depend on the scenario. Then, a new question comes up: is it possible to improve the Quality of Service in these scenarios, while still respecting the battery level at the end of the time window? To do so, the next chapter tries to mix the policies. For example, we can use *Load* for some positive compensations to improve QoS, then use *Last* for the remaining energy surplus. We proposed a model using reinforcement learning to find out which policy to use at each moment.

4.5 Conclusion

This chapter proposed new heuristics to approximate the battery storage level to the target level. These heuristics use surplus energy to improve the QoS, mainly the finished jobs. The policies try to reduce the impact on QoS in scenarios with less energy. The following chapter takes a step further, trying to mix the policies and improving QoS even more.

Chapter 5

Learning Power Compensations

Contents

5.1 Introduction

This chapter proposes the introduction of Reinforcement Learning (RL) to choose the best compensation policy. We previously saw that the heuristics proposed (Next, Peak, Last, and Workload) have good results compared to just following the plan. The best heuristic depends on the workload and power production. So, the idea is to let RL algorithms learn which are the best policies to use at each different moment inside the three-day time window. We expect that the global result will be good by improving decisions locally. The following sections will describe our approach for solving the compensation problem. First, we describe the model, detailing the states, actions, and rewards. Then, we define the algorithms used in this section to compare with the previous results. Finally, we present the results and a discussion.

5.2 Reinforcement Learning

As mentioned in Chapter 2, Reinforcement Learning (RL) performs a trial-and-error approach, where an agent explores an environment, takes actions, and receives feedback [84]. We chose RL algorithms, such as Q-Learning and Multi-Armed Bandit, because they are fast, have a simple implementation, and are lightweight compared, for example, to Deep neural networks [116]. So, a RL algorithm, for each time step t should evaluate the state S_t and define an action a_t. After applying the action a_t, it receives a reward r_t. These notations are used for our RL algorithms. We used two reinforcement learning algorithms in this thesis: Q-Learning and Contextual Multi-Armed Bandit. Section 5.6 presents them.

However, before presenting the algorithms, we describe our state, action, and rewards model. Our objective with RL is to define: *where to compensate the power difference (positive or negative) for each step within a three-day time window.* For example, the best actions in the early steps can be to put the power in the steps with a higher power deficit, but not too much to not dry the battery too fast. So, RL must find a balance between the actions. For ease the comprehension, this chapter uses the following terms:

- *Decision step*: The step that takes action to modify the power of a future step. It will receive a reward according to the impact of this action. It can impact only one future step;
- *Future step*: The step that receives more or less power from one or several decision steps. When this future step finishes, it is possible to give back a reward for the decision step(s);
- *Iterations*: During the learning process, an iteration is the re-execution of an experiment without changing the workload and production but using the previous knowledge.

5.3 States

In our problem, the state must englobe three aspects: *moment of the decision*, *how far we are from the plan*, and the *energy to compensate*. The *moment of the decision* is the decision step. For example, for a three-day time window divided into 5-minutes steps, we have an integer from 0 to 863. The idea of this aspect is that the best decisions depend on the moment inside the time window. For example, the best decision at the beginning of the time window can be different from the best decision at the end of the time window. For *how far we are from the plan* aspect, we calculate the difference between the actual and predicted state of charge at the moment of decision t:

$$\Delta SoC_t = SoC_t^{actual} - SoC_t^{plan} \tag{5.1}$$

ΔSoC_t indicates if the battery is close to the offline PDM plan (ΔSoC_t close to 0), has less energy than predicted ($\Delta SoC_t < 0$), or has more energy than predicted ($\Delta SoC_t > 0$). For example, if the battery has less energy than predicted, it is better to reduce the usage in the next steps. Finally, the *energy to compensate* is given by energy needed to compensate to make SoC_t^{plan} at the end of the time window equal to the target (given by Equation 3.31). This can be positive (we need to discharge the battery) or negative (we need to charge the battery). This variable can indicate, for example, that a big positive energy compensation must be done in the *Last* step, instead of *Load* (avoiding drying the battery too fast). So, our state is composed by:

- Decision step: Integer from 0 to 863;
- ΔSoC_t: Difference from the actual and planned SoC;
- Energy compensation: The power to compensate;

5.4 Actions

Since our problem is to define where to compensate the power, our actions are in which future step to compensate. Here, we have a big difference between RL and the previous

heuristics. Previously, at each decision step, we compensate the energy entirely. This means that one decision step can change several future steps. In RL, we will choose only one step, letting the remaining power to compensate in the future. Let the future step be t'. We define two ways to choose t'. The first way is similar to our heuristics from Chapter 4. In this way, RL has four actions: Next, Peak, Last, and Workload. Figure 4.1 shows these heuristics. The second way is to decide at which hour to compensate. This way gives more freedom to RL to define the best step. The hour action includes 72 possible actions (a three-day time window has 72 hours). Therefore, the RL algorithm takes a step inside the hour chosen. To specify exactly which step inside the hour, it applies the peak policy (takes the higher usage to negative compensation and lower usage to positive compensation). Another possibility would be to choose the exact step to compensate. However, this would increase the action space (864 possible actions), demanding higher learning time. So, we have the following actions:

- Heuristic: We choose next, peak, workload, or last;
- Hour: We choose the hour inside the time window.

5.5 Rewards

The reward is one of the most important elements in RL because it drives the learning process. We have defined two rewards linked to Quality of Service (QoS). We focus on the QoS to direct the actions to make better QoS decisions. Mainly, we consider started, finished, and killed jobs in our rewards. The two rewards are called *started jobs* and *finished jobs*.

Started Jobs Reward

The *started jobs* reward considers the number of started jobs. The reward given by an action is defined as:

- If the future step t' kills at least one job: -1 multiplied by the number of the jobs killed;
- If the sum of compensations on step t' is positive and step t' does not kill any job: 1 multiplied by the number of the jobs started;

Since we can have more than one decision step changing the future step t', we must distribute this reward between the decision steps. To do so, we spread the reward according to how much each decision step impacts the future step t'. Furthermore, we consider only the decision steps that helped to arrive at the reward. For example, when we have a negative reward, we take only the decision steps that reduce the energy at the step. For example, if we have only three decision steps that reduced the energy in the future step t' with -1500 Wh, -1000 Wh, and -500 Wh (sum of -3000 Wh), respectively, and the scheduler killed one job in the future step t', the reward (-1) will be:

- Decision step 1: $\frac{-1}{3000} \times 1500 = -0.5$
- Decision step 2: $\frac{-1}{3000} \times 1000 = -0.333333333$
- Decision step 3: $\frac{-1}{3000} \times 500 = -0.166666667$

Finished Jobs Reward

The second reward uses the number of finished jobs. This reward gives the reward for all decision steps that impacted the job. It is defined as:

- If a job is killed: -1;
- If a job is finished: 1;

We distribute the reward between the decision steps in the same way as the previous reward. The only difference here is that we consider all steps that the job passed by. For example, if a job is killed, we give a proportion of -1 for each decision step that decreases the power in the steps that the job was running, according to how much it impacted. The same is done for finished jobs. This approach aims to reinforce the decisions that help to finish jobs and discourage the ones that kill jobs. Since the environment can calculate the reward only after executing the future step (to know how many started, finished, and killed jobs in the step t'), the RL algorithm will not receive the reward right after the decisions. For example, in iteration 0, the RL algorithm does not have prior knowledge, choosing only random actions. We standardized the reward update at the end of the iteration, reducing the bias (e.g., giving a reward earlier for *Next* action than *Last* can make *Next* be chosen more times). So, at the end of iteration 0, we calculate all rewards for all actions in this iteration. Then, iteration 1 uses the knowledge from iteration 0 in the decision-making process, updating the reward at the end of iteration 1.

5.6 Algorithms

In this section, we present the algorithms used in this section. We compare the following algorithms with the baselines and heuristics from the previous chapter.

5.6.1 Random

Before describing the RL algorithms, we present two random algorithms. We used these random algorithms to verify the RL results. We expect the rewards from random actions to be equal to RL in the first steps, but, after some iterations, the RL must improve its reward. Since we have two possible actions (heuristic and hour), we proposed two random algorithms. The first is the *Random heuristic* which chooses a heuristic from Chapter 4 randomly (*Peak*, *Next*, *Last*, or *Load*). The second is named *Random step*, choosing a random step. Both random algorithms compensate only in one future step, in the same way as the RL.

5.6.2 Q-Learning

Q-Learning can be understood as a table where the rows are the states, the columns are the actions, and the values are the Q-values. Therefore, for each row (state) and column (action), a Q-value is calculated using the Bellman equation:

$$Q^{new}(S_t, a_t) = (1 - \alpha)\ Q(S_t, a_t) + \alpha(r_t + \gamma \max_a Q(S_{t+1}, a)) \tag{5.2}$$

Where:

- $Q^{new}(S_t, a_t)$: New Q-value;
- S_t: State;

- a_t: Action;
- r_t: Reward observed;
- $Q(S_t, a_t)$: Actual Q-value;
- α: Learning factor;
- γ: Discount factor;
- S_{t+1}: New state after taking action a_t at state S_t;
- $\max_a Q(S_{t+1}, a)$: Best expected future reward.

Q-Learning updates the Q-value using this equation for every action taken in a state. $Q(S_t, a_t)$ is the Q-Value before the action is taken. α indicates how the new information overrides the old information. $\alpha = 0$ makes the agent exploits prior knowledge exclusively, while $\alpha = 1$ makes the agent consider only the most recent information. γ determines the importance of the expected future reward. $\gamma = 0$ makes the agent ignore future rewards, while $\gamma = 1$ makes the agent makes it attempt for a long-term high reward. As presented before, we calculate the reward only at the end of the iteration. Therefore, at the end of each iteration, we use the Bellman equation to update the Q-values of all actions taken in this iteration. Since we updated the reward in the end, we know the state transition (from S_t to S_{t+1}).

A Q-Learning limitation is that both state and action must be discrete because having state or action as continuous would demand an infinite table. Hence, we simplified some of the state variables (action is discrete already). The Decision step variable is discrete (from 0 to 863), so there is no need to change it. Considering the Power compensation, we calculate the percentage of the power compensation according to the battery size, splitting the result into slices of 10% (from -100% to 100%). For example, if the compensation is 140000 Wh and the battery size is 800000 Wh, the compensation is 17.5% of the battery, which puts it in the slice between 10% and 19.99999%. Each slice has an index, discretizing this value. We did the same for the delta SoC variable (also from -100% to 100%).

We use the epsilon-greedy policy for the exploration-exploitation model (or $\epsilon-greedy$). We start with $\epsilon = 1$, reducing it for every new iteration. Using ϵ, we verify if we take the higher $Q(S_t, a_t)$ or a random action:

$$a_t = \begin{cases} \max Q(S_t, a_t), & with\ probability\ 1 - \epsilon \\ random\ a_t, & with\ probability\ \epsilon \end{cases} \tag{5.3}$$

So, iteration 0 has $\epsilon = 1$, choosing random actions for every state. At the end of every iteration, we reduce ϵ by 0.0125. For example, iteration 1 will have $\epsilon = 0.9875$, allowing it to use a little from prior knowledge. ϵ will be 0.975 in iteration 2, 0.9625 in iteration 3, and so on. We define the learning factor $\alpha = 0.1$ and discount factor $\gamma = 0.9$. These were the values with better results in our different tests.

5.6.3 Contextual Multi-Armed Bandit with LinUCB

While Q-Learning works with a table creating the relationship between states, actions, and rewards, Contextual Multi-Armed Bandit (especially using the LinUCB algorithm) tries to find a linear relation between them [117]. LinUCB (Linear Upper Confidence Bound) algorithm learns the correlation between the state (named context in Multi-Armed Bandit) and the reward using linear regression. Each arm (action) has its linear model of context

(state) and reward. This algorithm estimates the upper confidence bound, using the standard deviation of previous rewards. So, the algorithm chooses the arm (action) with higher upper confidence bound, taking the arm with the higher possible return. Figure 5.1 illustrates this behavior, even if action (arm) 3 has a higher average, it chooses action 2 because of the higher UCB.

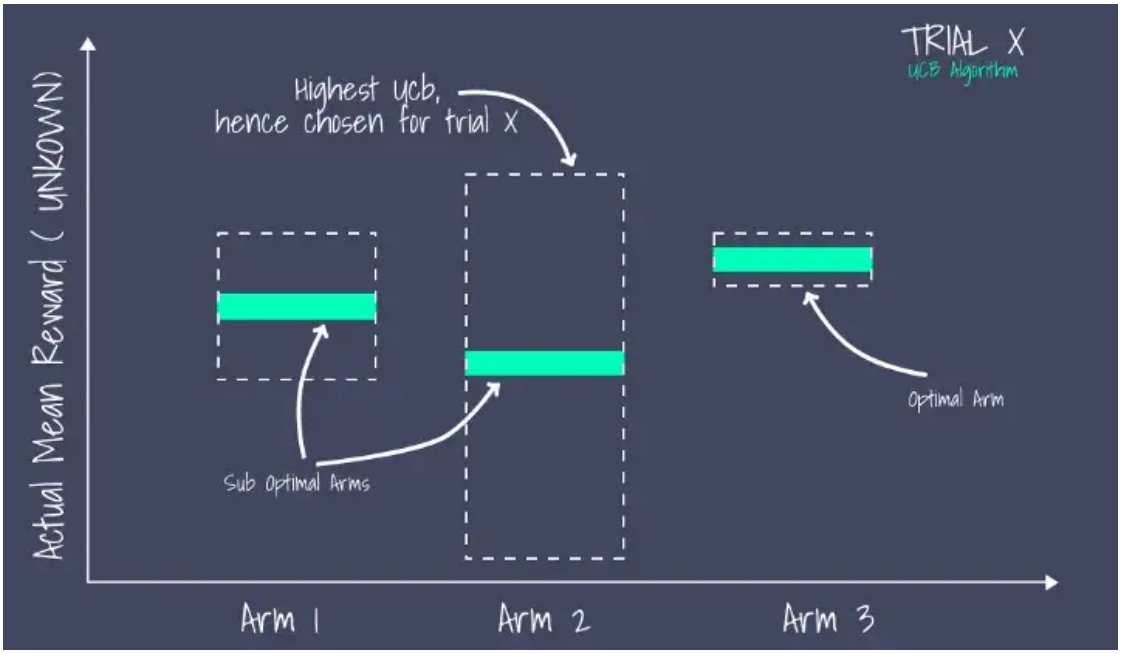

Figure 5.1: LinUCB algorithm for choosing the best arm [118].

We implemented the LinUCB algorithm from 2 [118]. Let d be the number of variables of the context $x_{t,a}$ (state). In our experiments, $d = 3$. A_a and b_a are the variables used in ridge regression. This regression tries to find the correlation between state and reward. Lines 2-5 initialize both A_a and b_a for all actions in the set of possible actions A_t. This initialization makes all arms (actions) start with a very high UCB, forcing each arm to be chosen at least once. So, it calculates the UCB for every arm at each step t (lines 6-10). To do so, line 8 calculates the ridge regression ($\hat{\theta_a}$), and line 9 applies it to find the UCB. The UCB $\rho_{t,a}$ is calculated by $\rho_{t,a} \leftarrow \hat{\theta_a}^\top x_{t,a} + \nu\sqrt{x_{t,a}^\top A_a^{-1} x_{t,a}}$ where the first part ($\hat{\theta_a}^\top x_{t,a}$) is the expected mean, and the second part ($\nu\sqrt{x_{t,a}^\top A_a^{-1} x_{t,a}}$) is the upper confidence bound. ν is a hyperparameter to indicate the importance of the standard deviation in the UCB. The higher ν is, the wider the confidence bounds become. So, a higher ν results in a higher emphasis on exploration instead of exploitation. We defined $\nu = 20$ after some experiments, with the best results with this value.

After that, line 11 verifies the action a_t with higher UCB among the actions in A_t. This line applies the action and receives a reward r_t. Then, it updates the ridge regression variables (A_a and b_a) for action a_t in lines 12 and 13 (A_{a_t} and b_{a_t}). Differently from Q-Learning, Contextual Multi-Armed Bandit does not need a discretization of the state. Therefore, we can use the state directly without modifications.

Algorithm 2: LinUCB algorithm [117].

begin
  forall $a \in A_t$ **do**
    $A_a \leftarrow I_d$ (d-dimensional identity matrix);
    $b_a \leftarrow 0_{dx1}$ (d-dimensional zero vector);
  end
  for t = *0, 1, 2, 3, ..., T* **do**
    forall $a \in A_t$ **do**
      $\hat{\theta_a} \leftarrow A_a^{-1} b_a$;
      $\rho_{t,a} \leftarrow \hat{\theta_a}^{\top} x_{t,a} + \nu\sqrt{x_{t,a}^{\top} A_a^{-1} x_{t,a}}$;
    end
    Choose arm $a_t = \arg\max_{a \in A_t} \rho_{t,a}$, and observe a real-valued reward r_t;
    $A_{a_t} \leftarrow A_{a_t} + x_{t,a_t} x_{t,a_t}^{\top}$;
    $b_{a_t} \leftarrow b_{a_t} + r_t x_{t,a_t}$;
  end
end

5.7 Results Evaluation

After presenting the algorithms, we apply them to the critical scenarios from the previous chapter. We have four different RL executions combining the RL algorithm and action types:

1. *Bandit + heuristic*;
2. *Q-Learning + heuristic*;
3. *Bandit + hour*;
4. *Q-Learning + hour*;

These algorithms use the same scheduling from Chapter 4, changing only in which step compensating. We run 200 iterations for each critical case of each RL algorithm with the different reward types. The idea is to verify if the RL algorithms can learn by repeating the same inputs (workload and weather). In addition, we want to verify if they can improve the number of finished jobs. We do not execute the random cases due to their complexity. There are 100 different cases, demanding high processing time for executing 200 iterations of each one. Again, the critical cases are:

1. *Critical 1*: Profile best-case and workload in the beginning;
2. *Critical 2*: Profile best-case and workload in the end;
3. *Critical 3*: Profile worst-case and workload in the beginning;
4. *Critical 4*: Profile worst-case and workload in the end;

The results from the baselines and compensation policies (*Last*, *Next*, *Peak*, and *Workload*) are the same results from the previous chapter. We compare them with the new results from Reinforcement Learning. The results of the random algorithms are an average of 200 iterations.

5.7.1 Started Jobs Reward

Beginning with the Started Jobs reward, we present the results in the four critical cases in Figures 5.2, 5.3, 5.4, and 5.5. Each figure shows five graphs. The first two on top are the graphs about the finished jobs. The two graphs in the middle are the battery level and wasted energy. The last graph is the reward evolution over the 200 iterations. The reward is calculated by summing all rewards from the different actions inside the iteration.

Figure 5.2: Results of reward started jobs in critical case 1.

Figure 5.2 shows that in critical case 1, all RL improved their reward over time. They started with similar rewards to random executions. However, at the end of 200 iterations, they improved the reward. This result shows that the algorithms find the decisions which give higher rewards. Considering the reward, the best algorithm is *Bandit + heuristic*. *Q-Learning + heuristic*, *Q-Learning + hour*, and *Bandit + hour* have very similar rewards, with *Bandit + hour* having more variance than the other two. However, considering the finished jobs in number (first graph), the results indicate *Q-Learning + hour* as the best one, with *Bandit + hour* as the second one with a similar result. So, even if *Bandit + heuristic* has the best reward, it does not mean it has the best global QoS. This result

Figure 5.3: Results of reward started jobs in critical case 2.

shows a problem in this reward, where just starting jobs does not guarantee they will finish. Additionally, the best RL result is still worst than the *Last* policy in the finished jobs metric (both in number and size). *Random step* has a similar result to the best RL algorithm, showing that even a random choice is a good option here. Considering the Battery level, all RL algorithms have the same results. Finally, both Bandit and Q-Learning algorithms expend more energy than *Last* policy in this case.

The results of critical case 2 in Figure 5.3 show that both RL algorithms using the action of the hour have the highest reward. However, it is not translated to good QoS. These algorithms have worse Finished jobs in number than *Q-Learning + heuristic*, *Random step*, *Random heuristic*, *Workload*, *Last*, and *Peak* policies (ignoring the baselines). *Q-Learning + heuristic* has the best number of finished jobs among the RL algorithms, with the third-best reward. However, it kills more jobs. *Bandit + Heuristic* has the worst QoS among the RL algorithms. Both random algorithms finished several jobs, but they also have a high number of killed jobs. Considering the size, *Q-Learning + hour* and *Q-Learning + heuristic* finished more jobs than the other RL algorithms and the random,

Figure 5.4: Results of reward started jobs in critical case 3.

but they are still far from *Workload* and *Last* policies. Regarding the battery level, all RL algorithms ended close to the target level. Finally, *Q-Learning + heuristic*, *Bandit + hour*, and *Q-Learning + hour* wasted less energy than the random algorithms but more than *Last* and *Workload* policies.

Starting the cases with less energy, Figure 5.4 illustrates the results of critical case 3. Here, again, all RL algorithms have higher rewards than both random algorithms. Again, the *Q-Learning + hour* and *Bandit + hour* have the best results among the RL algorithms. *Bandit + hour* has a lower reward but a higher number of jobs finished. However, all RL algorithms have fewer finished jobs than *Next* policy (in number and size). Besides, they kill more jobs than *Next* and *Peak* policies. These results show again that the RL could not learn which are the best policies to use. Considering the battery level, *Q-Learning + hour* and *Bandit + hour* use slightly more battery than the other policies. Nevertheless, they are all below the baselines. Finally, the wasted energy results of the RL algorithms are higher than the *Peak* and *Next* policies.

Figure 5.5 presents the last scenario, critical case 4. All RL algorithms have higher

Figure 5.5: Results of reward started jobs in critical case 4.

rewards than random heuristics. In this scenario, the reward varies a lot, showing that it still has indecision to define the best actions. *Bandit + hour* has the highest number of finished jobs among the RL algorithms. However, it kills more than the other RL algorithms. No RL algorithm is better than the *Peak* policy in finished and killed jobs in both number and size (ignoring the baselines). All RL algorithms finished with the battery level close to the target. Finally, only *Bandit + hour* wasted less energy than *Workload* (the best policy).

Consolidating the results of the Started Jobs Reward, Table 5.1 presents a ranking of the different tested scenarios. We highlight in green the top 3 results on each metric for each scenario. The bottom 3 results are in red. Both finished and killed jobs are considered in number. Killed jobs are: killed jobs + reach the walltime + not completely finished. For SoC, we assume the best results as the higher real SoC at the end of the time window. The learning algorithms could not improve the metrics, compared to the reactive algorithms and the policies from Chapter 4. In the two scenarios with profile best-case, *Last* has the best results, with no metric at bottom 3 and 7 metrics at top

3. On the other hand, in profile worst-case executions, *Next* and *Peak* are quite good. The RL algorithms have some results in top 3, but they generally stay out of the top 3 results. Regarding the reward, it is possible to notice that *Q-Learning + hour* and *Bandit + hour* improved the reward after 200 iterations. Usually, the reward from these cases has a distance from the random rewards after 200 iterations. This result indicates that the RL learning algorithms are learning the policies to choose the actions which give a high reward. However, having a high reward does not result in good global results (mainly finished/killed jobs). This reward reinforces the actions which put more energy into steps that start more jobs. However, starting jobs does not mean that they will finish. So, it can start several jobs but kill a lot also. Aiming to solve this problem (starting some jobs but not finishing them), we proposed the next reward.

Table 5.1: Consolidate average results in every scenario for reward started.

Scenario	Metric	Follow plan	Power reactive	Workload reactive	Peak	Next	Last	Load	Rand. heuris.	Rand. step	Bandit + heuristic	Q-Learn. + heuristic	Bandit + hour	Q-Learn. + hour
Profile best-case and workload in beginning	Finished jobs	13th	12th	11th	7th	10th	1st	5th	6th	3rd	8th	9th	4th	2nd
	Killed jobs	13th	12th	11th	7th	10th	1st	5th	6th	3rd	8th	9th	4th	2nd
	SoC	1st	13th	2nd	9th	12th	3rd	7th	4th	5th	11th	8th	10th	6th
	Wasted energy	11th	13th	1st	7th	10th	2nd	8th	6th	4th	12th	9th	5th	3rd
Profile best-case and workload in end	Finished jobs	11th	4th	1st	7th	13th	2nd	6th	5th	8th	12th	3rd	10th	9th
	Killed jobs	13th	6th	1st	8th	9th	3rd	2nd	10th	12th	7th	11th	4th	5th
	SoC	1st	13th	12th	4th	5th	6th	7th	3rd	8th	10th	2nd	11th	9th
	Wasted energy	4th	13th	1st	11th	12th	2nd	3rd	8th	9th	10th	6th	7th	5th
Profile worst-case and workload in beginning	Finished jobs	2nd	1st	4th	9th	3rd	11th	13th	10th	8th	12th	7th	5th	6th
	Killed jobs	12th	10th	13th	2nd	1st	4th	9th	6th	11th	5th	3rd	8th	7th
	SoC	11th	13th	12th	3rd	1st	4th	7th	6th	8th	5th	2nd	10th	9th
	Wasted energy	12th	13th	11th	2nd	1st	5th	10th	4th	7th	6th	3rd	9th	8th
Profile worst-case and workload in end	Finished jobs	1st	3rd	2nd	4th	7th	9th	12th	10th	11th	8th	6th	5th	13th
	Killed jobs	12th	2nd	6th	1st	9th	4th	3rd	10th	13th	5th	8th	11th	7th
	SoC	11th	12th	13th	1st	3rd	10th	4th	9th	8th	6th	5th	2nd	7th
	Wasted energy	12th	13th	11th	5th	10th	6th	2nd	4th	9th	8th	3rd	1st	7th

5.7.2 Finished Jobs Reward

The next reward aims to solve the problem of starting jobs and not finishing them. So, this reward reinforces the actions that help to finish more jobs. Figures 5.6, 5.8, 5.10, and 5.12 present the results of this reward.

Figure 5.6: Results of finished jobs reward in critical case 1.

We introduce in this section a new graph in Figures 5.7, 5.9, 5.11, and 5.13. In these figures, the graphs on the top are from finished jobs, where blue+red means the total number of finished jobs, blue is the finished jobs included in the reward, and red is the finished jobs ignored in the reward. The higher the red+blue, the better. On the other hand, the bottom graphs are the killed jobs, where blue+red means the total number of killed jobs, blue is the jobs included in the reward, and red is the jobs finished but ignored by the reward. The higher the red+blue, the worse. The idea in these graphs is to see if, over the iterations (x-axis), the global finished jobs increase and the killed jobs reduce. Besides, these graphs illustrate if the reward reflects our objective of improving the global QoS (by improving locally). A job is considered ignored in the reward when no previous step impacts the power usage of the steps where it executes. For example, let's say a job

executes from steps 50 to 52. A job is ignored in the reward if no previous step increases the power usage in steps 50, 51, and 52. The same for killed jobs.

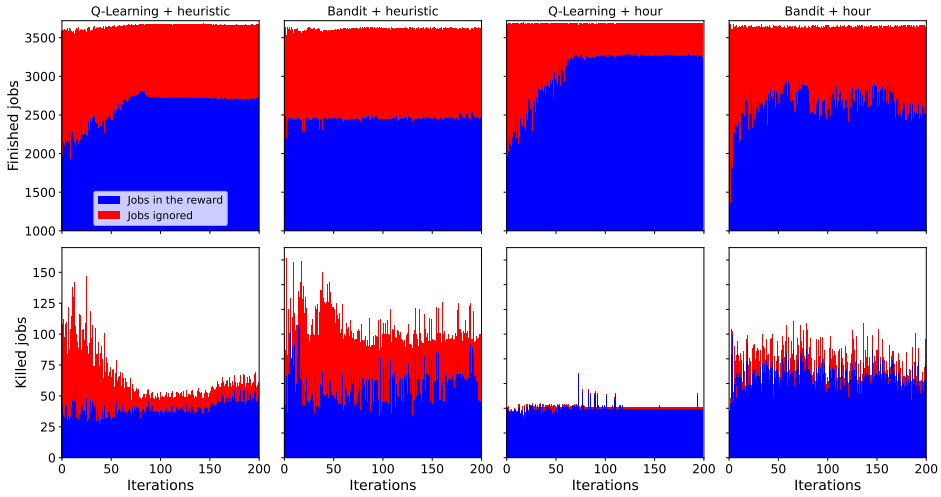

Figure 5.7: The jobs (finished and killed) included in the reward in critical case 1.

Starting with critical case 1, Figure 5.6 illustrates the results. The highest reward is from *Q-Learning + hour*, which has the highest finished jobs in number and size among the RL algorithms. The second-best reward is *Q-Learning + heuristic*, having also the second-best finished jobs (in number and size). However, we see a large difference between both rewards, which does not appear in the final finished jobs. Figure 5.7 highlights the problem. It is possible to notice that the reward's improvement of *Q-Learning + hour* is not linked to more jobs finished, but due to more finished jobs included its reward. So, it puts energy into steps with more finished jobs just to receive the reward of them. *Q-Learning + heuristic* "touches" fewer finished jobs, even finishing quite the same number of jobs. *Q-Learning + heuristic*, *Bandit + heuristic*, and *Bandit + hour* do not increase the number of jobs finished, comparing the first iterations to the last ones, having constant finished jobs. Therefore, they do not improve the global finished jobs in this execution. *Q-Learning + heuristic* is the only algorithm improving the global QoS comparing first iterations to the last ones (finished and killed jobs). Regarding the killed jobs, *Q-Learning + hour* kills fewer jobs and almost every killed job is inside the reward. On the other hand, *Q-Learning + heuristic* kills more jobs and has some killed jobs not included in the reward. Both Bandit algorithms have higher killed jobs and high reward variance, even in the last iterations. Going back to the results, *Q-Learning + hour* presents quite the same result as the *Last* policy in the number of finished jobs and lower finished jobs in size. Considering the battery, all RL algorithms are close to the target battery level. So, compared with *Last*, the RL algorithms use more energy from the battery but not finishing more jobs. The wasted energy metric shows that the RL wasted more energy than the *Last* policy. Hence, the more energy used by RL algorithms is wasted. We can observe that even doing a random algorithm is a good option in this scenario. *Random step* has good finished jobs (in number), finishing more jobs than three of four RL algorithms. Furthermore, *Random step* has the lowest wasted energy between random and RL algorithms.

Figure 5.8 shows the results of critical case 2. In this scenario, *Q-Learning + hour* has the highest reward, with *Bandit + hour* being the second best. However, *Q-Learning + heuristic* has the best number of finished jobs and killed jobs among the RL algorithms. Figure 5.9 shows that *Q-Learning + hour* and *Bandit + hour* touched more finished jobs

Figure 5.8: Results of finished jobs reward in critical case 2.

than *Q-Learning + heuristic*. Besides, *Bandit + hour* touched less killed jobs. This algorithm "learns" to avoid some steps with killed jobs. So, it results in a higher reward, even with a lower QoS. Another important aspect is that *Q-Learning + hour* reduces the number of finished jobs, comparing the first iterations to the last ones. This result indicates that this algorithm prefers to "touch" more finished jobs than finish more globally. Here, we can see that improving locally the reward does not mean that we will improve it globally. The same happened to *Q-Learning + heuristic*. Going back to Figure 5.8, all RL algorithms are worst than *Workload* and *Last*, considering both finished and killed jobs. Even *Random heuristic* is better than RL algorithms in finished jobs (in number), but it has higher killed jobs. Considering the battery level, the RL algorithm results are close to the target level. Finally, they wasted more energy than *Last* and *Workload* policies.

Starting the cases with less energy, Figure 5.10 shows the results of critical case 3. *Q-Learning + hour* has the best reward but is not the best algorithm in finished/killed jobs metrics. The best result comes from *Bandit + heuristic*, with the highest number of finished jobs and the lowest number of killed jobs among the RL algorithms. However,

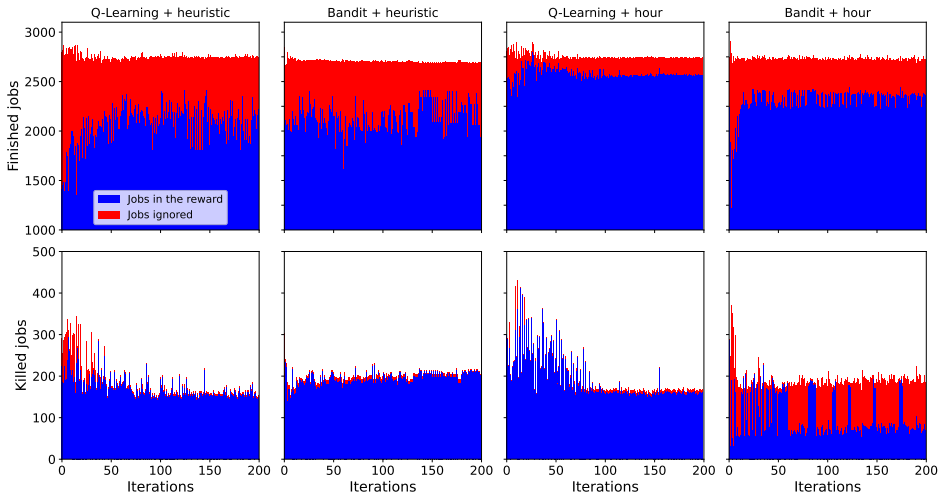

Figure 5.9: The jobs (finished and killed) included in the reward in critical case 2.

Bandit + heuristic has the lowest reward. Figure 5.11 shows a higher global number of finished jobs and lower killed jobs in *Bandit + heuristic* compared to the other algorithms. So, we can see that doing good local decisions does not mean good global results in our problem. Both *Q-Learning + hour* touched more jobs than *Bandit + heuristic*, increasing their reward. However, *Q-Learning + hour* increases the number of touched finished jobs, which leads to a reduction in the global finished jobs. At least, this algorithm arrives to reduce the number of killed jobs in this scenario. Coming back to Figure 5.10, no RL is better than *Next* policy in number of finished jobs and better than *Peak* policy in size of finished jobs. Both *Next* and *Peak* are still the executions with the lowest killed jobs. Both random algorithms kill several jobs. So, random is not an option either. Considering the battery level, all RL algorithm results are close to the policies. Regarding wasted energy, all RL algorithms wasted more than *Peak* and *Next* policies. This scenario shows that even including QoS decisions in the algorithm, a renewable-only data center demands better battery management, like *Next* policy. Maintaining the QoS above SoC_{min} is the most important constraint in a scenario with less energy.

Finally, the last results of critical case 4 are presented in Figure 5.12. This scenario also has less energy arriving. We can see that *Q-Learning + hour* has the highest reward. It also has the second-best finished jobs among the RL. Again, the highest number of finished jobs come from the RL with not the highest reward, *Q-Learning + heuristic*. Figure 5.13 illustrates that *Q-Learning + heuristic* touches fewer jobs than *Q-Learning + hour*, but finishes more. This shows again that the best local decisions (see the reward obtained by *Q-Learning + hour*) do not improve the global QoS. In this case, *Q-Learning + hour* improves more jobs but increases the number of killed jobs. It avoids the steps with these killed jobs, increasing its reward. However, both *Q-Learning + hour* and *Bandit + hour* do not stabilize the number of killed jobs, having high variation even in later iterations. Returning to Figure 5.12, *Q-Learning + heuristic* has the best finished jobs considering the size, even better than the policies. However, it also kills more than *Peak* and *Workload*. Considering the battery level, again, all the RL algorithms are close to the target level. Finally, the *Q-Learning + heuristic* presents the lowest wasted energy. This result is linked to the finished jobs considering the size, where this algorithm finished more than all policies. So, this algorithm arrives to maintain more big jobs running for longer, using the energy well.

Figure 5.10: Results of finished jobs reward in critical case 3.

Similarly to the previous reward, Table 5.2 presents the final metrics of the algorithms. Again, the RL algorithms do not improve the results, having just some metrics at the top 3 and some at the bottom 3. The results majority of RL algorithms stay out of the top 3 and bottom 3 ranking. After presenting all results, Section 5.7.3 discusses them more generally, indicating the problems of our RL model and proposing future possibilities in machine learning.

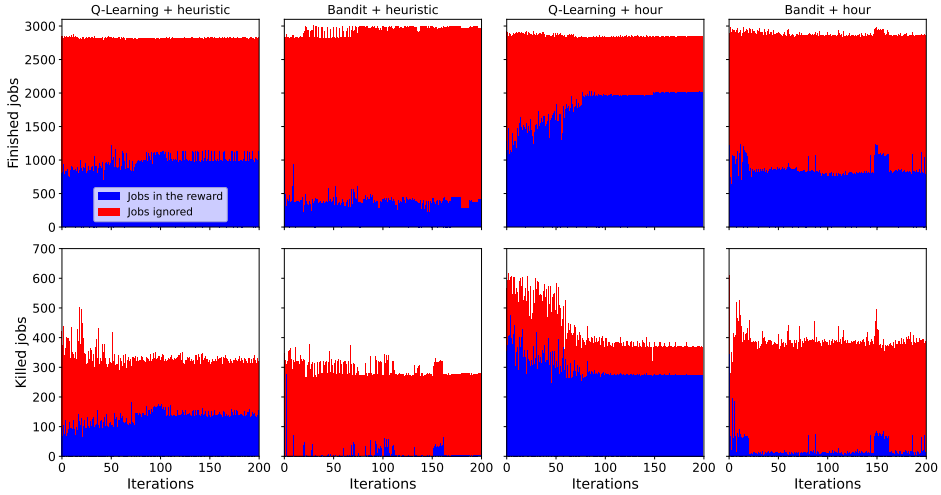

Figure 5.11: The jobs (finished and killed) included in the reward in critical case 3.

Figure 5.12: Results of finished jobs reward in critical case 4.

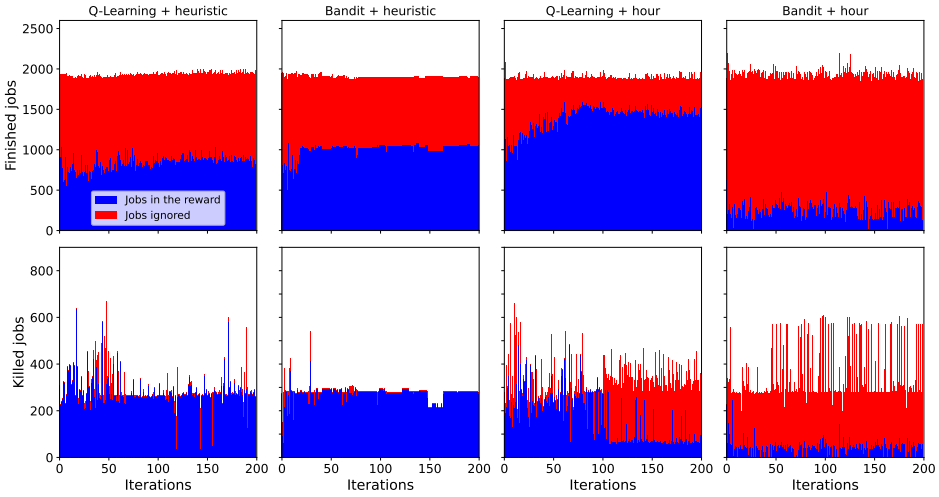

Figure 5.13: The jobs (finished and killed) included in the reward in critical case 4.

Table 5.2: Consolidate average results in every scenario for reward finished.

Scenario	Metric	Follow plan	Power reactive	Workload reactive	Peak	Next	Last	Load	Rand. heur.	Rand. step	Bandit + heuristic	Q-Learn. + heuristic	Bandit + hour	Q-Learn. + hour
Profile best-case and workload in beginning	Finished jobs	13th	12th	11th	9th	10th	1st	5th	8th	3rd	7th	4th	6th	2nd
	Killed jobs	13th	12th	11th	9th	10th	1st	5th	8th	3rd	7th	4th	6th	2nd
	SoC	1st	13th	2nd	10th	12th	3rd	7th	4th	5th	11th	8th	9th	6th
	Wasted energy	12th	13th	1st	7th	10th	2nd	8th	6th	3rd	11th	5th	9th	4th
Profile best-case and workload in end	Finished jobs	12th	3rd	1st	7th	13th	2nd	5th	4th	10th	11th	6th	9th	8th
	Killed jobs	13th	7th	1st	9th	10th	5th	3rd	11th	12th	8th	2nd	6th	4th
	SoC	1st	13th	12th	3rd	4th	6th	7th	2nd	9th	8th	10th	5th	11th
	Wasted energy	5th	13th	1st	11th	12th	2nd	3rd	8th	10th	9th	4th	7th	6th
Profile worst-case and workload in beginning	Finished jobs	2nd	1st	5th	8th	3rd	11th	13th	10th	6th	4th	12th	7th	9th
	Killed jobs	12th	10th	13th	2nd	1st	5th	9th	8th	11th	3rd	4th	7th	6th
	SoC	11th	13th	12th	3rd	1st	4th	7th	6th	9th	2nd	5th	10th	8th
	Wasted energy	12th	13th	11th	2nd	1st	6th	10th	4th	9th	3rd	5th	8th	7th
Profile worst-case and workload in end	Finished jobs	1st	3rd	2nd	4th	6th	8th	12th	9th	11th	10th	5th	13th	7th
	Killed jobs	11th	2nd	7th	1st	8th	5th	3rd	9th	12th	6th	4th	13th	10th
	SoC	11th	12th	13th	1st	2nd	10th	4th	7th	6th	5th	8th	9th	3rd
	Wasted energy	12th	13th	11th	4th	10th	5th	2nd	3rd	7th	6th	1st	8th	9th

5.7.3 Discussion

After presenting and briefly discussing the results of the Reinforcement Learning algorithms in critical scenarios, this section analyzes the RL proposal deeper. We chose the critical scenarios to start with because they have more room for improvement than the random cases. The idea is that if the RL could improve in these scenarios, we could take a step further and apply RL in a scenario with more uncertainty. We proposed 200 iterations in each critical scenario, letting the RL test the different approaches to deal with them. However, even in this simplified test, our RL model was not sufficient to improve the QoS. In every case, at least one policy is better than the RL algorithms. Tables 5.1 and 5.2 show that RL did not improve the final metrics, even after several iterations. Several reasons help to explain these results. First, we verified a difference between Bandit and Q-Learning algorithms learning. Usually, Q-Learning has a slower learning curve, going from bad rewards in the beginning but increasing over the iterations. On the other hand, Bandit does not have this learning curve. The reason is linked to the algorithm idea. While Q-Learning has a different Q-value for each state, Bandit defines a unique linear function for all states. For example, let's consider only the step variable as the state. Q-Learning can define that the best action in step 1 is *Next*, in step 2 is *Last*, and in step 3 is *Next* again. On the other hand, Bandit defines a linear function, making it harder to do the same variation. However, while Q-Learning needs several iterations to try all action possibilities in all states, Bandit has the linear function for all states in the first iteration. Therefore, Q-Learning is slower to learn but can find the best action for each state, while Bandit is faster but must have a linear relation between state and reward.

Our proposed RL model also introduces some problems. The three concepts (state, action, and reward) present issues. Starting with the reward, it is possible to verify in the experiments that our reward does not represent well our environment. More specifically, it rewards the decisions considering the local impact (finish more and kill less) and does not include the global QoS. This behavior introduces two problems. First, since we share the reward of finishing/killing jobs among all the steps which impacted the jobs, the RL algorithm prefers to modify steps with fewer modifications from other steps. For example, choosing *Next* action would give a higher reward to the step because it receives all the reward from the step without sharing it with the other steps. The second problem is that our model only considers the jobs impacted by the actions. So, RL finds ways to impact more jobs, even with no need to receive more rewards. However, this behavior can lead to QoS degradation, such as in Figure 5.11 of *Q-Learning + hour* algorithm. This algorithm increases the reward but reduces the total finished jobs. Aiming to solve this reward problem, we could change the reward model to use the global finished jobs for all actions. Nevertheless, this new reward would demand a higher learning time (more iterations) to try all combinations of actions, which can be impeditive in an online system.

Regarding the action, we tested two actions: heuristic and hour. We noticed that the hour action has more freedom to choose the best action. Usually, *Q-Learning + hour* is among the best results. We could not use the compensation step as action since it increases the action space, demanding higher learning time (more iterations). Future work can introduce a more flexible action space.

Finally, the last problem of our model is the state space. We defined a simplified state space to reduce the learning time. However, this state space could include some QoS metrics, such as queue length, slowdown, waiting time, etc. Another big problem that the state could solve is related to the SoC. As mentioned before, the scheduler kills the jobs when the SoC arrives at SoC_{min}. So, it chooses future steps without knowing the impact of the action on the SoC. It would be interesting to introduce the expected SoC time series

in the state. However, time series has too much information for the RL state, demanding a migration to a more complex algorithm, such as Deep Reinforcement Learning. This algorithm introduces a neural network to find the relation between a complex state to the expected reward. However, this kind of algorithm demands more learning time and more computational effort to maintain. Since it is heavier than a simple RL, this kind of algorithm also expends more energy, which is the opposite of what we look for.

Another problem that could be solved by changing the learning algorithm is decision dependency. Decision dependency means that the actions are chained. So, the best actions in the last steps depend on the first steps actions. These chained decisions, linked to the global reward, turn this problem really hard to solve using a simple RL algorithm. Additionally, we still maintain scheduling and compensation divided. So, the scheduling can start new jobs even if the SoC is in a dangerous place or if the battery does not have the energy to maintain this job running until the end. A new algorithm should mix compensations and scheduling decisions, improving the decisions. Due to the complexity of all these elements, our next approach will be a new heuristic that mixes all these aspects. Chapter 6 presents this heuristic.

5.8 Conclusion

This chapter presented a new model for solving the compensation problem using Reinforcement Learning algorithms. We presented two algorithms: Bandit and Q-Learning. The algorithms analyze the state, trying to find the best compensation steps. We proposed two rewards linked to QoS. Then, we presented the results and a discussion. The results show that our problem could not be solved by a simple RL algorithm due to the different constraints. Therefore, the next chapter presents a new heuristic, which mixes compensations and scheduling decisions.

Chapter 6

Adding Battery Awareness in EASY Backfilling

Contents

6.1 Introduction

After trying to introduce reinforcement learning in our model to choose the best compensation policy, this chapter describes a heuristic that englobes several elements to make better decisions. We named this heuristic *BEASY*. This heuristic considers the power production and demand to find the best moment to compensate. In addition, *BEASY* mixes the compensations with the scheduling decisions. We present the algorithm, followed by the results compared with the previous heuristics.

6.2 BEASY

This heuristic acts in three different moments. First, Section 6.2.1 explains the predictions used through *BEASY*'s decisions. These predictions are made at the beginning of the time window, just one time. Then, Section 6.2.2 describes the modifications in the EASY Backfilling heuristic to introduce battery awareness. The modified EASY Backfilling acts every time a job finishes, arrives, or new servers are available. Finally, Section 6.2.3 defines the compensation policies. *BEASY* compensates every time step.

6.2.1 Predictions

As presented in Section 3.2.2, ODM receives two predictions from offline modules: power production and power demand. Since ODM works online, it will not predict itself but use the predictions from offline. So, this section will not focus on the forecasting method but on using its results.

Figure 6.1 illustrates both forecasts showing the area of uncertainty. The real value can be any value inside the uncertainty area. *BEASY* uses these predictions to create

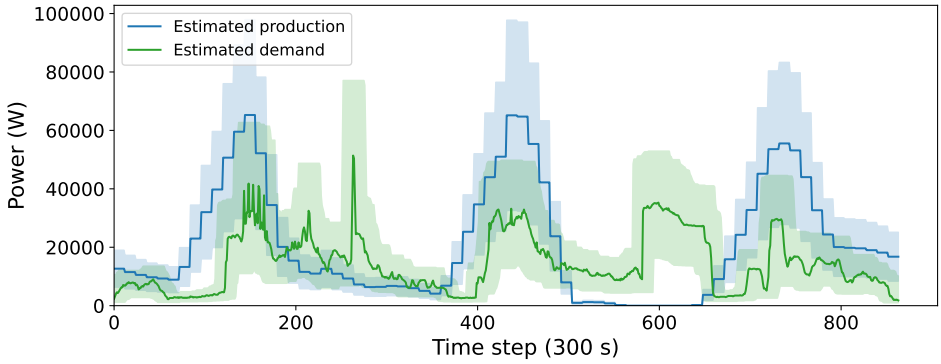

Figure 6.1: Renewable production and demand prediction. The blue (production) and green (demand) areas are the uncertainty given by the forecast.

different possible states of charge using equations 2.3 and 2.4. To do so, we estimated P_{dch} and P_{ch} using Equations 6.1, and 6.2:

$$P_{ch}(t) = \begin{cases} P^{est}_{renew} - P^{est}_{load}, & \text{if } P^{est}_{renew} > P^{est}_{load} \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

$$P_{dch}(t) = \begin{cases} P^{est}_{load} - P^{est}_{renew}, & \text{if } P^{est}_{renew} < P^{est}_{load} \\ 0, & \text{otherwise} \end{cases} \tag{6.2}$$

Where:

- P^{est}_{renew}: Estimated power production from renewable;
- P^{est}_{load}: Estimated power demanded.

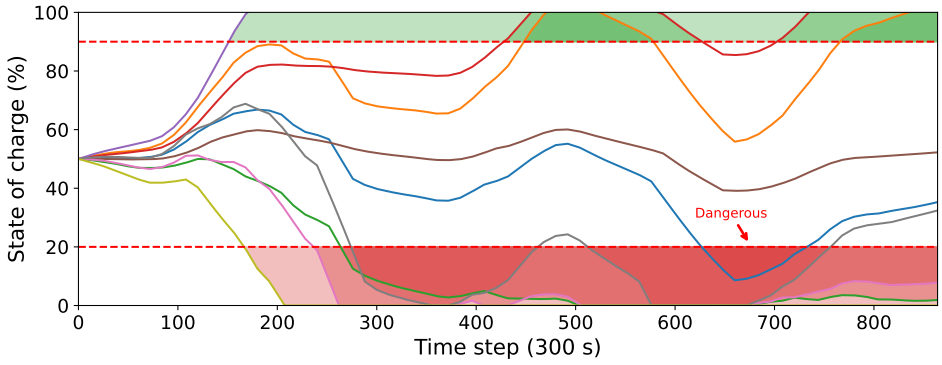

Figure 6.2: Result of the Equation 2.4 for different predictions. The dangerous area is when 5 or more curves (so, more than half of them) are below 20%.

Figure 6.2 demonstrates the result of applying Equation 2.4 using nine different predictions. *BEASY* calculates these SoC combining lower, median, and upper boundaries from the area presented in Figure 6.1 (e.g., demand lower boundary + production lower boundary, demand median + production lower boundary, demand median + production higher boundary, etc...). It is possible to notice that the state of charge can vary a lot. Section 6.2.3 will describe how we use these SoCs to compensate. Figure 6.2 also illustrates both SoC upper and lower thresholds (red dashed lines). Setting upper and lower

thresholds helps to increase the battery lifetime [25]. The narrower the range, the longer the expected lifetime [25]. However, selecting a narrow range limits the battery benefits. The figure presents both thresholds as 90-20%, but they are parameterizable. Finally, *BEASY* estimates dangerous areas in the time window. Figure 6.2 indicates this moment. It considers the dangerous areas when more than half of the predicted SoC curves are below the lower threshold. Taking Figure 6.2 example with nine predictions, *BEASY* considers the dangerous point where five curves are below 20%. Section 6.2.2 will explain how it uses these moments to make better scheduling decisions.

6.2.2 Job Scheduling

One of the most important ODM's duties is placing jobs on servers. To do so, *BEASY* implements EASY backfilling using two different sorts [114, 115]. We detail how we use both sorts in this section. Algorithm 3 presents the main idea. This algorithm is similar to Algorithm 1 but with some modifications (we highlighted them). *BEASY* runs this algorithm when a job arrives, finishes, or new servers are available. First, this heuristic sorts the jobs in the queue in a priority order P_R (line 2). Then, it finds the servers to run this job (line 5). The server must be available at least at the actual time step to be chosen. Line 6 has the first modification. Usually, EASY backfilling only verifies if the server's S is available now. We added the following verifications:

1. It verifies if the server's S is available during the entire execution, considering the walltime given by the user as the execution time. If so, it returns true. If not, it goes to the next verification;

2. It verifies if it is possible to change the plan to keep the servers S running the entire execution. To do so, it does the following steps:

   (a) First, it calculates how much energy is needed. Let's say Ed_t is the energy demanded in step t without modification and $Ed_t^{'}$ is with modification. So, it calculates $Ed_t^{'}$ for each time step t that the server sleeps, putting the server in the same state/speed as the previous time step $(t-1)$. The total energy demanded is $\sum Ed_t^{'} - Ed_t$ considering all the time steps that the job executes;

   (b) Then, it calculates how much energy is possible to take from future steps, putting idle servers to sleep. Let it be E_{poss}. Since we need to maintain the state of charge between both thresholds, we can not "migrate" all the energy to use now. So, it only considers the idle servers from the actual time step until the time step where the SoC will be equal or lower to the lower threshold. We can migrate the energy freely between the actual time step and this future one. Figure 6.3 illustrates this verification;

   (c) Then, it tests if $E_{poss} >= \sum Ed_t^{'} - Ed_t$. If this is false, it returns false and does not change the plan. If this is true, it makes $Ed_t = Ed_t^{'}$, changes the server speeds, and recalculates the planned state of charge (using Equation 2.4).

These verifications do not increase the complexity. Verification 1 goes through the plan with limited size (e.g., in our three-day time window, we have a plan with 864 time steps). Verification 2-a is done together with verification 1. Verification 2-b is faster than the others since it can process fewer steps and stop when $E_{poss} >= \sum Ed_t^{'} - Ed_t$. Verification 3-c is just to apply the modifications. It recalculates the state of charge to maintain the SoC updated for the next jobs to schedule. So, if a job puts a future state of charge close

Algorithm 3: *BEASY* scheduling. Modified from [115].

```
   input  : Queue Q of waiting jobs, P_R as priority order, and P_B as backfilling order.
   output: None (calls to Start())
 1 begin
 2     Sort Q according to P_R;
 3     for job j in Q do
 4         Pop j from Q;
 5         S ← select_servers(j);
 6         if j can be started and finished in servers S then
 7             Start(j, S);
 8         else
 9             Reserve j at the earliest time possible according to the walltime of the currently
                running jobs;
10             Sort Q according to P_B;
11             for job j' in Q do
12                 S ← select_servers(j');
13                 if j' can be started and finished in servers S without delaying the reservation
                    on j then
14                     Start(j', S);
15                 end
16             end
17             break;
18         end
19     end
20 end
```

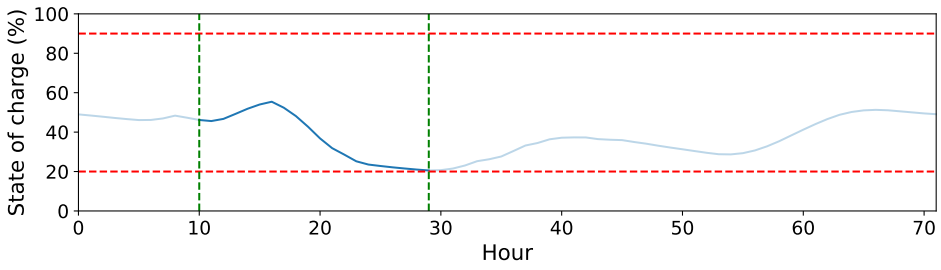

Figure 6.3: Verification of possible energy to save. In this example, the actual step is at hour 10. In this step, it needs to verify how much energy is possible to save from future steps. So, it verifies the idle servers from hour 10 to hour 29, because at hour 30 the state of charge is equal to 20%. It can change the usage from hour 10 to hour 29 freely. Taking energy from after hour 30 could violate the lower threshold since we will use more energy from the batteries.

to the lower threshold, the next job takes it into account. Continuing in algorithm 3, if the tests pass, then it starts the job (line 7). When it finds a job that can not be placed now, the algorithm does the backfill process (lines 9-17). Then, it finds the first moment to run this job (named priority job) in the future (line 9). So, it re-sorts the queue using P_B (line 10), placing the other jobs in the servers (lines 11-16) without delaying the (future) priority job execution (line 13). Line 13 does the same verification as line 6.

As mentioned before, EASY backfilling sorts the jobs by P_R and P_B. Our implementation starts with P_R using Bounded Slowdown from Equation 3.26. Bounded Slowdown estimates the ratio between the total time a job stays in the system and its actual processing time. This order helps to let a job wait proportionately to its size. For P_B, *BEASY*

sorts the waiting queue by the smallest sizes first (walltime multiplied by the number of resources needed). This order helps in the backfill process since sometimes the "holes" in the scheduling demand very small jobs. Besides, these jobs are less likely to demand more energy from future time steps (*BEASY* lets the energy to the priority ones). Figure 6.2 highlights a dangerous area. In this area, *BEASY* changes P_R to also use the smallest sizes first. These are not good moments to start big jobs, even if they are waiting too long in the queue. Small jobs demand less energy and are more likely to finish.

6.2.3 Power compensation

After describing the scheduling algorithm, this section explains the heuristic to compensate for power fluctuations. While the scheduling algorithm runs for every job arrival, end, or server state modification, the power compensation algorithm will execute at every new time step. Since the scheduling algorithm modifies future time steps (it places the jobs in servers that are already on) and verifies the violations, we do not need to run the power compensations for every placement. In addition, changing the server state too much between on and off can degrade it faster and takes time to power on/off. So, we defined that the state and speed stay constant inside each time step.

The main objective of this part of the heuristic is to finish the time window with the state of charge as close as possible to the planned. Renewable sources can produce more or less than predicted. Furthermore, the power usage can vary due to server idleness or scheduling modifications. So, at each time step, *BEASY* calculates the state of charge for all future time steps using Equation 2.4. Then, it calculates the energy difference E_{comp} between the target and the estimated SoC at the end of the time window using Equations 3.30 and 3.31. *BEASY* needs to reintroduce/remove the energy E_{comp} before the end of the time window.

When the compensation is positive ($E_{comp} > 0$), we can increase the speed of the servers or run more jobs. First, *BEASY* uses the E_{comp} to speed up the running jobs. It goes improving from the actual step until the last step. In each step, *BEASY* increases the processor's speed of the running jobs on this step. So, it tends to put more energy into the jobs right now than in the future. This behavior helps in avoiding jobs to reach their walltime. After that, if there is still energy, it verifies if there are jobs in the waiting queue. If so, it turns on some servers to run these jobs. If there is not or it turned all the servers needed to run jobs, it lets the remaining energy in the battery. This is a conservative approach. *BEASY* could be aggressive, using the remaining energy to turn on machines in the future. However, we prefer to finish with more energy in the batteries than expend this energy not wisely.

In the negative compensation ($E_{comp} < 0$), *BEASY* considers the estimated SoCs from Figure 6.2. First, it finds the time step with the higher number of predictions below 20% or the last time step if there are no predictions below 20% (let's name it the violation time step). The idea is to reduce the usage before the violation, reducing the violation probability. Then, *BEASY* reduces servers speed in the following order (stopping when it is enough):

1. Impacts idle servers from the violation time step to the actual time step (it goes through the time steps backward);

2. Impacts idle servers from the violation time step to the last time step (it goes through the time steps forward);

3. Impacts running servers from the violation time step to the last time step (it goes through the time steps forward);

4. Impacts running servers from the violation time step to the actual time step (it goes through the time steps backward);

BEASY focuses first on idle servers because impacting running servers can increase the number of killed jobs. Killing jobs increases wasted energy. So, *BEASY* searches for idle servers in both ways (violation time step → actual time step and violation time step → last time step). If reducing the usage from idle servers is insufficient, we start to impact running servers (steps 3 and 4). Our idea is to impact them as far as possible from the actual step, considering the violation step. The real total job execution time is uncertain (e.g., they could finish earlier than predicted). If we change the order (step 4 before step 3), the chance of really impacting the job is higher since it will reduce the energy from the violation step to the actual step. Doing step 3 before, we expect that the job finishes before these changes, while impacting the steps around the violation step.

6.3 Results Evaluation

After presenting *BEASY*, we compare its results with the algorithms presented previously. We execute the same critical cases and 100 random cases presented in Section 4.3. Just to remember, the critical cases are:

1. *Critical 1*: Profile best-case and workload in the beginning;

2. *Critical 2*: Profile best-case and workload in the end;

3. *Critical 3*: Profile worst-case and workload in the beginning;

4. *Critical 4*: Profile worst-case and workload in the end;

The random cases are the 100 workloads and power productions generated by Gaussian noise over 10 different traces of each (workload and power production). The results of the baselines and policies are the same as the previous sections.

6.3.1 Critical cases

Scenario Critical 1

Figure 6.4 illustrates all the results obtained for the execution with the profile best-case and workload in the beginning. This scenario has more space for improvement because the job majority arrives on the first day, and we have more energy to finish them than predicted. So, the heuristics have time to decide when to start the jobs and how to approximate the target SoC. The best algorithm is *BEASY*, with 99.01% finished jobs in number and 96.07% size. The jobs not finished are postponed and not killed. It ends above the target SoC. Regarding wasted energy, it is possible to notice that *BEASY* better expends the energy received, resulting in a saving of 35.33% compared with the second-best wasted energy result (*Workload reactive*). It has a higher bounded slowdown compared to the policies but a lower mean than the baselines. It runs more jobs, which makes more jobs wait longer. However, it maintains all slowdowns below 100, while all other algorithms have some jobs with higher slowdowns.

Workload reactive execution kills 8.05% of jobs. This heuristic is too aggressive compared to *BEASY*. As mentioned before, *Workload reactive* puts all jobs to start as soon

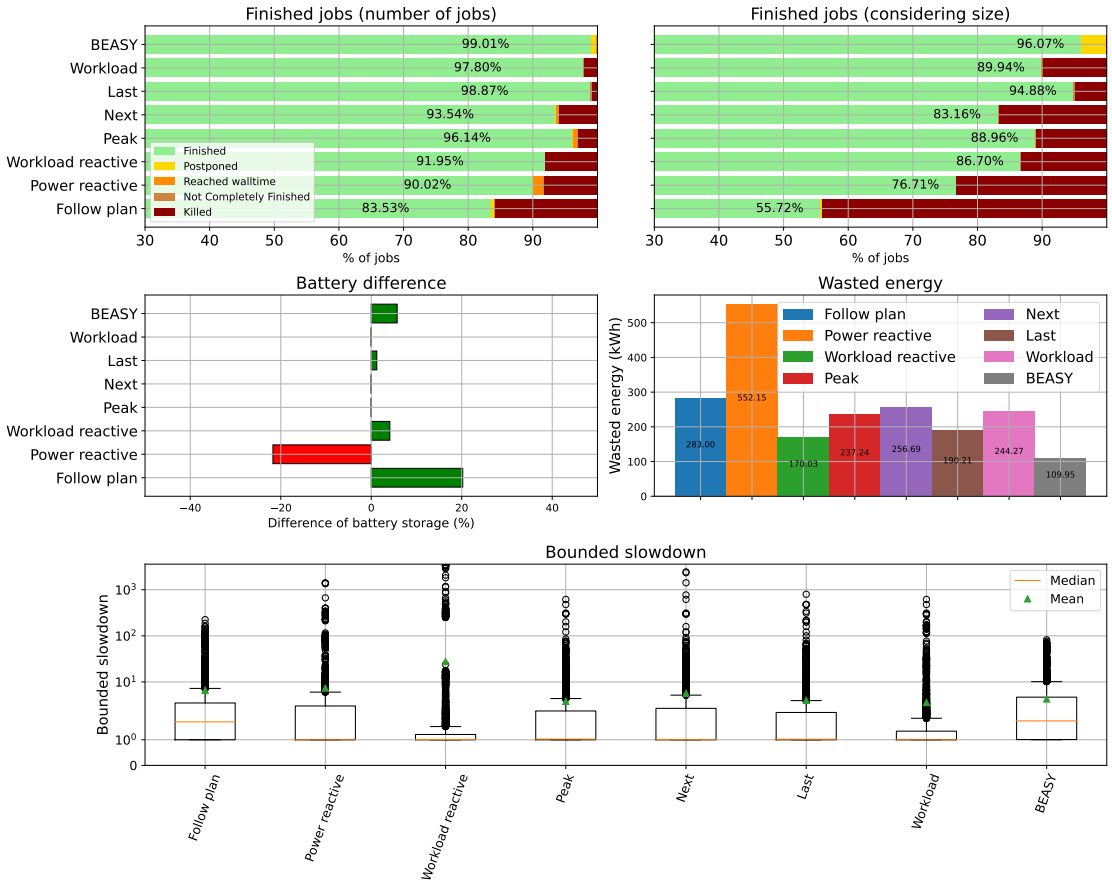

Figure 6.4: Results of *BEASY* on critical case 1.

they arrive. So, it dries the battery too fast. Figure 6.5 compares the state of charge of the *Workload reactive* and *BEASY*. *Workload reactive* dries too fast the battery and needs to kill the jobs, while *BEASY* is conservative. We can see in this scenario that *BEASY* makes better decisions than all other executions. Here, being conservative helped in the result. It finishes more than 99% of the submitted jobs, guaranteeing they have the energy to finish. The 1% is postponed to the next time window. We discuss this approach later. In addition, it finishes saving energy in the battery, which can help the next time step to run the postponed jobs.

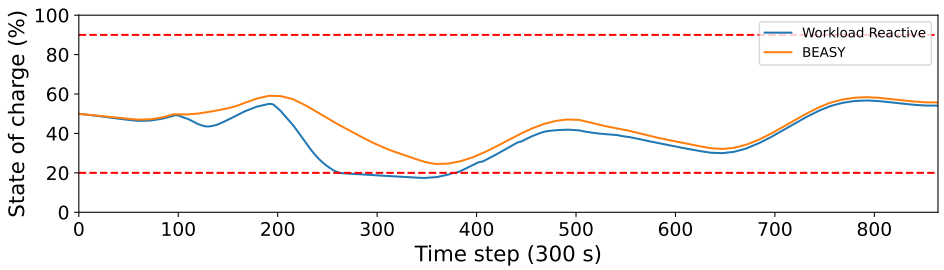

Figure 6.5: Comparison between the state of charge of *Workload reactive* and *BEASY*.

Scenario Critical 2

The second case is more complicated than the previous one. Here, the job majority arrives on the last day of the time window. So, the algorithms have a shorter time to schedule them. Figure 6.6 illustrates the results. *BEASY* is the second best in finished jobs in number. Considering the size, it has lower finished jobs than *Workload* and *Last* policies. However, *BEASY* does not kill any job, while *Workload* and *Last* policies kill almost 5% (in number). Only *Workload reactive* is better than *BEASY* in finished and killed jobs (in number and size). As mentioned before, this is the perfect scenario for the *Workload reactive* approach. Considering the battery level, *BEASY* ended with more energy in the battery. Since the jobs arrive at the end of the time window, *BEASY* prefers to save this energy than starting jobs without knowing if they would finish. *Workload reactive* starts the jobs, ignoring the target level and resulting in a deficit of energy in the battery. Considering the wasted energy, *BEASY* wasted 22.10% more energy than *Workload reactive*, but it wasted less than all the other algorithms. Finally, *BEASY* has a similar bounded slowdown to *Workload reactive*, but with some values above 100. Again, it is complicated to compare algorithms with different jobs finished. But it has a better slowdown than the other algorithms.

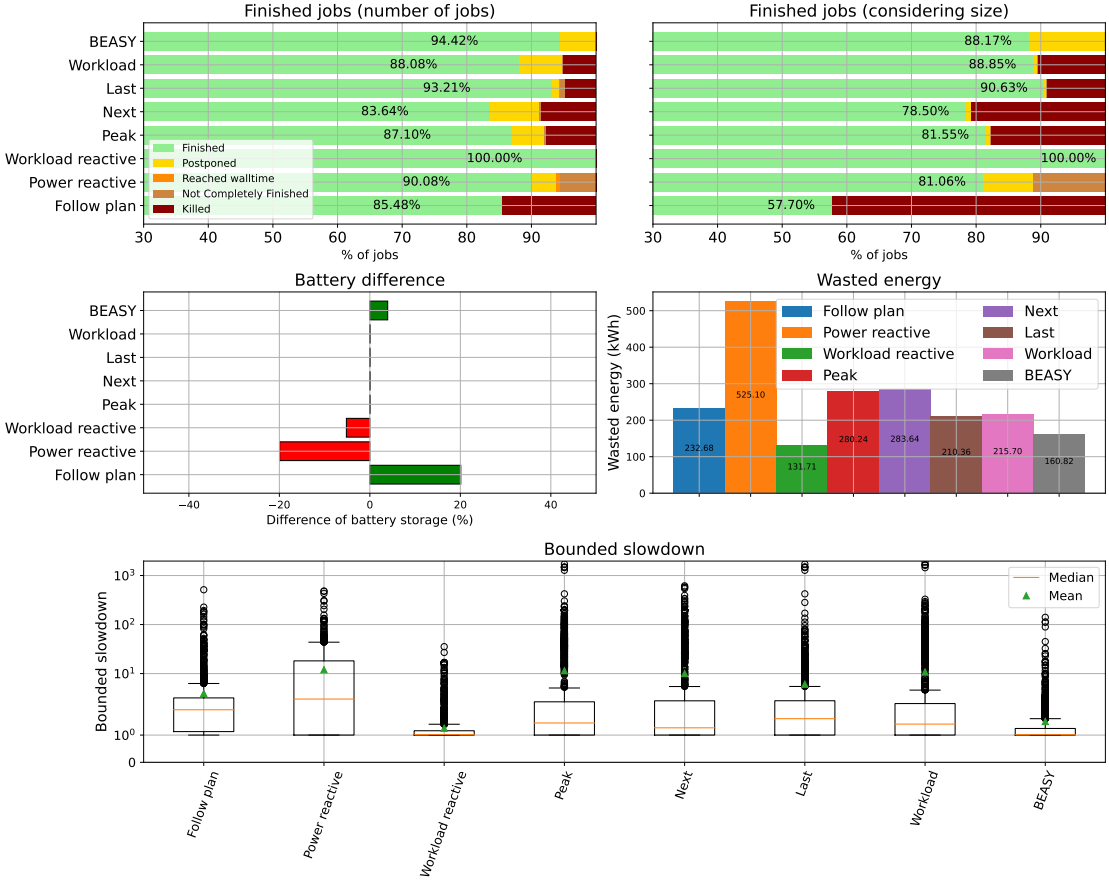

Figure 6.6: Results of *BEASY* on critical case 2.

Scenario Critical 3

The third case is with the worst-case profile and workload in the beginning. In this case, the algorithms have time to schedule the jobs but receive way less energy coming from renewable. So, besides finding the best moment to place the jobs, they must adapt their power usage. Figure 6.7 demonstrates the results. *BEASY* is the second best in finished jobs, very close to the best one *Power reactive*. Besides, *BEASY* kills very few jobs (0.10%). Considering the size, *BEASY* is the best one, finishing more than the *Power reactive*. Comparing *BEASY* and *Power reactive*, the former has a better battery level. *Power reactive* finishes with more than 20% of battery deficit. *Next* policy is the best policy, but it is worst than the *BEASY* in finished jobs (in number and size). The policies and *BEASY* have quite similar battery levels. *BEASY* has the best result on wasted energy, reducing by 31.17% compared to the *Next* (the second-best). This is a scenario where it is essential to use energy efficiently. So this is an excellent result. Finally, *BEASY* has the best mean bounded slowdown, with no job having more than 1000. The other algorithms have several jobs with a very high bounded slowdown (> 1000). However, it has a higher median than all baselines.

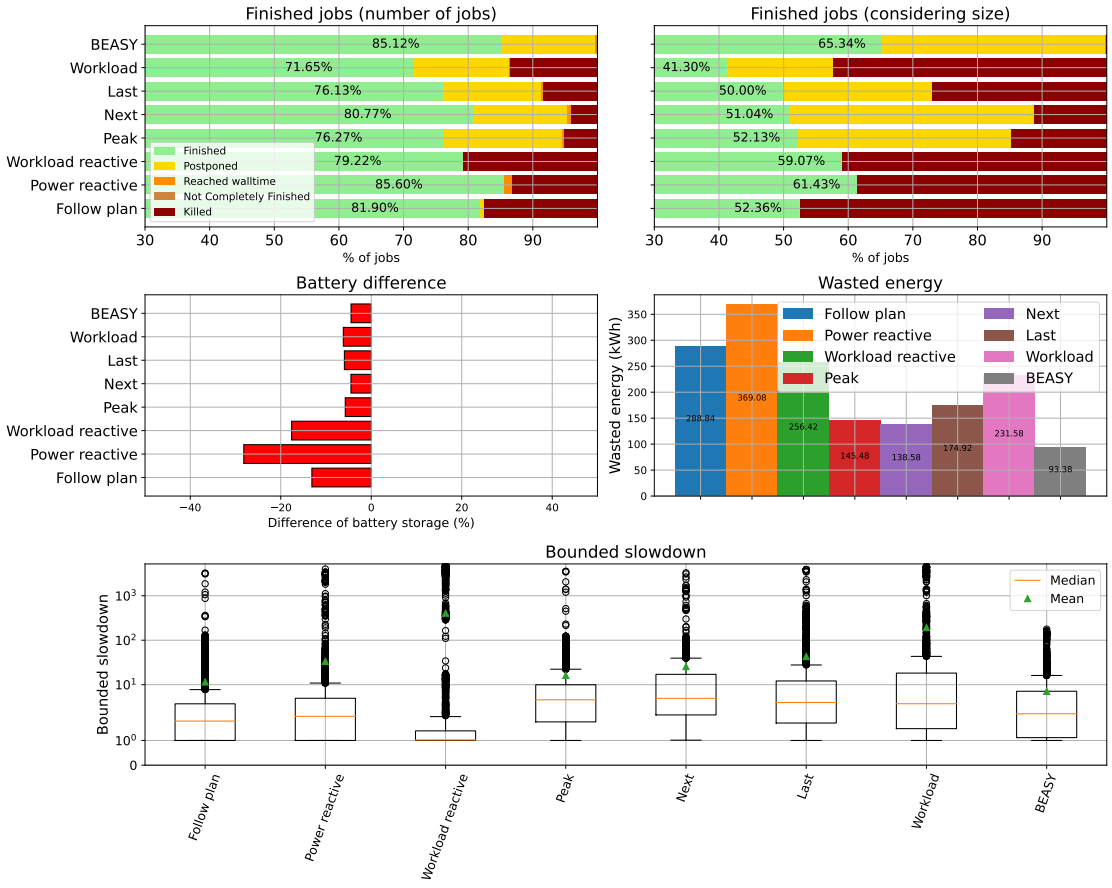

Figure 6.7: Results of *BEASY* on critical case 3.

Scenario Critical 4

The last case is with the profile worst-case and the jobs arriving at the end of the time window. Figure 6.8 shows the results. Among the executions that respected the battery level, *BEASY* has the higher finished jobs and lower killed jobs (in number and sizing). All the baselines have more finished jobs than *BEASY* in size. In number, *BEASY* is best than the policies and the *Power reactive*, but finishes less than *Workload reactive* and *Follow plan*. However, *BEASY* kills fewer jobs than all other algorithms. We can see that *Workload reactive* finishes in number almost the same percentage as *BEASY* but finishes far from the battery target level. This scenario is hard to start big jobs, resulting in a big impact on the size of the jobs executed by *BEASY*. Again, *BEASY* has an outstanding result regarding wasted energy. It wastes less energy than all the other algorithms, with 19.70% less than the second-best, *Workload policy*. Finally, in this case, *BEASY* has a higher median bounded slowdown compared to the other algorithms, but the *Next* policy. The mean value of *BEASY* is similar to *Workload*, *Last*, and *Power reactive* but higher than *Follow plan* and *Workload reactive*. *BEASY* finishes more small jobs due to the power constraints, in which the waiting time has higher importance. Since our bounded slowdown considers only the finished jobs, executing more small jobs can produce a higher slowdown.

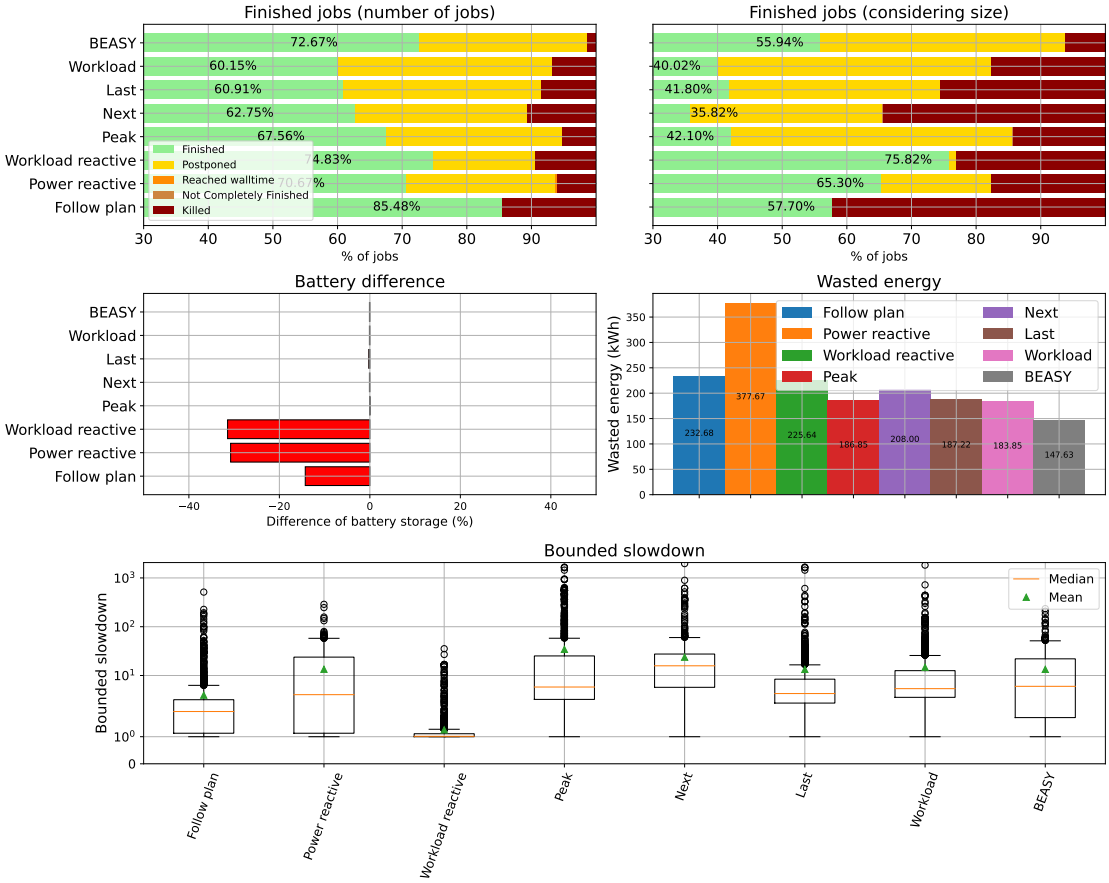

Figure 6.8: Results of *BEASY* on critical case 4.

6.3.2 Random cases

After presenting the critical cases, this section details the random cases. As mentioned previously, we have taken ten different profiles and workloads. Figure 6.9 illustrates the results of the ten executions. Like in the critical cases, *BEASY* presents the lowest number of killed jobs. *BEASY* has no execution with more than 5% (in number) or 20% (in size) of killed jobs. The other algorithms have a higher mean and worst-case. For example, *Workload reactive* has two executions with more than 20% killed jobs in number and one execution with more than 50% in size. The policies reduce these numbers but have higher values than *BEASY*. Considering finished jobs, *BEASY* has the second-best mean of 93.29% (in number) and 85.42 (in size). *Workload reactive* Workload reactive has the best mean with 97.27% (in number) and 90.86 (in size). *Workload reactive* is very aggressive, which helps to have a higher number of finished jobs but also leads to a high killed job number. Figure 6.10 compares the state of charge of *Workload reactive* and *BEASY* in the execution where *Workload reactive* killed several jobs. *Workload reactive* could not avoid the 20% threshold, resulting in several killed jobs. *BEASY* avoids this threshold.

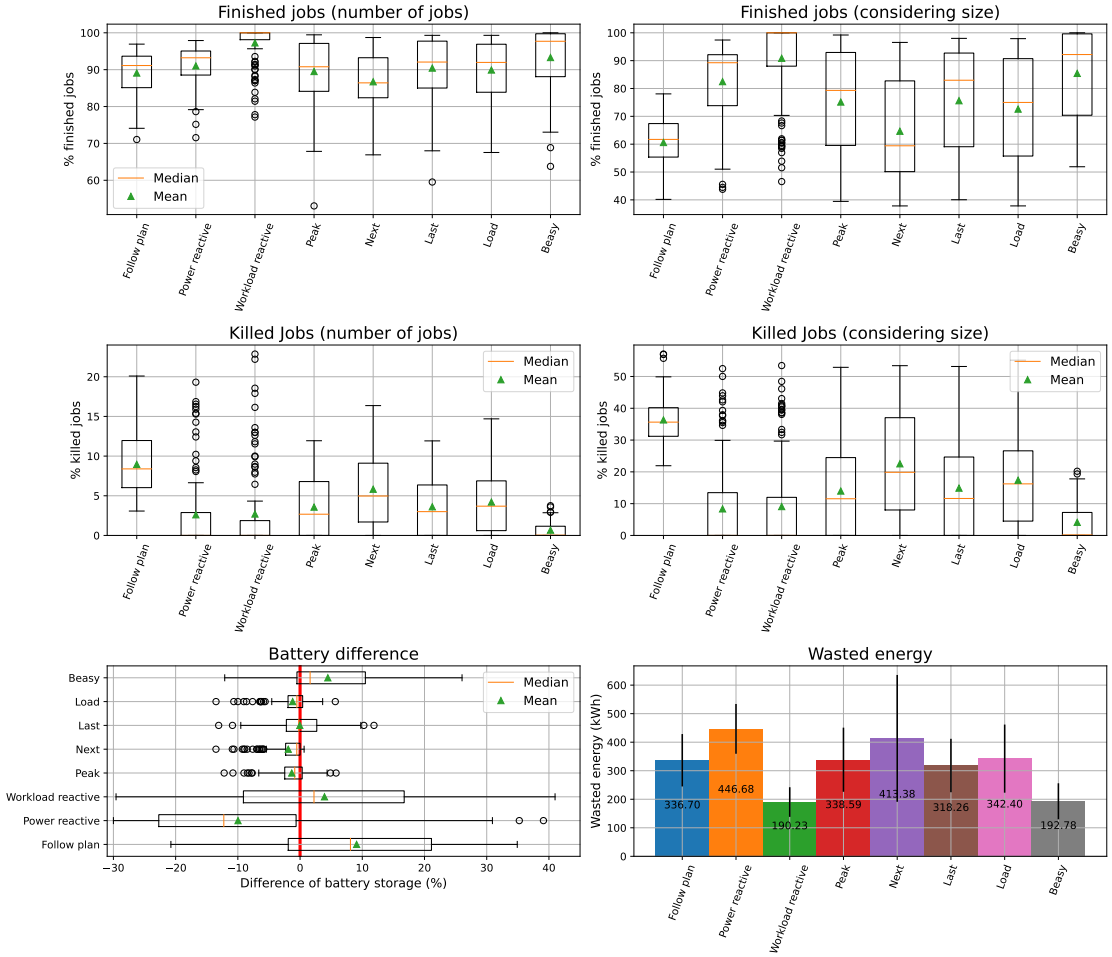

Figure 6.9: Results of *BEASY* on random cases.

Comparing the number of jobs killed, *Workload reactive* has 19% of the executions above 5% of killed jobs (in number), while *BEASY* has none. This result shows that *BEASY* can maintain the state of charge in control. Furthermore, *Workload reactive* has the two worst number of jobs killed among all executions. *Power reactive* has the third-

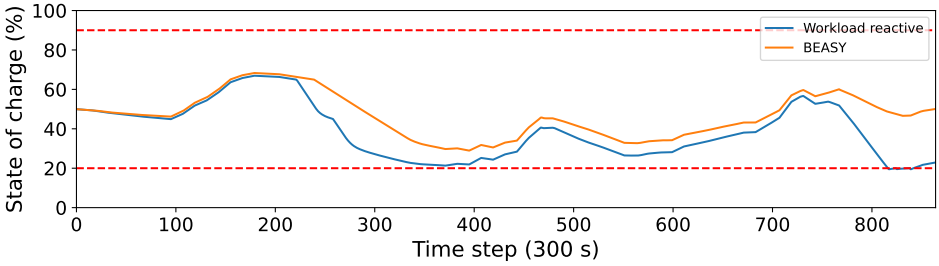

Figure 6.10: State of charge in one of the scenarios. *Workload reactive* kills several jobs when the battery is lower than 20%. *BEASY* avoids this threshold and could maintain the jobs running.

best number of completed jobs but also 19% of the executions above 5% of killed jobs (in number). Considering the battery target level, *BEASY* has good levels since it is near the target level (average of 54.47%). 75% of the executions in *BEASY* finished with the battery at the target level or higher. While *Workload reactive* has a wider battery level, with the minimal being close to -30%, *BEASY* has a better result. The best executions in this metric are the policies (*Peak*, *Next*, *Last*, and *Workload*), always around 50% and with a very low standard deviation. But, we could see they can not reduce the killed jobs as much as *BEASY*. Finally, *BEASY* and *Workload reactive* have the best result in wasted energy, well below the other executions. *Workload reactive* is very energy aware since it tends to maintain running only the servers needed due to its DPM technique. So, being close to the same result is quite outstanding for *BEASY*.

6.3.3 Discussion

After presenting the results of critical and random cases, this section consolidates our discussion. Table 6.1 ranks all algorithm results over the different tested scenarios. We highlight in green the top 3 results on each metric for each scenario. The bottom 3 results are in red. Both finished and killed jobs are considered in number. Killed jobs are: killed jobs + reach the walltime + not completely finished. For SoC, we assume the best results as the higher real SoC at the end of the time window. This table highlights the excellent results of *BEASY* over the different experiments, where it finishes in top 3 in all cases. *BEASY* is always the best one in killed jobs. This heuristic can identify when it is possible to execute more jobs and when it is better to be conservative, postponing some jobs. Running everything is not always possible due to power constraints in our data center environment. Postponing the jobs allows us to plan the next time window considering them. For example, the next time window could use more energy coming from hydrogen to deal with these jobs. In an online way, it is not possible to change the hydrogen usage since it has a warm-up time. Besides, offline can consider the seasonality of renewable production in its decision (e.g., spend more energy from hydrogen in winter and recharge it in summer). Even postponing jobs, *BEASY* is always between the top 3 finished jobs.

The worst result of *BEASY* is third place in finished jobs in critical case 4 (Profile worst-case and workload in the end). This critical case is complicated to guarantee the execution of all jobs. The algorithm with the highest finished jobs metric (*Follow plan*) also has the highest killed jobs in this scenario Furthermore, the second-best (*Workload reactive*) is the third-worst in killed jobs. On the other hand, *BEASY* is the third-best

in finished jobs and the best in killed jobs. *Follow plan* and *Workload reactive* also have worse SoC level than *BEASY*. Regarding the random cases, *Workload reactive* seems a good possibility. However, the third place in SoC (considering the mean) does not illustrate the variance in the state of charge. Figure 6.9 shows that this algorithm varies its SoC, being very unstable. *BEASY* has the second-best SoC. We can see in Figure 6.9 that *BEASY* has a higher SoC variance than the policies, but it is still lower than the baselines. In addition, 75% of its values are higher than the target level, showing that it usually finishes with more energy to use in the next time window. *Follow plan* has a similar result in SoC, but this algorithm has the worst killed jobs and second-worst finished jobs. *Follow plan* saves battery because it does not adapt its plan to improve QoS (finished/killed jobs).

Besides, *BEASY* also has good wasted energy over all executions. It has the best wasted energy three times and the second-best two times. The only algorithm with better results is *Workload reactive*. We expected *Workload reactive* to have good wasted energy since it reacts to job arrival and applies DPM to turn off the servers. Even so, *BEASY* outperforms *Workload reactive* in three of five scenarios. In the random cases, *Workload reactive* has lower wasted energy than *BEASY*, but with a very similar result. *Workload reactive* has two high wasted energy results in the scenarios with lower power production due to the high number of killed jobs. Besides *BEASY*, only *Last* policy has no wasted energy in the bottom 3. However, it is always in third or fourth place.

Finally, let's compare *BEASY* globally. *BEASY* has the best overall results, showing a balanced algorithm. As mentioned before, it finishes in top 3 in all cases. *Workload reactive* also has several good results, mainly regarding job metrics. However, it is too aggressive, which is not a good approach in critical cases 1 and 3 (with more jobs on the first day). On the other hand, following strictly the plan in a conservative way (*Follow plan*) is not a good approach either. The proposed policies are the first step away from being too conservative, showing some good results. For example, *Last* policy has only two bottom 3 QoS results in the scenarios with less energy. However, they are still not enough. *BEASY* can find a balance between QoS and power constraints. The SoC predictions help the SoC to finish close to the target. The finished and killed jobs have good results in *BEASY* due to its scheduling approach. The *BEASY*'s scheduling validation helps to guarantee job executions. Finally, the wasted energy good results come from both scheduling and SoC predictions. Executing small jobs in dangerous moments maintains the SoC under control and reduces killed jobs. Reducing killed jobs also reduces wasted energy. So, combining good scheduling and power compensation decisions makes *BEASY* achieve these outstanding results.

Table 6.1: Consolidate average results in every scenario.

Scenario	Metric	Follow plan	Power reactive	Workload reactive	Peak	Next	Last	Load	BEASY
Profile best-case and workload in beginning	Finished jobs	8th	7th	6th	4th	5th	2nd	3rd	1st
	Killed jobs	8th	7th	6th	4th	5th	2nd	3rd	1st
	SoC	1st	8th	3rd	6th	7th	4th	5th	2nd
	Wasted energy	7th	8th	2nd	4th	6th	3rd	5th	1st
Profile best-case and workload in end	Finished jobs	7th	4th	1st	6th	8th	3rd	5th	2nd
	Killed jobs	8th	5th	1st	6th	7th	4th	3rd	1st
	SoC	1st	8th	7th	3rd	4th	5th	6th	2nd
	Wasted energy	5th	8th	1st	6th	7th	3rd	4th	2nd
Profile worst-case and workload in beginning	Finished jobs	3rd	1st	5th	6th	4th	7th	8th	2nd
	Killed jobs	7th	6th	8th	3rd	2nd	4th	5th	1st
	SoC	6th	8th	7th	3rd	2nd	4th	5th	1st
	Wasted energy	7th	8th	6th	3rd	2nd	4th	5th	1st
Profile worst-case and workload in end	Finished jobs	1st	4th	2nd	5th	6th	7th	8th	3rd
	Killed jobs	8th	3rd	6th	2nd	7th	5th	4th	1st
	SoC	6th	7th	8th	1st	3rd	5th	4th	2nd
	Wasted energy	7th	8th	6th	3rd	5th	4th	2nd	1st
100 average cases	Finished jobs	7th	3rd	1st	6th	8th	4th	5th	2nd
	Killed jobs	8th	6th	2nd	4th	7th	3rd	5th	1st
	SoC	1st	8th	3rd	6th	7th	4th	5th	2nd
	Wasted energy	4th	8th	1st	5th	7th	3rd	6th	2nd

6.4 Conclusion

This chapter presented and evaluated *BEASY*, a heuristic that mixes power and scheduling decisions. *BEASY* uses power production and demand predictions to estimate the state of charge. This SoC estimation helps to make better power compensation decisions. Besides, *BEASY* identifies dangerous moments where the SoC could be too close to the battery's lower boundary. In these dangerous moments, *BEASY* starts to be conservative, preferring to run small over big jobs. On the scheduling side, it validates if the job can run entirely. It verifies if the job's servers are available during the entire execution or if it is possible to migrate energy to maintain the servers running. We compared *BEASY* with three baselines (*Follow plan*, *Power reactive*, and *Workload reactive*) and the four policies presented in Chapter 4. *BEASY* showed outstanding results in finished jobs, killed jobs, SoC difference from the target, and wasted energy. It is in the top 3 of all metrics in every scenario.

Chapter 7

Conclusion and Perspectives

In this chapter, we make overall conclusions and summarize the contributions of this work, along with a discussion of future work.

7.1 Conclusion

The world has a big challenge to reduce its GHG emissions and stop global warming. GHG reduction demands the engagement of all sectors of governments, industry, and research. One sector is Information and Communication Technology (ICT) with emissions in 2020 around 1.8%-2.8% or 2.1%-3.9% considering the supply chain pathways. Data centers are one of the actors inside ICT with a good share of the emissions. Both academy and industries focus their efforts to reduce the emissions of the big data center providers. Reducing their emissions is particularly difficult due to the growth in Internet users and networked devices. Even with this growth, the emissions did not increase at the same pace due mainly to the power optimizations. However, some authors claim that these power consumption improvements are close to their limit. Therefore, data center providers must find other ways to reduce the emissions, such as renewable energy migration.

Some providers, such as Google and Amazon, are investing in an off-site renewable production approach to maintain their servers. In an off-site generation, the data center and power production are not in the same place. So, the providers generate the same amount of energy to the grid that they expend on their servers. However, the biggest challenge of renewable sources is the power intermittence. Since RES production comes from nature, it depends on the climate conditions. Therefore, these providers transfer the problem to third parties. In a net zero scenario (where no energy comes from brown sources), they can not maintain their quality of service without changing the strategy.

The Datazero2 project proposes an architecture and decision process for a renewable-only data center. The architecture is disconnected from the power grid, using only the energy produced by solar panels and wind turbines but with power storage as backups. Using just renewable sources demand different decision levels. We divided the decision level into two possibilities: offline and online. The offline approach has time to search for an optimal solution to match power production and demand. Usually, it predicts both production and demand. The main drawback of the offline approach is that it does not know the real events, so its optimal solution only works if the predictions are good enough. On the other hand, the online approach reacts to real events. However, an online-only approach is myopic, ignoring long-term decisions. In this thesis, we proposed ways to mix offline and online. We demonstrated in Chapter 2 a lack of propositions mixing both. The main objectives of an integrated offline-online approach are to improve the

QoS (maximizing finished jobs and reducing killed jobs), respect the power constraints (approximating battery target level), and reduce the wasted energy.

First, we focused on respecting the power constraints. So, we proposed four compensation policies to adapt power usage, approximating the battery target level at the end of the time window. The compensation changes the power usage according to the power fluctuations derived from production/consumption variations. The compensation policies are *Peak*, *Last*, *Next*, and *Workload*. All policies arrive with better battery levels than the baselines (*Follow plan*, *Power reactive*, and *Workload reactive*). The policies improved the finished jobs in the scenarios with more energy available, compared to an offline-only execution (*Follow plan*). In scenarios with less energy available, they reduced the number of finished jobs but approximated the target battery level at the end of the time window. Nevertheless, there was not a good policy in every execution, demanding further improvements.

After, we tried to introduce a Reinforcement Learning (RL) model to make better compensation decisions. Since every policy arrived at a good battery level at the end of the time window, we tried to mix the compensations driven by a QoS metric. Therefore, we proposed an RL model with two reward propositions linked to QoS (finished, killed, and started jobs). The idea is still to compensate for power variations, which proved to be enough to respect the power constraint, but choosing the actions which improved QoS. We proposed two RL algorithms: Q-Learning and Contextual Multi-Armed Bandit with LinUCB. The results show that they could not improve the Qos, resulting in even worse QoS. These results were explained by several factors, such as chained decisions making it harder to find the best actions, the reward did not represent the global QoS, state/action size limitations, and separated scheduling and power decisions.

Finally, the last contribution is a heuristic which consolidates all the aspects discovered in previous sections. The heuristic is named *BEASY* and englobes the compensation idea (from the policies) and solves the problems from the Reinforcement Learning, such as mixing scheduling and power decisions and using more information from the state of charge estimation (impossible in RL due state/action size limitations). We compared *BEASY* with the baselines and policies, showing good overall results of the new heuristic. *BEASY* had the lowest killed jobs than all the other algorithms while respecting the power constraints. Regarding the finished jobs, it was among the top-3 in every scenario. *Workload reactive* maintains the servers sleeping until a job arrives. This behavior helps this algorithm to have low wasted energy. Even so, *BEASY* had lower wasted energy than *Workload reactive* in three of five cases.

7.2 Perspectives

The work presented in this thesis proposes some ways to mix offline and online decisions in a renewable-only data center. As perspectives, there are some remaining subjects for further improvement.

Change learning process

As we presented in Chapter 5, our model for learning the best compensations did not improve the policy's results. We presented several problems in our model. Therefore, future work can introduce a different learning algorithm, such as Deep Reinforcement Learning (DRL). DRL allows a larger state and action space. Using a larger state space, we could introduce more variables, such as jobs waiting to run, estimated SoC of the future steps (time series), the actual state of running jobs, etc. Besides, the action space can be more precise, using the right step to compensate. Future work can compare DRL with

the heuristics in the time and energy spent to learn and apply the algorithms. Another possibility is changing our reward for a more global metric, such as the time window finished and killed jobs. This reward demands more learning iterations in a simple RL but could produce better results. We hypothesized that doing the best local actions would result in the best global results. However, we saw that this is not true.

Add flexibility in energy constraints

The model presented in Chapter 3 considered a hard constraint of the battery's SoC at the end of the time window. Even if the algorithms (policies and *BEASY*) did not finish with the exact target level, their decisions consider this battery constraint. Future work can evaluate the possibility of a flexible battery level. For example, they can accept $\pm 5\%$ as flexibility at the end of the time window. This flexibility can improve even more the QoS, running more jobs and reducing killed jobs. However, this flexibility must be studied along with the PDM module. Giving $\pm 5\%$ as the battery's flexibility would finish the battery with -5% than the target in every scenario (the algorithms use all the possible energy available). Nevertheless, this behavior can lead to PDM demanding an overestimated target level (e.g., always $+5\%$), which does not change the actual behavior. Another possibility is to introduce hydrogen decisions in the model. In the current model, we considered hydrogen production as fixed input production without changes due to the hydrogen warming-up time. However, the online model can introduce this warming-up time and use hydrogen in critical cases. As for the battery's flexibility, this must be studied together with PDM.

Modify the application type

This thesis focused on HPC batch applications. Future work can evaluate *BEASY* and the policies for other application types. For example, let's imagine a renewable-only streaming scheduler using *BEASY*. A new user sends a request to watch a video from the catalog. The catalog has several videos with different sizes. The video size can be our walltime, which *BEASY* uses to estimate the energy demanded. The user should wait for his time to watch (supposing that they agreed with this to use clean energy). Changing the processor's speed would impact the video quality. Killing a job interrupts the streaming. In dangerous moments (SoC close to the battery lower boundary), *BEASY* lets only small videos. We can evaluate the QoS (waiting time, quality degradation, etc) of *BEASY* in this environment. A possible uncertainty comes from the possibility of pausing the video, which changes the video's finish time. Another possible future work is to introduce in the model other job aspects, such as memory, network communication, etc. This thesis considered that the jobs are impacted only by the CPU. Even if this is the main factor, it is not the only one.

Explore different time definitions

Our work considered a time window of three days and a time step of five minutes. The time window size is not too long to make it difficult to predict the weather conditions and not too short to reduce ODM decision possibilities. In addition, the time step is short enough to maintain ODM reactive (e.g., fastly adapt battery usage) and long enough to finish the server transitions (e.g., the server on and off). However, new works can compare different time windows and time steps, finding the best configurations. For example, would a time window of one day impact the ODM decisions? And a time window of one week? Does a longer time step be too late to adapt power usage? Another possible evaluation is to evaluate chained time windows. For example, the execution of Datazero2 during one year with all modules integrate. This thesis created some offline modules without a direct connection with the online. We ran the offline optimizations, generating input for the online experiments. A complete one-year execution demands the total Datazero2

middleware implementation.

Bibliography

[1] Core Writing Team, Hoesung Lee, José Romero, et al. Climate change 2023: Synthesis report. contribution of working groups i, ii and iii to the sixth assessment report of the intergovernmental panel on climate change. Technical report, Intergovernmental Panel on Climate Change, Geneva, Switzerland, 2023.

[2] Climate Action Tracker. 2100 warming projections: Emissions and expected warming based on pledges and current policies. *Warming Projections Global Update, November*, 2022. Available at: `https://climateactiontracker.org/global/temperatures/`.

[3] AG Olabi and Mohammad Ali Abdelkareem. Renewable energy and climate change. *Renewable and Sustainable Energy Reviews*, 158:112111, 2022.

[4] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon Blair, and Adrian Friday. The climate impact of ict: A review of estimates, trends and regulations, 2021.

[5] George Kamiya. Data centres and data transmission networks. Technical report, International Energy Agency, Paris, 2022.

[6] Md Anit Khan, Andrew P Paplinski, Abdul Malik Khan, Manzur Murshed, and Rajkumar Buyya. Exploiting user provided information in dynamic consolidation of virtual machines to minimize energy consumption of cloud data centers. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 105–114. IEEE, 2018.

[7] U Cisco. Cisco annual internet report (2018–2023) white paper, 2020.

[8] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020. ISSN 0036-8075. doi: 10.1126/science.aba3758. URL `https://science.sciencemag.org/content/367/6481/984`.

[9] Chad Augustine, Richard Bain, Jamie Chapman, Paul Denholm, Easan Drury, Douglas G Hall, Eric Lantz, Robert Margolis, Robert Thresher, Debra Sandor, et al. Renewable electricity futures study. volume 2. renewable electricity generation and storage technologies. Technical report, National Renewable Energy Lab.(NREL), Golden, CO (United States), 2012.

[10] N L Panwar, S C Kaushik, and Surendra Kothari. Role of renewable energy sources in environmental protection: A review. *Renewable and sustainable energy reviews*, 15(3):1513–1524, 2011.

[11] Gustavo Rostirolla, Léo Grange, T Minh-Thuyen, Patricia Stolf, Jean-Marc Pierson, Georges Da Costa, Gwilherm Baudic, Marwa Haddad, Ayham Kassab, Jean-Marc Nicod, et al. A survey of challenges and solutions for the integration of renewable energy in datacenters. *Renewable and Sustainable Energy Reviews*, 155:111787, 2022.

[12] What is renewable energy? `https://www.un.org/en/climatechange/what-is-renewable-energy`. Accessed: 2023-06-08.

[13] Robert Gross, Matthew Leach, and Ausilio Bauen. Progress in renewable energy. *Environment international*, 29(1):105–122, 2003.

[14] Mary Branscombe. Google's solar deal for nevada data center would be largest of its kind. *Informa PLC, London*, 2020.

[15] Philipp Wiesner, Dominik Scheinert, Thorsten Wittkopp, Lauritz Thamsen, and Odej Kao. Cucumber: Renewable-aware admission control for delay-tolerant cloud and edge workloads. In *Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22–26, 2022, Proceedings*, pages 218–232. Springer, 2022.

[16] Marwa Haddad, Jean Marc Nicod, Christophe Varnier, and Marie-Cécile Peéra. Mixed integer linear programming approach to optimize the hybrid renewable energy system management for supplying a stand-alone data center. In *2019 Tenth international green and sustainable computing conference (IGSC)*, pages 1–8. IEEE, 2019.

[17] Yiwen Lu, Ran Wang, Ping Wang, Yue Cao, Jie Hao, and Kun Zhu. Energy-Efficient Task Scheduling for Data Centers with Unstable Renewable Energy: A Robust Optimization Approach. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 455–462, July 2018. doi: 10.1109/Cybermatics_2018.2018.00101.

[18] Íñigo Goiri, Md E Haque, Kien Le, Ryan Beauchea, Thu D Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. Matching renewable energy supply and demand in green datacenters. *Ad Hoc Networks*, 25:520–534, 2015.

[19] Wenyu Liu, Yuejun Yan, Yimeng Sun, Hongju Mao, Ming Cheng, Peng Wang, and Zhaohao Ding. Online job scheduling scheme for low-carbon data center operation: An information and energy nexus perspective. *Applied Energy*, 338:120918, 2023.

[20] Huaiwen He, Hong Shen, Qing Hao, and Hui Tian. Online delay-guaranteed workload scheduling to minimize power cost in cloud data centers using renewable energy. *Journal of Parallel and Distributed Computing*, 159:51–64, 2022.

[21] Stephane Caux, Paul Renaud-Goud, Gustavo Rostirolla, and Patricia Stolf. Phase-based tasks scheduling in data centers powered exclusively by renewable energy. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 136–143. IEEE, 2019.

[22] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: managing server clusters on intermittent power. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 185–198, 2011.

[23] Vanamala Venkataswamy, Jake Grigsby, Andrew Grimshaw, and Yanjun Qi. Rare: Renewable energy aware resource management in datacenters. In *Job Scheduling Strategies for Parallel Processing: 25th International Workshop, JSSPP 2022, Virtual Event, June 3, 2022, Revised Selected Papers*, pages 108–130. Springer, 2023.

[24] Jean-Marc Pierson, Gwilherm Baudic, Stéphane Caux, Berk Celik, Georges Da Costa, Léo Grange, Marwa Haddad, Jerome Lecuivre, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo, Robin Roche, Gustavo Rostirolla, Amal Sayah, Patricia Stolf, Minh-Thuyen Thi, and Christophe Varnier. DATAZERO: DATAcenter with Zero Emission and RObust management using renewable energy. *IEEE Access*, 7:(on line), juillet 2019. URL `http://doi.org/10.1109/ACCESS.2019.2930368`.

[25] Bolun Xu, Alexandre Oudalov, Andreas Ulbig, Göran Andersson, and Daniel S Kirschen. Modeling of lithium-ion battery degradation for cell life assessment. *IEEE Transactions on Smart Grid*, 9(2):1131–1140, 2016.

[26] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, 1:1–14, 2011.

[27] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[28] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, et al. Characterizing job microarchitectural profiles at scale: Dataset and analysis. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[29] Dalibor Klusáček, Šimon Tóth, and Gabriela Podolníková. Real-life experience with major reconfiguration of job scheduling system. In *Job Scheduling Strategies for Parallel Processing: 19th and 20th International Workshops, JSSPP 2015, Hyderabad, India, May 26, 2015 and JSSPP 2016, Chicago, IL, USA, May 27, 2016, Revised Selected Papers 19*, pages 83–101. Springer, 2017.

[30] Stefan Pfenninger and Iain Staffell. Long-term patterns of european pv output using 30 years of validated hourly reanalysis and satellite data. *Energy*, 114:1251–1265, 2016.

[31] Igor Fontana de Nardin, Patricia Stolf, and Stephane Caux. Mixing offline and online electrical decisions in data centers powered by renewable sources. In *IECON 2022–48th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6. IEEE, 2022.

[32] Igor Fontana de Nardin, Patricia Stolf, and Stephane Caux. Analyzing power decisions in data center powered by renewable sources. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 305–314. IEEE, 2022.

[33] John Houghton. Global warming. *Reports on progress in physics*, 68(6):1343, 2005.

[34] United Nations. The Paris Agreement. URL `https://www.un.org/en/climatechange/paris-agreement`. Publisher: United Nations.

[35] Climate Action Tracker. Massive gas expansion risks overtaking positive climate policies. *Warming Projections Global Update, November*, 2022.

[36] Valérie Masson-Delmotte, Panmao Zhai, Hans-Otto Pörtner, Debra Roberts, Jim Skea, Priyadarshi R Shukla, Anna Pirani, Wilfran Moufouma-Okia, Clotilde Péan, Roz Pidcock, et al. Global warming of 1.5 c. *An IPCC Special Report on the impacts of global warming of*, 1(5):43–50, 2018.

[37] IPCC Climate Change. A threat to human wellbeing and health of the planet. *Taking Action Now Can Secure our Future*, 2022.

[38] Tim Wheeler and Joachim von Braun. Climate change impacts on global food security. *Science*, 341(6145):508–513, 2013. doi: 10.1126/science.1239402. URL `https://www.science.org/doi/abs/10.1126/science.1239402`.

[39] Timothy M Lenton, Chi Xu, Jesse F Abrams, Ashish Ghadiali, Sina Loriani, Boris Sakschewski, Caroline Zimm, Kristie L Ebi, Robert R Dunn, Jens-Christian Svenning, et al. Quantifying the human cost of global warming. *Nature Sustainability*, pages 1–11, 2023.

[40] UNESCO. Guide to measuring information and communication technologies (ict) in education, 2009.

[41] Measuring digital development - facts and figures 2022. `https://www.itu.int/itu-d/reports/statistics/facts-figures-2022/`. Accessed: 2023-06-07.

[42] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015.

[43] Lotfi Belkhir and Ahmed Elmeligi. Assessing ict global emissions footprint: Trends to 2040 & recommendations. *Journal of cleaner production*, 177:448–463, 2018.

[44] What is a data center? `https://www.ibm.com/topics/data-centers`. Accessed: 2023-06-07.

[45] Electricity domestic consumption. `https://yearbook.enerdata.net/electricity/electricity-domestic-consumption-data.html`. Accessed: 2023-06-07.

[46] Data centres metered electricity consumption 2021. `https://www.cso.ie/en/releasesandpublications/ep/p-dcmec/datacentresmeteredelectricityconsumption2021/keyfindings/`. Accessed: 2023-06-07.

[47] Klimastatus og -fremskrivning 2023. `https://ens.dk/service/fremskrivninger-analyser-modeller/klimastatus-og-fremskrivning-2023`. Accessed: 2023-06-07.

[48] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[49] Jens Malmodin and Dag Lundén. The energy and carbon footprint of the global ict and e&m sectors 2010–2015. *Sustainability*, 10(9):3027, 2018.

[50] Nana Yaw Amponsah, Mads Troldborg, Bethany Kington, Inge Aalders, and Rupert Lloyd Hough. Greenhouse gas emissions from renewable energy sources: A review of lifecycle considerations. *Renewable and Sustainable Energy Reviews*, 39: 461–475, 2014.

[51] Piotr Bojek. Renewables - energy system overview. Technical report, International Energy Agency, Paris, 2022.

[52] Pierre L Kunsch and Jean Friesewinkel. Nuclear energy policy in belgium after fukushima. *Energy policy*, 66:462–474, 2014.

[53] Chuangang Ren, Di Wang, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Carbon-aware energy capacity planning for datacenters. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 391–400. IEEE, 2012.

[54] Amazon sets a new record for most renewable energy purchased by a single company. `https://www.aboutamazon.eu/news/sustainability/amazon-sets-a-new-record-for-most-renewable-energy-purchased-by-a-single-company`, 2023. Accessed: 2023-06-08.

[55] Raquel S Garcia and Daniel Weisser. A wind–diesel system with hydrogen storage: Joint optimisation of design and dispatch. *Renewable energy*, 31(14):2296–2320, 2006.

[56] Weiqiang Dong, Yanjun Li, and Ji Xiang. Optimal sizing of a stand-alone hybrid power system based on battery/hydrogen with an improved ant colony optimization. *Energies*, 9(10):785, 2016.

[57] Akbar Maleki and Fathollah Pourfayaz. Optimal sizing of autonomous hybrid photovoltaic/wind/battery power system with lpsp technology by using evolutionary algorithms. *Solar Energy*, 115:471–483, 2015.

[58] Sunanda Sinha and SS Chandel. Review of recent trends in optimization techniques for solar photovoltaic–wind based hybrid energy systems. *Renewable and Sustainable Energy Reviews*, 50:755–769, 2015.

[59] Di Wang, Chuangang Ren, Anand Sivasubramaniam, Bhuvan Urgaonkar, and Hosam Fathy. Energy storage in datacenters: what, where, and how much? In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 187–198, 2012.

[60] Ahmet Yilanci, Ibrahim Dincer, and Hasan K Ozturk. A review on solar-hydrogen/fuel cell hybrid energy systems for stationary applications. *Progress in energy and combustion science*, 35(3):231–244, 2009.

[61] Thomas Pregger, Daniela Graf, Wolfram Krewitt, Christian Sattler, Martin Roeb, and Stephan Möller. Prospects of solar thermal hydrogen production processes. *International journal of hydrogen energy*, 34(10):4256–4267, 2009.

[62] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2015.

[63] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys (CSUR)*, 46(4):1–31, 2014.

[64] Qingxia Zhang, Zihao Meng, Xianwen Hong, Yuhao Zhan, Jia Liu, Jiabao Dong, Tian Bai, Junyu Niu, and M Jamal Deen. A survey on data center cooling systems: Technology, power consumption modeling and control strategy optimization. *Journal of Systems Architecture*, 119:102253, 2021.

[65] Ali Hammadi and Lotfi Mhamdi. A survey on architectures and energy efficiency in data center networks. *Computer Communications*, 40:1–21, 2014.

[66] Jindou Yuan, Wenhan Zhang, Ying Zhou, Songsong Chen, and Ciwei Gao. Optimal scheduling of data centers considering renewable energy consumption and temporalspatial load characteristics. In *2022 Power System and Green Energy Conference (PSGEC)*, pages 283–288. IEEE, 2022.

[67] Leila Ismail and Huned Materwala. Computing server power modeling in a data center: Survey, taxonomy, and performance evaluation. *ACM Computing Surveys (CSUR)*, 53(3):1–34, 2020.

[68] Franz Christian Heinrich, Tom Cornebize, Augustin Degomme, Arnaud Legrand, Alexandra Carpen-Amarie, Sascha Hunold, Anne-Cécile Orgerie, and Martin Quinson. Predicting the energy-consumption of mpi applications at scale using only a single node. In *2017 IEEE international conference on cluster computing (CLUSTER)*, pages 92–102. IEEE, 2017.

[69] Issam Raïs, Anne-Cécile Orgerie, Martin Quinson, and Laurent Lefèvre. Quantifying the impact of shutdown techniques for energy-efficient data centers. *Concurrency and Computation: Practice and Experience*, 30(17):e4471, 2018.

[70] Georges Da Costa, Léo Grange, and Inès De Courchelle. Modeling, classifying and generating large-scale google-like workload. *Sustainable Computing: Informatics and Systems*, 19:305–314, 2018.

[71] Mohammad Masdari and Afsane Khoshnevis. A survey and classification of the workload forecasting methods in cloud computing. *Cluster Computing*, 23(4):2399–2424, 2020.

[72] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings. International Conference on Parallel Processing Workshop*, pages 514–519. IEEE, 2002.

[73] Shinichiro Takizawa and Ryousei Takano. Effect of an incentive implementation for specifying accurate walltime in job scheduling. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 169–178, 2020.

[74] Ignacio J Perez-Arriaga. Managing large scale penetration of intermittent renewables. In *MITEI Symposium on Managing Large-Scale Penetration of Intermittent Renewables, Cambridge/USA*, volume 20, page 2011, 2011.

[75] Aidan Tuohy, John Zack, Sue Ellen Haupt, Justin Sharp, Mark Ahlstrom, Skip Dise, Eric Grimit, Corinna Mohrlen, Matthias Lange, Mayte Garcia Casado, et al. Solar forecasting: methods, challenges, and performance. *IEEE Power and Energy Magazine*, 13(6):50–59, 2015.

[76] Saurabh S Soman, Hamidreza Zareipour, Om Malik, and Paras Mandal. A review of wind power and wind speed forecasting methods with different time horizons. In *North American power symposium 2010*, pages 1–8. IEEE, 2010.

[77] Rahul Sharma and Diksha Singh. A review of wind power and wind speed forecasting. *Journal of Engineering Research and Application*, 8(7):1–9, 2018.

[78] Edward Baleke Ssekulima, Muhammad Bashar Anwar, Amer Al Hinai, and Mohamed Shawky El Moursi. Wind speed and solar irradiance forecasting techniques for enhanced renewable energy integration with the grid: a review. *IET Renewable Power Generation*, 10(7):885–989, 2016.

[79] Anne-Cecile Orgerie, Laurent Lefevre, and Jean-Patrick Gelas. Demystifying energy consumption in grids and clouds. In *International Conference on Green Computing*, pages 335–342. IEEE, 2010.

[80] Michael K Patterson. The effect of data center temperature on energy efficiency. In *2008 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pages 1167–1174. IEEE, 2008.

[81] Avneesh Vashistha and Pushpneel Verma. A literature review and taxonomy on workload prediction in cloud data center. In *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 415–420. IEEE, 2020.

[82] Yves Robert and Frédéric Vivien. *Introduction to scheduling*. CRC Press, 2009.

[83] Pragati Agrawal and Shrisha Rao. Energy-efficient scheduling: classification, bounds, and algorithms. *Sādhanā*, 46(1):46, 2021.

[84] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[85] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information sciences*, 237:82–117, 2013.

[86] Roger B Myerson. *Game theory: analysis of conflict*. Harvard university press, 1991.

[87] Martin J Osborne et al. *An introduction to game theory*, volume 3. Oxford university press New York, 2004.

[88] Chonglin Gu, Chunyan Liu, Jiangtao Zhang, Hejiao Huang, and Xiaohua Jia. Green scheduling for cloud data centers using renewable resources. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 354–359. IEEE, 2015.

[89] Ayham Kassab, Jean-Marc Nicod, Laurent Philippe, and Veronika Rehn-Sonigo. Scheduling independent tasks in parallel under power constraints. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 543–552. IEEE, 2017.

[90] Ayham Kassab, Jean-Marc Nicod, Laurent Philippe, and Veronika Rehn-Sonigo. Assessing the use of genetic algorithms to schedule independent tasks under power constraints. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 252–259. IEEE, 2018.

[91] Ayham Kassab, Jean-Marc Nicod, Laurent Philippe, and Veronika Rehn-Sonigo. Green power constrained scheduling for sequential independent tasks on identical parallel machines. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 132–139. IEEE, 2019.

[92] Fengsong Hu, Xiajie Quan, and Can Lu. A schedule method for parallel applications on heterogeneous distributed systems with energy consumption constraint. In *Proceedings of the 3rd International Conference on Multimedia Systems and Signal Processing*, pages 134–141, 2018.

[93] Yiwen Lu, Ran Wang, Ping Wang, Yue Cao, Jie Hao, and Kun Zhu. Energy-efficient task scheduling for data centers with unstable renewable energy: A robust optimization approach. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 455–462. IEEE, 2018.

[94] Stephane Caux, Paul Renaud-Goud, Gustavo Rostirolla, and Patricia Stolf. It optimization for datacenters under renewable power constraint. In *European Conference on Parallel Processing*, pages 339–351. Springer, 2018.

[95] Jiechao Gao, Haoyu Wang, and Haiying Shen. Smartly handling renewable energy instability in supporting a cloud datacenter. In *2020 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 769–778. IEEE, 2020.

[96] Baris Aksanli, Jagannathan Venkatesh, Liuyi Zhang, and Tajana Rosing. Utilizing green energy prediction to schedule mixed batch and service jobs in data centers. In *Proceedings of the 4th workshop on power-aware computing and systems*, pages 1–5, 2011.

[97] Chao Li, Rui Wang, Depei Qian, and Tao Li. Managing server clusters on renewable energy mix. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11 (1):1–24, 2016.

[98] Yunbo Li, Anne-Cécile Orgerie, and Jean-Marc Menaud. Balancing the use of batteries and opportunistic scheduling policies for maximizing renewable energy consumption in a cloud data center. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 408–415. IEEE, 2017.

[99] Léo Grange, Georges Da Costa, and Patricia Stolf. Green it scheduling for data center powered with renewable energy. *Future Generation Computer Systems*, 86: 99–120, 2018.

[100] Kawsar Haghshenas, Somayye Taheri, Maziar Goudarzi, and Siamak Mohammadi. Infrastructure aware heterogeneous-workloads scheduling for data center energy cost minimization. *IEEE Transactions on Cloud Computing*, 2020.

[101] Sanjib Kumar Nayak, Sanjaya Kumar Panda, Satyabrata Das, and Sohan Kumar Pande. An efficient renewable energy-based scheduling algorithm for cloud computing. In *International Conference on Distributed Computing and Internet Technology*, pages 81–97. Springer, 2021.

[102] Xiaopu Peng, Tathagata Bhattacharya, Jianzhou Mao, Ting Cao, Chao Jiang, and Xiao Qin. Energy-efficient management of data centers using a renewable-aware scheduler. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8. IEEE, 2022.

[103] Raphael Hunger. *Floating point operations in matrix-vector calculus*. Munich University of Technology, Inst. for Circuit Theory and Signal . . . , 2005.

[104] Sonal Saha and Binoy Ravindran. An experimental evaluation of real-time dvfs scheduling algorithms. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[105] Georges da Costa. MojitO/S, November 2021. URL `https://hal.archives-ouvertes.fr/hal-03453537`.

[106] Georges da Costa. Keynote Lecture: Performance and Energy models for modern HPC servers, June 2022. URL `https://pdco2022.sciencesconf.org/resource/page/id/5`.

[107] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[108] Dalibor Klusáček, Šimon Tóth, and Gabriela Podolníková. Real-life experience with major reconfiguration of job scheduling system. In *Job scheduling strategies for parallel processing*, pages 83–101. Springer, 2015.

[109] Michele M Rienecker, Max J Suarez, Ronald Gelaro, Ricardo Todling, Julio Bacmeister, Emily Liu, Michael G Bosilovich, Siegfried D Schubert, Lawrence Takacs, Gi-Kong Kim, et al. Merra: Nasa's modern-era retrospective analysis for research and applications. *Journal of climate*, 24(14):3624–3648, 2011.

[110] Iain Staffell and Stefan Pfenninger. Using bias-corrected reanalysis to simulate current and future wind power output. *Energy*, 114:1224–1239, 2016.

[111] Violaine Villebonnet, Georges Da Costa, Laurent Lefèvre, Jean-Marc Pierson, and Patricia Stolf. Energy aware dynamic provisioning for heterogeneous data centers. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 206–213. IEEE, 2016.

[112] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator. In *20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, United States, May 2016. doi: 10.1007/978-3-319-61756-5\{_}10. URL `https://hal.archives-ouvertes.fr/hal-01333471`.

[113] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437. IEEE, 2001.

[114] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE transactions on parallel and distributed systems*, 12(6):529–543, 2001.

[115] Jérôme Lelong, Valentin Reis, and Denis Trystram. Tuning easy-backfilling queues. In *Job Scheduling Strategies for Parallel Processing: 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers 21*, pages 43–61. Springer, 2018.

[116] Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, and Vivienne Sze. A method to estimate the energy consumption of deep neural networks. In *2017 51st asilomar conference on signals, systems, and computers*, pages 1916–1920. IEEE, 2017.

[117] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[118] Recommender systems using linucb: A contextual multi-armed bandit approach. `https://towardsdatascience.com/recommender-systems-using-linucb-a-contextual-multi-armed-bandit-approach-35a6f0e` 2020. Accessed: 2023-07-24.