# BEASY: Making EASY backfilling renewable-only

Igor Fontana de Nardin
*Laplace UMR5213, IRIT,*
*Université de Toulouse, CNRS, Toulouse INP*
Toulouse, France
igor.fontana@irit.fr

Patricia Stolf
*IRIT, Université de Toulouse,*
*CNRS, Toulouse INP, UT3*
Toulouse, France
patricia.stolf@irit.fr

Stephane Caux
*Laplace UMR5213,*
*Université de Toulouse*
Toulouse, France
stephane.caux@laplace.univ-tlse.fr

***Abstract*—Reducing greenhouse gas (GHG) emissions from Information and Communication Technology (ICT) has become a hot topic since the Paris Agreement. Data centers are one of the most impactful ICT energy consumers since they are built to run 24 hours / 7 days. An emerging discussion is switching their power supply from brown to green energy, using Renewable Energy Sources (RES). However, this change introduces uncertainties linked to production intermittence. This work is part of the Datazero2 project. This project designs a data center powered only by renewable production, adding storage elements to reduce the impact of the intermittence. A clean-by-design data center requires several decisions at different levels of management. To do so, it uses predictions to plan the actions for the next few days. However, it also needs to react to the actual events that can vary from the forecast. This work presents the *BEASY* heuristic. *BEASY* mixes power and scheduling decisions in a renewable-only data center, seeking to reduce the number of killed jobs and wasted energy. The results demonstrate that *BEASY* reduced wasted energy by up to 35.33% in critical cases. Considering the killed jobs, it also kills fewer jobs than the state of art algorithms in all executions.**

***Index Terms*—Scheduling, Renewable sources, Data center, Storage management**

## I. Introduction

The Information and Communications Technology (ICT) share of global greenhouse gas (GHG) emissions is around 1.8-2.8%, or something between 2.1% and 3.9% considering its supply chain [1]. The data centers sector is one of the most electricity-expensive ICT actors since they provide services with a 100% uptime guarantee [2]. For example, a report revealed that, in 2015, Google data centers consumed the same energy amount as San Francisco, California [3]. Predictions show that the tendency is getting worsen without political and industrial action [1]. These predictions indicate that we are reaching the limit of improvements in processor technologies while expanding internet usage [1], [4]. This scenario makes the ICT community discuss how to reduce the impact of data centers on greenhouse gas emissions [5]. One possibility is migrating from brown (from polluting sources, such as gas, coal, and oil) to green energy (from renewable sources, such as wind and sunlight) [5]. However, renewable sources introduce uncertainties due to weather conditions. Big cloud providers (e.g., Google and Amazon) reduce the impact of power intermittence by inserting grid connections [6]. So, they are not 100% green data centers. This work is part of the Datazero2 project. This project designed a data center operated only by Renewable Energy Sources (RES) without any link to the grid. Datazero2 aims to provide a feasible architecture to maintain data centers 100% clean.

A clean-by-design data center introduces several elements to provide energy to the IT servers, such as Wind turbines, Solar panels, Batteries, and Hydrogen tanks. Aiming to reduce the impact of weather intermittence, it forecasts the near future power production. Knowing the incoming energy allows it to plan the storage (battery and hydrogen) usage. Also, it predicts the power demand from the users since it is another source of uncertainty. For example, if the data center receives fewer requests on the weekend, it can store energy to deal with the week's demand. So, the project includes an offline part to plan the actions for the next few days. However, the actual power production and demand can vary from the predictions. Therefore, this project also includes an online module to react to the actual events. The online module receives jobs to execute, scheduling them when possible. It also adapts storage usage according to power production and demand. For example, if the power production is higher than expected, it will profit to speed up the running jobs or to start new ones. On the other hand, if the power production is lower than expected, it tries to reduce the impact of it in the running jobs, mainly avoiding killing them. This article describes a heuristic named *BEASY* to aggregate all these decisions. This heuristic is a modification of the well-known algorithm EASY Backfilling. Before placing a job in the servers, *BEASY* verifies if the servers will be available during all the estimated job execution. If so, it places the job. If not, *BEASY* evaluates the possibility of migrating energy from the battery usage to finish this job. Also, it considers the power demand and production predictions to calculate the potential battery state of charge (SoC). It uses the possible SoC to find the moments to be more careful, reducing the number of killed jobs due to lack of energy. Results show that it reduces the number of killed jobs and provides better energy usage.

This paper is organized as follows. Section II presents the related work, highlighting the gap in state-of-the-art. Then Section III shows an overview of the problem. Section IV addresses the proposed model. Section V presents the experi-

This work was supported by the French Research Agency under the project Datazero 2 (ANR-19-CE25-0016). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

ments and the discussion about the results. Finally, Section VI concludes the article.

## II. Related work

Considering the challenge of reducing the impact of ICT on GHG emissions, several works propose power data centers using RES. In [7], the authors propose RARE, a Renewable energy Aware Resource management. This manager uses Deep Reinforcement Learning (DRL) to define the job to run, considering the job's demanded resources. Their model maximizes the job value (a value given by the user for each job). The energy comes from solar, wind, batteries, and the grid, but they do not make decisions on the power side (it is a future work). The authors in [8] focus on scheduling different tasks from a job. They receive a Directed Acyclic Graph (DAG) from the user. They propose a DRL to find the best scheduling considering the tasks' dependency and reducing the cost of the data center. The reward considers the carbon emission price, the price to buy energy from the grid, and the QoS penalty. In work [9], the authors also try to minimize the data center cost, considering the power consumption of the servers and cooling. Renewable production includes wind and solar, adding batteries to reduce the intermittency impact. Their work has integration with the grid to buy/sell energy. [10] proposes an admission control policy, named Cucumber, that accepts jobs only if they can be computed within their deadlines without the use of grid energy. They estimate the power production and consumption, using the remainder of both (production minus consumption) to accept new jobs. Cucumber has no link to the grid. However, it focuses on using well the energy from the moments with peak production.

Work [11] also proposes an optimization to reduce the data center cost. The authors calculate the cost considering the price of the energy from the grid, but also the cost of building a renewable power plant, daily maintenance cost, and power capacity. They show an online algorithm to solve the optimization problem. This algorithm considers the load as a power demanded, divided into delay tolerant and delay sensitive. The decision is a trade-off between executing delay tolerant workload or buying energy from the grid. The authors in [12] proposed a renewable-only scheduler. This scheduler considers phase-based tasks. The scheduler meets the power constraint from an external module. The scheduler can degrade the jobs to meet the power capping. However, they do not consider power decisions, such as using more power from the battery. Work [13] introduces a scheduler to define task placement considering energy consumption constraints. The authors do not specify the source of this constraint, but it could be from renewable production. They focus on, knowing all the tasks in a job, which task placement minimizes scheduling length while satisfying energy consumption constraints.

The authors in [14] apply the Lyapunov optimization technique to optimize a stochastic optimization problem taking heterogeneous service delay guarantees and battery management into account. The energy comes from the grid, but they introduce batteries to stock when the grid's price is low. The authors in [15] design Blink, an abstraction for handling intermittent power constraints. This abstraction blinks (activating and deactivating them in succession) the servers to control the power consumption. Blink can be useful for some web applications, but it is not for all applications type. The authors of [16] proposed a Mixed Integer Linear Programming (MILP) to optimize the commitment of a data center powered by only wind turbines, solar panels, batteries, and hydrogen storage systems. This work focuses on offline decisions, using demand and production estimation. Finally, [17] proposes a mix of offline and online decisions, dealing with a renewable-only data center. Offline predicts usage and demand, creating an offline plan. Online modifies this plan according to the actual production demand. The modifications are mainly the compensation of the variance between planned and actual. For example, if the production is higher than predicted, they can use more power to run more jobs. They propose three policies of compensation.

It is possible to notice that the majority of the works consider only online [7], [8], [11], [12], [15] or offline decisions [9], [10], [13], [16]. Therefore, to the best of our knowledge, only work [17] proposes a mix between offline and online decisions, for a renewable-only data center and considering battery awareness. However, this work does not estimate battery usage from jobs in future steps and proposes simplistic compensation policies. Also, it only evaluates the information given by the offline plan, whereas *BEASY* uses predictions to find the best battery changes. We compare our results with the policies proposed in [17].

## III. Problem statement

The main problem in a renewable-only data center is dealing with uncertainties from weather and workload. In mixed data centers (renewable and grid), the grid mitigates the impact of these uncertainties since they can buy power when needed. However, these mixed data centers are not 100% green energy consumption. A renewable-only data center can use storage (e.g., battery and hydrogen) to help in power production intermittence. So, when there is more energy, it can store the surplus energy, using it later. Nevertheless, storages introduce another decision level (when recharge/discharge). Renewable-only data centers can introduce predictions, helping in these decisions. For example, introducing power production prediction can help to find the best moments to recharge and discharge the batteries. However, the reality can be different from the forecasts, and it needs to react correctly to the actual events. Thus, it is crucial to mix offline and online decisions, which is a gap in the state of the art (as presented in Section II).

Datazero2 designs a renewable-only data center. It includes all elements, from the architecture (e.g., electrical and IT sizing, power connections) to the decisions (e.g., power engagement, job scheduling, server states). Renewable power production comes from wind turbines and solar panels. It includes batteries and hydrogen as storage to deal with this intermittence. Fig. 1 illustrates the decision process elements.

There are two main blocks: Offline and Online. The offline predicts demand and production and creates a plan for the near future (also named time window, see Fig. 2). The online receives the offline plan and adapts it according to reality. In the offline part, IT Decision Module (ITDM) predicts the power demand from the workload for the near future, while Power Decision Module (PDM) forecasts the power production from renewable sources. ITDM focuses on turning on/off servers and defining their speed to deal with the predicted demand, and PDM concentrates on responsibly engaging the sources (renewable and storage). Each module calculates a power profile using its predictions. The power profile is a time series of the power given to the data center. So, while ITDM tries to approximate the power profile to the demand, PDM verifies which power profile is possible with the incoming renewable production.

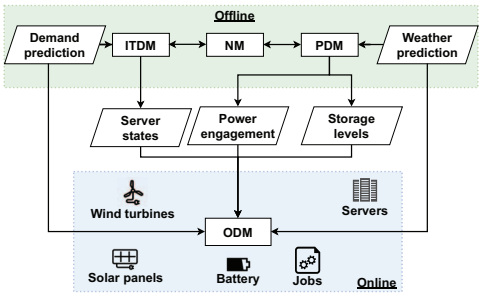


Fig. 1. Online and offline integration.

Both decision modules send a proposition of power profile to the Negotiation Module (NM). NM evaluates both power profiles and returns a new one. Offline runs several negotiations until both modules agree or it reaches a timeout. After the negotiation step, ITDM defines the server configuration (server on/off and speed) for the following days (time window), which meets the negotiated power profile. On the other side, PDM plans source engagement to provide the agreed power profile. Fig. 2 shows the time window and decision steps. The time window is divided into several time steps of 5 minutes. The modules can change the decisions from one time step to another, but the actions stay constant inside each step. Actions mean: turning on/off a server, changing the server's speed, and using more or less power from storage.

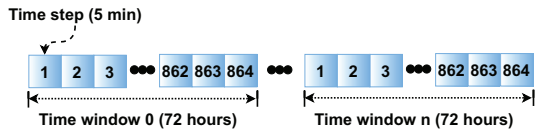


Fig. 2. Offline modules plan the actions for the next three days (time window).

Both offline decision modules dispatch their results to the Online Decision Module (ODM). This article focuses on ODM. ODM uses offline plans to guide its decision. However, the actual power production and demand can vary from the predicted ones. So, ODM must react to the real values, adapting the plans. Besides changing the power plans, ODM is also in charge of the job scheduling decisions. The user submits a job with the maximum execution time (named walltime) and the resources demanded by the job. Using both information and the submission time, ODM needs to select the jobs in the queue to execute and define the moment and the servers to process them. The job result can be finished, killed, or postponed to the next time window. ODM has three objectives: maximizing finished jobs, reducing the number of jobs killed, and respecting the storage level at the end of the time window. Every job killed represents wasted energy since it only processes a part of the job (the job's result is not valuable). It is better to postpone a job than to kill it. However, a scheduler is not good if it postpones all jobs. Thus, ODM needs to balance starting new jobs that can be killed or postponing them. It kills jobs when there is no energy to keep them running. Usually, batteries can deal with the uncertainty of production and demand, providing the energy to maintain servers running. However, when the state of charge of the batteries arrives at critical levels, it is not possible to keep the servers processing. At this point, ODM puts servers to sleep, killing their jobs.

ODM also kills a job when the job's execution time is greater than the walltime. Several HPC data centers, such as GRID5000[1] and METACENTRUM[2], apply this policy. Nevertheless, ODM uses Dynamic Voltage-Frequency Scaling (DVFS) technique to reduce the energy spent by the servers' processor in critical moments. Reducing the processor energy spent also reduces its speed. So, ODM can not maintain all processors at minimum speed all the time because this would result in several jobs reaching the walltime. Additionally, the offline plan gives an expected state of the charge of the storage elements at the end of the time window. ODM must finish the time window with the battery level as close as possible to this target level. Since there are chained time windows (see Fig. 2), it is not effective to dry the battery every time window. Thus, ODM introduces power compensations to deal with this target battery constraint. For example, if ODM increases energy usage now, it must use less in the future. To sum up, ODM must schedule the jobs, consider the predictions and the battery levels to avoid killing them, balance the servers' speed finishing the jobs before the walltime, and finalize the time window with the battery level close to the target.

## IV. Proposed Model

This section presents the heuristic applied in ODM to deal with its different responsibilities. We named this heuristic *BEASY* (Battery Easy Backfilling). This heuristic acts in three different moments. First, Section IV-A explains the predictions used through the *BEASY*'s decisions. These predictions are made at the beginning of the time window, just one time. Then, Section IV-B describes the modifications in the EASY Backfilling heuristic to introduce battery awareness. The modified EASY Backfilling acts every time a job finishes, arrives,

[1]https://www.grid5000.fr/
[2]https://metavo.metacentrum.cz/

or new servers are available. Finally, Section IV-C defines the compensation policies. *BEASY* compensates every time step.

### A. Predictions

As presented in Fig. 1, ODM receives two predictions from offline modules: power production and power demand. Offline can predict both using different methods, such as Regression-based (ARIMA, Support Vector Regression), Classifiers (Neural Network, Support Vector Machine), and Stochastic (Markov Model-based, Queueing Theory-based) [18], [19]. 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. Fig. 3 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 different possible states of charge using equations 1, 2, and 3:

$$SoC_t = (SoC_{t-1} \times \sigma) + (Ech_t \times \eta_{E_{ch}}) - (Edch_t \times \eta_{E_{dch}}) \quad (1)$$

$$Ech_t = \begin{cases} Er_t - Ed_t, & \text{if } Er_t > Ed_t \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

$$Edch_t = \begin{cases} Ed_t - Er_t, & \text{if } Er_t < Ed_t \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

Where:

- $SoC_t$: State of charge in time step $t$;
- $\sigma$: The natural battery discharge rate;
- $Ech_t$: Energy to charge the battery;
- $Edch_t$: Energy to discharge the battery;
- $\eta_{E_{ch}}$: Charge efficiency;
- $\eta_{E_{dch}}$: Discharge efficiency;
- $Er_t$: Estimated energy production from renewable;
- $Ed_t$: Estimated energy demanded.

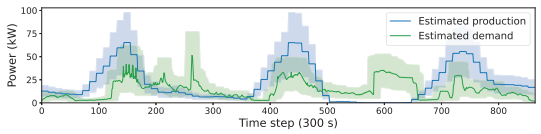


Fig. 3. Renewable production and demand prediction. The blue (production) and green (demand) areas are the uncertainty given by the forecast.

Fig. 4 demonstrates the result of applying Equation 1 using nine different predictions. *BEASY* calculates 9 SoC possibilities combining lower, median, and upper boundaries from the area presented in Fig. 3 (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 IV-C will describe how we use these SoCs to compensate. Fig. 4 also illustrates both SoC upper and lower thresholds (red dashed lines). Setting upper and lower thresholds helps to increase the battery lifetime [20]. The narrower the range, the longer the expected lifetime [20]. 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. Fig. 4 indicates this moment. It considers a dangerous area when more than half of the predicted SoC curves are below the lower threshold. Taking Fig. 4 example with nine predictions, *BEASY* considers the dangerous point where five curves are below 20%. Section IV-B will explain how it uses these moments to make better scheduling decisions.

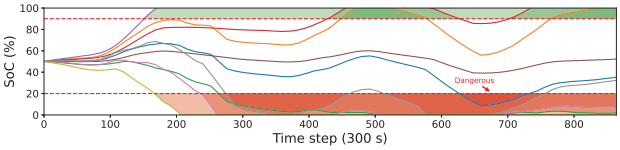


Fig. 4. Result of the Equation 1 for different predictions. The dangerous area is when 5 or more curves (so, more than half of them) are below 20%.

### B. BEASY scheduling

One of the most important ODM's duties is placing jobs on servers. To do so, *BEASY* implements a well-known heuristic name EASY backfilling, but using two different sorts [21], [22]. We detail how we use both sorts in this section. Algorithm 1 presents the main idea (based on [22], but with some modifications). *BEASY* runs this algorithm when a job arrives, a job finishes, or there are new servers 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 servers $S$ are available now. We added the following verifications:

1) It verifies if the servers $S$ are 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. To do 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 have 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. Fig. 5 illustrates this verification. In the figure 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 30 freely. Taking energy from after hour 30 could violate the lower threshold since we will use more energy from the batteries.;

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 1).

**Algorithm 1:** *BEASY* scheduling. Modified from [22].

```
   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
```

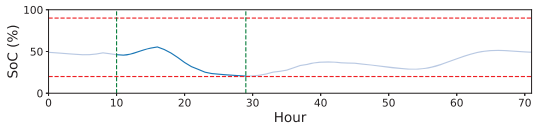


Fig. 5. Verification of possible energy to save.

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 it can 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 to the lower threshold, the next job takes it into account. Continuing in algorithm 1, if the tests pass, then it starts the job (line 7). When it finds a job that can not be placed now, it starts to backfill (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:

$$bsld_j = \max(\frac{w_j + p_j}{\max(p_j, \tau)}, 1) \quad (4)$$

Where:

- $bsld_j$: Bounded Slowdown of job $j$;
- $p_j$: Job $j$ execution time (we consider the walltime here);
- $w_j$: Job $j$ waiting time;
- $\tau$: Constant to avoid smaller jobs from reaching very high Bounded Slowdown (fixed in 10 seconds).

Bounded Slowdown estimates the ratio between the total time that 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 jobs 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. Also, these jobs are less likely to demand more energy from future time steps (so, *BEASY* lets the energy to the priority ones). Fig. 4 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.

### C. Power compensations

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. Also, 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. Also, 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 1. Then, it calculates the energy difference $E_{diff}$ between the target and the estimated SoC at the end of the time window. For example, if the target

level is 50% and the estimated SoC is 51%, $E_{diff}$ is 1%. So, we need to reintroduce 1% of the battery. On the other hand, if the estimated last SoC is 49%, $E_{diff}$ is -1%. So, we need to reduce the usage by 1%. Therefore, it needs to reintroduce/remove the energy $E_{diff}$ before the end of the time window. When the compensation is positive ($E_{diff} > 0$), we can increase the speed of the servers or run more jobs. First, *BEASY* uses the $E_{diff}$ to speed up the running jobs. It increases the speed from the actual time step to the time step that the job finishes. This 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 ($E_{diff} < 0$), *BEASY* considers the estimated SoCs from Fig. 4. 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 not idle 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 $\rightarrow$ actual time step and violation time step $\rightarrow$ last time step). If reducing the usage from idle servers is not sufficient, we start to impact running servers (steps 3 and 4). Our idea is to impact them as far as possible from the actual step, but 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. *BEASY* kills jobs only when there is no power action possible (e.g., migrating power from the future) to maintain them running.

## V. Experiments

This section will present the results of our experiments. The idea is to compare the *BEASY* with the algorithms from the literature. *Follow plan* is an algorithm that follows the offline plan without changing it. *Power reactive* changes the server state according to the renewable power incoming. It uses power from the batteries when jobs are running and there is not enough renewable power. *Workload reactive* turns on the servers according to the job's arrival. When there is no job to start, it waits to turn off the servers using the DPM technique [23]. *Peak*, *Next*, and *Last* policies are proposed by [17]. Fig. 6 illustrates their main idea. The blue curve is the usage plan. 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). We included a new policy named *Workload*, which compensates where there is a higher difference between predicted demand and envelope. They use the default implementation of EASY backfilling (first sorting by bounded slowdown and second sorting by jobs size) to schedule. These policies compensate for the power changes, trying to approximate the target level. All implementations (*BEASY*, *Follow plan*, *Power reactive*, *Workload reactive*, *Peak*, *Next*, *Last*, and *Workload*) kill the jobs when the SoC goes bellow 20%.

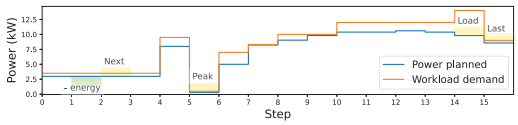


Fig. 6. Compensation policies [17].

We have divided the experiments into two parts. The first part aims to analyze the decisions in critical scenarios. To do so, we have taken two workloads from the Metracentrum dataset with the size of three days [24]. One of them has jobs arriving mainly on the first day, and the second one has jobs arriving mainly on the third day. Then, we have generated one profile with the renewable production of three days in the city of Toulouse from the Renewable Ninja website [25], [26]. After that, we have created an offline plan for each workload, using the same power production. We have created this plan using the MILP proposed by [16]. Finally, we introduce noise in the workloads and power production emulating the difference between offline and online. For the workload, we have applied a Gaussian noise in the job inter-arrival and duration. Considering the power production, we have taken two cases: worst-case and best-case. As presented in Fig. 4, we have two boundaries in a prediction. Best-case takes the higher power from the uncertainty interval, and the worst-case takes the lower power production from the uncertainty interval. Fig. 7 illustrates the power demand (workload) and production (profile) without and with noise. Combining both workloads with these power profiles, we have 4 cases to evaluate all the algorithms:

1) Profile best-case and workload in the beginning;

2) Profile best-case and workload in the end;
3) Profile worst-case and workload in the beginning;
4) Profile worst-case and workload in the end;

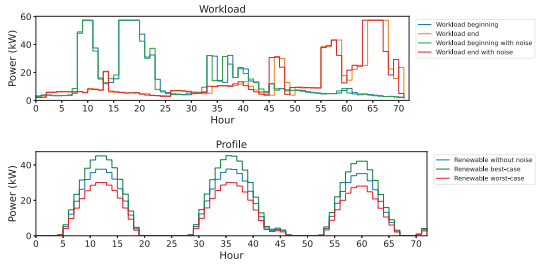


Fig. 7. The power demanded and production for the critical scenarios.

The second part of our experiments consists of taking ten tuples of workloads and profiles and creating a plan for each one. Fig. 8 shows the power production and demand for these cases. Then, we create ten new profiles and workloads for each tuple, adding different Gaussian noises. Therefore, we have 100 executions (ten plans with ten noises each). Differently from the first part, in this part, the power profile can have any value between lower and higher prediction boundaries for each time step. These experiments reveal the performance of our algorithm on 100 average cases. In both parts, we introduce uncertainty in the walltime given by the user in the same way as [27]. To do that, we equally divided the jobs into five groups. The jobs of the first group calculate the walltime by multiplying the (real) execution time by 5, the second by 3.333333333, the third by 2, the fourth by 1.428571429, and the last one by 1.111111111. This uncertainty complicates the *BEASY* scheduling decisions but it is more realistic. So, we introduced noises in jobs arrival, execution time, and walltime.

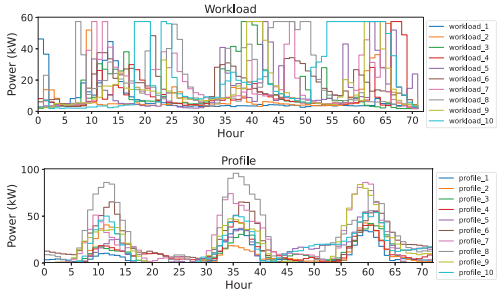


Fig. 8. The power demanded and production for different scenarios.

The following sections will analyze the results from three aspects: Jobs finished, real SoC at the end of the time window, and wasted energy. As mentioned before, the 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. Also, the algorithms must end the time window with SoC as close as possible to the planned. That means the algorithms can not "cheat" using more battery than planned. Finally, we verify how much energy was wasted. 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 maintaining idle servers turned on.

### A. Results - Critical scenarios

This section presents the results for each critical scenario, finishing with a global analysis. For each scenario, we show three graphs (for example, Fig. 9). First, a graph showing the impact on the jobs. This graph shows:

1) Finished: Jobs that finished completely 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 we arrive at the end of the time window and they are still running;
5) Killed: The killed jobs.

In the second graph, we demonstrate how far the state of charge is from the target level at the end of the time window. In the last graph, we illustrate the wasted energy.

*1) Profile best-case and workload in the beginning:* Fig. 9 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 majority of the jobs arrive 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. Also, the jobs not finished are not killed but postponed. It ends a little 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*).


Fig. 9. Results in the scenario with profile best-case and the workload in beginning.

*Follow plan* is the worst in jobs finished, killing several jobs. It ends with 20% more battery than the target, showing that it could use this power to avoid killing jobs. Considering the wasted energy, it is the second worst. These results highlight the need for adaptations in the plan. *Power reactive* has the worst wasted energy since it turns on servers even if it is not necessary. It also does not finish almost 10% of the jobs, using 20% more battery. *Workload reactive* execution kills 8.05% of jobs. This heuristic is too aggressive and puts all jobs to start as soon they arrive. So, it dries the battery too fast. Fig. 10 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. *Peak*, *Next*, *Last*, and *Workload* policies have good finished jobs, finishing with a good battery level and not bad wasted energy. However, they kill some jobs since they do not validate all the execution.

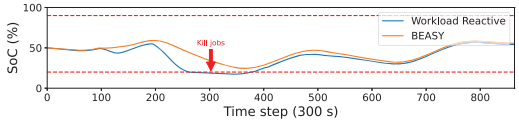


Fig. 10. Comparison between the state of charge of *Workload reactive* and *BEASY*. The red arrow indicates where *Workload reactive* kills some jobs.

*2) Profile best-case and workload in the end:* The second case is more complicated than the previous one. Here, the majority of the jobs arrive on the last day of the time window. So, the algorithms have a shorter time to schedule them. Fig. 11 illustrates the results. *Follow plan* has the second worst finished jobs and the worst killed jobs, even finishing with 20% more battery. The *Next* policy stays close to the plan since it makes the compensations in the next possible time step. It is possible to notice that it has a similar result, but kills fewer jobs than *Follow plan*. *Power reactive* still wastes more energy than the other algorithms and uses more battery. *Workload reactive* has perfect finished jobs and very good wasted energy. It uses 6% more battery than the target, which helps it to run more jobs. This is the best case for *Workload reactive* because it can stock a lot of energy in the first two days and expend all in the third day. This behavior explains why it finishes all the jobs and expends energy well. *Last* policy also has good results because it put all the power in the best moment to use it. However, it can dry the battery too fast and kill some jobs. *BEASY* is the second best in finished jobs and wasted energy. It does not kill any job, but it can not guarantee to finish some jobs, so it postpones them. Here, *BEASY* wasted 22.10% more energy than *Workload reactive*, but it wasted less than all the other heuristics.

*3) Profile worst-case and workload in the beginning:* 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 less energy coming from renewable. So, besides finding the best moment to place the jobs, they must adapt their power usage. Fig. 12 demonstrates the results. It is possible to notice that *Follow plan*, *Power reactive*, and

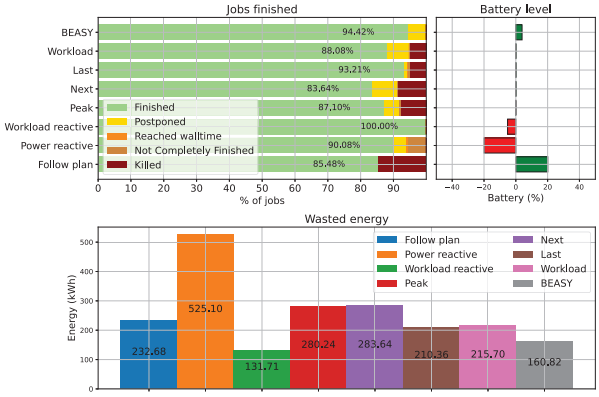


Fig. 11. Results in the scenario with profile best-case and the workload in end.

*Workload reactive* use a lot more battery (13.12%, 28.17%, and 17.60%, respectively). Even so, they are among the executions that killed more jobs (*Workload reactive* finishes with more than 20% of killed jobs).

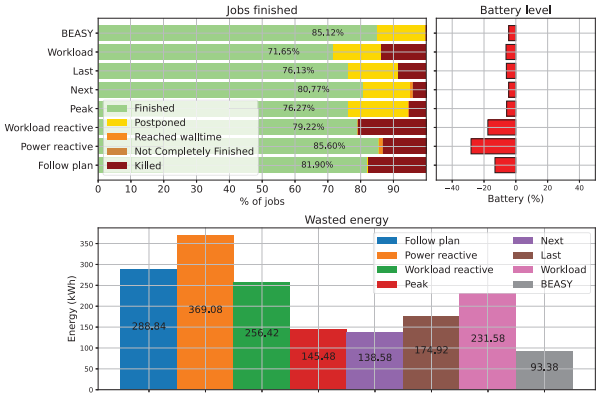


Fig. 12. Results in the scenario with profile worst-case and workload in the beginning.

*Power reactive* finishes more jobs than the other heuristics but kills more than 14% of jobs. *Last*, *Next* and *Peak* kill less than 10% of jobs, but only *Next* finishes more than 80% of the jobs. *BEASY* is the second best in finished jobs (very close to the best one), killing less than 1%. This result is outstanding in a scenario with less energy than predicted, showing the efficacity of the conservative approach in this case. Also, *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. Finally, this heuristic has the best battery level at the end of the time window but is very close to the *Peak*, *Next*, *Last*, and *Workload* policies.

*4) Profile worst-case and workload in the end:* The last case is with the profile worst-case and the jobs arriving in the end of the time window. Fig. 13 shows the results. Again,

*Follow plan*, *Power reactive*, and *Workload reactive* are far from the battery target level. *Follow plan* is the best in the finished jobs but the worst in killed jobs. *Workload reactive* has the second-best finished jobs but with almost 10% of killed jobs. Both results are explained because they used more battery than the other executions. Among the executions that respected the battery level, *BEASY* has the higher finished jobs and lower killed jobs. Also, it wastes less energy than all the other algorithms (19.70% less than the second-best, *Workload policy*). Again, it is an outstanding result in a critical scenario.

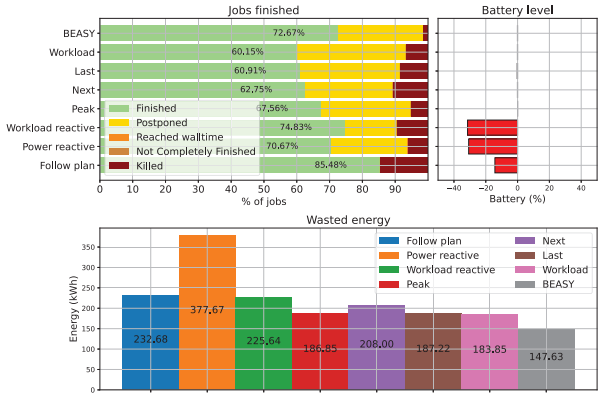


Fig. 13. Results in the scenario with profile worst-case and workload in the end.

*5) Conclusions of critical cases:* After presenting all results of the critical cases, this section consolidates our discussion about them. Regarding the number of killed jobs, *BEASY* has the overall best results by far. This heuristic can identify when it is possible to execute more jobs and when it is better to be conservative. Since our data center is in a power constraint environment, sometimes it is not possible to run everything. Postponing the jobs allows us to plan the next time window considering them. For example, the next time window could use more power 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. Also, 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. While the other algorithms (mainly *Power reactive* and *Workload reactive*) do not respect the battery level at the end, *BEASY* has good results arriving with close final SoC. Finally, *BEASY* generally wastes less energy than the other algorithms. This result is crucial mainly in cases with less energy to use.

## B. Results - Average cases

After discussing the critical cases, this section discusses the average cases. As mentioned in Section V, we have taken ten different profiles and workloads. Fig. 14 presents the results of the ten executions. Finished jobs, battery level, and wasted energy are the same metrics as presented in the previous section. The killed jobs graph considers the jobs that reach the walltime, were not completely finished, or were killed. Like in the critical cases, *BEASY* presents the lowest number of killed jobs (mean of 0.67%). Regarding finished jobs, it has the second-best result with a mean of 93.29%. *Workload reactive* is the best one in this metric with a mean of 97.27%. Fig. 15 compares the state of charge of *Workload reactive* and *BEASY* in the execution where *Workload reactive* killed several jobs. *Workload reactive* almost killed jobs around time step 400, but the SoC stays equal to or above 20%. At the end of the time window, it kills several jobs.

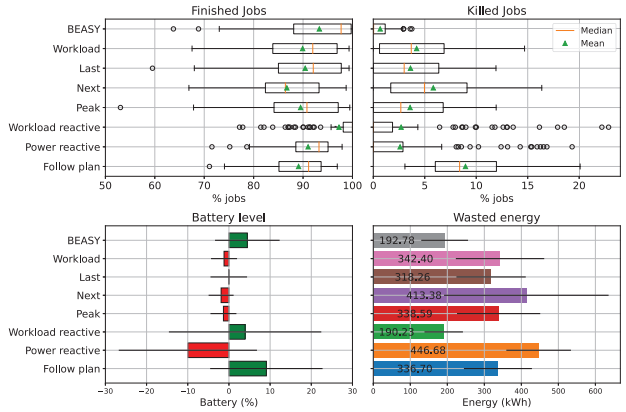


Fig. 14. Results in 100 executions.


Fig. 15. State of charge in one of the scenarios. The red arrow indicates where *Workload reactive* kills some jobs.

*Workload reactive* could not avoid the 20% threshold, resulting in several killed jobs. *BEASY* avoids this threshold. Comparing the number of jobs killed, *Workload reactive* has 19% of the executions above 5% of killed jobs, while *BEASY* has none. This result shows that *BEASY* can maintain the state of charge in control. On the other hand, *Workload reactive* has the two worst number of jobs killed among all executions. *Power reactive* has the third-best number of completed jobs but with also 19% of the executions above 5% of killed jobs. All the other executions have a higher number of killed jobs, finishing near 90%. The worst % of finished jobs is Next with an average of 86.72% and the worst killed jobs is Follow plan with 8.96%.

Considering the battery target level, *BEASY* has good levels since it is near the target level (average of 54.47%). Also, it does not vary a lot (standard deviation of 7.84%). 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 that they can not reduce

TABLE I
CONSOLIDATE AVERAGE RESULTS IN EVERY SCENARIO.

| Scenario | Metric | Follow plan | Power reactive | Workload reactive | Peak | Next | Last | Workload | 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 |

the killed jobs as much as *BEASY*. *Workload reactive*, *Power reactive*, and *Follow plan* have a large target level variance. Fig. 15 shows an example of how badly *Workload reactive* manages the battery. *Workload reactive* kills several jobs when the battery is lower than 20%. *BEASY* avoids this threshold and could maintain the jobs running. *Workload reactive* and *Follow plan* have more battery than the target (average of 53.92% and 59.09%, respectively) but with a higher variation (standard deviation of 18.50% and 13.64%). *Power reactive* has the worst result, with an average of 40.03% and a standard deviation of 16.78%. 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*. All these results show that *BEASY* has excellent all-around performance. It can balance power and IT decisions, expending energy wisely among the different simulations.

Table I presents a consolidation of all results over the different tested scenarios. The top-3 results on each metric for each scenario are highlighted in green and the bottom-3 in red. Killed jobs are: killed jobs + reach the walltime + not completely finished. For SoC, we consider 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. It is the third-best in the finished job one time. However, in the same case, the first (*Follow plan*) and second-best (*Workload reactive*) finished job metrics have the worst and third-worst killed jobs. In the cases where *BEASY* has the second-best finished jobs metric, the algorithm with the best one has bad battery management. For example, *Workload reactive* is the best one in profile best-case and workload in end but has the second-worst real SoC at the end of the time window. In 100 average cases, *Workload reactive* appears in third place in SoC, but we saw in Fig. 14 that it has a large variance in SoC. Excluding *BEASY*, all the other algorithms have at least one result at the bottom-3. It is possible to notice that just following the offline plan (*Follow plan*) is not a good option, with several metrics at the bottom-3. Reacting to the power (*Power reactive*) is even worst. *Workload reactive* has some good results, and, due to the DPM technique, it is not too energy aggressive. However, it can not manage the storage well, sometimes killing jobs and sometimes drying the battery. Regarding the policies, the *Last* has intermediate results, but with no result as the best one.

## VI. CONCLUSION

A renewable-only data center introduces several elements, such as batteries, hydrogen, wind turbines, and solar panels. It demands a plan for the following days using workload and weather predictions. However, just following this plan may not be sufficient. This work presented a heuristic for online adaptations to change an offline plan, aiming to improve QoS and deal with power fluctuations. The results demonstrated that simply following the offline plan or only reacting to the online events is not enough due to the variance in workload and renewable production. *BEASY* had an excellent overall wasted energy, the lowest number of killed jobs, and in top-3 finished jobs compared to the state of art algorithms. It achieves these outstanding results with the state of charge at the end of the time window near the target level. Future work will include a more complete job description (I/O, network communication, etc). Also, we will evaluate *BEASY* with new application types (e.g., services, streaming) and the impact of reintroducing killed jobs for execution. Finally, we can introduce energy flexibility, allowing the battery to finish around the target level (e.g., $\pm 5\%$).

## References


[1] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. Blair, and A. Friday, "The climate impact of ict: A review of estimates, trends and regulations," 2021.

[2] Q. Zhou, M. Xu, S. S. Gill, C. Gao, W. Tian, C. Xu, and R. Buyya, "Energy efficient algorithms based on vm consolidation for cloud computing: comparisons and evaluations," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 489–498.

[3] M. A. Khan, A. P. Paplinski, A. M. Khan, M. Murshed, and R. 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)*. IEEE, 2018, pp. 105–114.

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

[5] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates," *Science*, vol. 367, no. 6481, pp. 984–986, 2020. [Online]. Available: https://science.sciencemag.org/content/367/6481/984

[6] S. Kwon, "Ensuring renewable energy utilization with quality of service guarantee for energy-efficient data center operations," *Applied Energy*, vol. 276, p. 115424, 2020.

[7] V. Venkataswamy, J. Grigsby, A. Grimshaw, and Y. 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*. Springer, 2023, pp. 108–130.

[8] W. Liu, Y. Yan, Y. Sun, H. Mao, M. Cheng, P. Wang, and Z. Ding, "Online job scheduling scheme for low-carbon data center operation: An information and energy nexus perspective," *Applied Energy*, vol. 338, p. 120918, 2023.

[9] J. Yuan, W. Zhang, Y. Zhou, S. Chen, and C. Gao, "Optimal scheduling of data centers considering renewable energy consumption and temporalspatial load characteristics," in *2022 Power System and Green Energy Conference (PSGEC)*. IEEE, 2022, pp. 283–288.

[10] P. Wiesner, D. Scheinert, T. Wittkopp, L. Thamsen, and O. 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*. Springer, 2022, pp. 218–232.

[11] H. He, H. Shen, Q. Hao, and H. Tian, "Online delay-guaranteed workload scheduling to minimize power cost in cloud data centers using renewable energy," *Journal of Parallel and Distributed Computing*, vol. 159, pp. 51–64, 2022.

[12] S. Caux, P. Renaud-Goud, G. Rostirolla, and P. 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)*. IEEE, 2019, pp. 136–143.

[13] F. Hu, X. Quan, and C. 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*, 2018, pp. 134–141.

[14] L. Yu, T. Jiang, Y. Cao, and Q. Qi, "Joint workload and battery scheduling with heterogeneous service delay guaranteesfor data center energy cost minimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1937–1947, 2014.

[15] N. Sharma, S. Barker, D. Irwin, and P. Shenoy, "Blink: managing server clusters on intermittent power," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 185–198.

[16] M. Haddad, J. M. Nicod, C. Varnier, and M.-C. 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)*. IEEE, 2019, pp. 1–8.

[17] I. F. de Nardin, P. Stolf, and S. 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)*. IEEE, 2022, pp. 305–314.

[18] L. Naveen and H. Mohan, "Atmospheric weather prediction using various machine learning techniques: a survey," in *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, 2019, pp. 422–428.

[19] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Cluster Computing*, vol. 23, no. 4, pp. 2399–2424, 2020.

[20] B. Xu, A. Oudalov, A. Ulbig, G. Andersson, and D. S. Kirschen, "Modeling of lithium-ion battery degradation for cell life assessment," *IEEE Transactions on Smart Grid*, vol. 9, no. 2, pp. 1131–1140, 2016.

[21] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *IEEE transactions on parallel and distributed systems*, vol. 12, no. 6, pp. 529–543, 2001.

[22] J. Lelong, V. Reis, and D. 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*. Springer, 2018, pp. 43–61.

[23] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 8, no. 3, pp. 299–316, 2000.

[24] D. Klusáček, Š. Tóth, and G. Podolníková, "Real-life experience with major reconfiguration of job scheduling system," in *Job scheduling strategies for parallel processing*. Springer, 2015, pp. 83–101.

[25] S. Pfenninger and I. Staffell, "Long-term patterns of european pv output using 30 years of validated hourly reanalysis and satellite data," *Energy*, vol. 114, pp. 1251–1265, 2016.

[26] I. Staffell and S. Pfenninger, "Using bias-corrected reanalysis to simulate current and future wind power output," *Energy*, vol. 114, pp. 1224–1239, 2016.

[27] S. Takizawa and R. 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*, 2020, pp. 169–178.