Green Scheduling on the Edge

Joachim Cendrier, Rajini Wijayawardana, Anne Benoit, Yves Robert, Frédéric Vivien, Andrew A Chien

Green Scheduling on the Edge

Joachim Cendrier, Rajini Wijayawardana, Anne Benoit, Yves Robert, Frédéric Vivien, Andrew A. Chien

RESEARCH REPORT

N° 9580

March 2025

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9580--FR+ENG

Green Scheduling on the Edge

Joachim Cendrier*, Rajini Wijayawardana†, Anne Benoit‡,

Yves Robert§, Frédéric Vivien¶, Andrew A. Chien‖

Project-Team ROMA

Research Report n° 9580 — March 2025 — 27 pages

Abstract: This work aims at designing and evaluating scheduling algorithms that minimize carbon cost on edge platforms. When a job is released to some edge server, difficult scheduling questions arise: should the job be executed on that server? If yes, when? If no, which other edge server should the job be transferred to? Typically, jobs are submitted online, and have a deadline to enforce. On-line scheduling problems are already difficult without accounting for different energy sources, so one should not expect any optimal solution. Still, an important research goal is to revisit standard algorithms such as *Earliest Completion Time* (ECT) and *Earliest Deadline First* (EDF) in order to design and evaluate carbon-aware variants. This paper introduces several new algorithms that use sophisticated scheduling policies to efficiently decrease carbon cost; these algorithms maximize the use of green intervals both on local and remote edge servers, by re-evaluating previous decisions whenever needed to accommodate newly released jobs. We provide a comprehensive simulation campaign based on actual platform/job data and carbon traces and report an average gain of 48% over the standard approaches.

Key-words: Edge computing, green scheduling, carbon cost.

Authors' emails: {joachim.cendrier,anne.benoit,yves.robert,frederic.vivien}@inria.fr, {rajini,acchien}@uchicago.edu.

* ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
† University of Chicago, Chicago, IL, USA
‡ ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, and Institut Universitaire de France, and Institute for Data Engineering and Science (IDEaS), Georgia Tech, USA
§ ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
¶ ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
‖ University of Chicago, Chicago, IL, USA

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

Ordonnancement vert sur l'informatique en périphérie

Résumé : Ce travail vise à concevoir et à évaluer des algorithmes d'ordonnancement qui minimisent le coût du carbone sur les plates-formes périphériques (ou *edge computing*). Lorsqu'une tâche est confiée à un serveur périphérique, des questions difficiles d'ordonnancement se posent : doit-on l'exécuter sur ce serveur ? Si oui, quand ? Si non, vers quel autre serveur périphérique le travail doit-il être transféré ? En général, les tâches sont soumises en ligne et ont une date limite à respecter. Les problèmes d'ordonnancement en ligne sont déjà difficiles à résoudre sans tenir compte des différentes sources d'énergie, de sorte qu'il ne faut pas s'attendre à une solution optimale. Néanmoins, un objectif de recherche important consiste à réexaminer les algorithmes standard tels que *Earliest Completion Time* (ECT) et *Earliest Deadline First* (EDF) afin de concevoir et d'évaluer des variantes qui prennent en compte le coût carbone. Ce rapport de recherche présente plusieurs nouveaux algorithmes qui utilisent des politiques d'ordonnancement sophistiquées pour réduire efficacement le coût carbone. Ces algorithmes maximisent l'utilisation des intervalles verts à la fois sur les serveurs locaux et distants, en réévaluant les décisions antérieures chaque fois que cela est nécessaire pour prendre en compte les nouvelles tâches qui arrivent dans le système. Nous effectuons une campagne de simulation complète basée sur des données réelles de plateforme/travail et des traces de coût carbone, et nous rapportons un gain moyen de 48% par rapport aux approches standard.

Mots-clés : Informatique en périphérie, ordonnancement vert, coût carbone.

Contents

1 Introduction

With growing concern about climate change and the corollary desire to make computing "greener" (e.g., reduce its carbon footprint), there is much interest in powering computing resources with renewable energy, and aligning the use of computing resources with when renewable energy is available. The progress in decarbonizing the power grid has reduced the carbon emissions of datacenters [31, 18] and the results have been shown to vary heavily based on location (what renewables are available, weather, etc.) and use (load types, load competition). These variations create opportunities for sophisticated scheduling across time (temporal load shifting) and space (geographic load shifting) to reduce the carbon emissions associated with computing [35, 15, 40, 20, 21, 43, 44, 9, 32, 42, 19]. But the rapid growth in computing is not only happening in the cloud, but also at the edge where the rise of intelligent city, consumer AI services, hierarchical machine-learning models are increasingly deployed [13, 33]. Because of their distributed deployment, varying local consumer use, and even varying weather factors, edge resources often have excess computing capacity. From the perspective of "greening" computing use, the situation at the edge is perhaps even more complex and varied. Choice of power supplier, different on site renewables (solar or wind), competing local electrical load, competing compute load, and more, create a landscape in which the carbon-emission content of power locally consumed varies from 0 to 100's grams of CO2e/kWh.

To address this need, this work aims at designing and evaluating scheduling algorithms that minimize carbon cost on edge platforms. When a compute job is released into the system, difficult questions include: should the job be executed on that server? If yes, when? If no, which other edge server should the job be transferred to? Typically, jobs are submitted online, and they have a deadline to enforce. This means that delaying the start of the execution of a job to benefit from a green energy source later is a risky bet if more jobs are to be submitted soon. Online scheduling problems are already difficult without accounting for different energy sources, so one should not expect any optimal solution. Still, an important research goal is to revisit standard algorithms such as *Earliest Completion Time* (ECT) and *Earliest Deadline First* (EDF) in order to design and evaluate carbon-aware variants. In this introductory section, we provide a high-level overview of the problem that we are addressing and a brief description of the novel algorithms that we have designed to minimize carbon cost.

Edge platforms typically consist of a completely connected set of edge servers, complemented by a powerful but carbon-costly Cloud resource, as illustrated in Figure 1. The edge servers are identical but have different carbon profiles. A carbon profile is defined as a continuous set of alternating *green* and *brown* intervals; as the name indicates, computing during a *green* interval has no carbon cost, while computing during a *brown* interval incurs some cost per second of execution. Jobs are submitted online to a local edge server, called their *origin*. Each job has several parameters: length, release time, deadline, and data volume. When a job is submitted to its *origin*, there are three possibilities:

- The job is executed locally on its *origin*. The carbon cost is then proportional to the time spent executing during brown intervals (while execution during *green* intervals comes for free);
- The job is transferred to another edge server. Then some transfer cost (proportional to the volume of data transferred) is added to the cost of the execution on the other server;
- The job is delegated to the Cloud, typically with a high carbon cost both for transfer and for execution, but also with a much faster total completion time than when using edge servers.

The first rule of the game is that each job should meet its deadline, which motivates the addition of the Cloud to the platform; we suppose that there is always enough time to execute

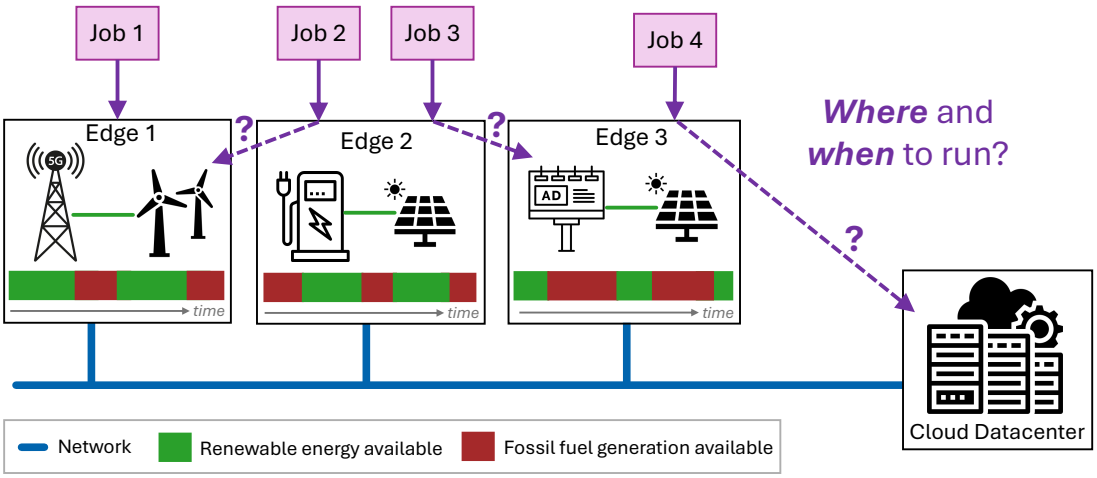

Figure 1: The edge green scheduling problem: minimizing carbon emissions from computing, whilst balancing job deadlines, carbon costs of communication, and distributed resources. All in the presence of varying carbon emissions from power.

a job on the cloud between its release time and deadline. The second rule of the game is that jobs can be frozen but cannot be migrated during execution: once the execution has started on some edge server, it has to finish on that server, but the execution can be temporarily frozen to benefit from forthcoming *green* intervals. Similarly, once the decision to delegate a job to the CLOUD is taken, it cannot be reversed; but no freezing is needed since the CLOUD is always carbon-costly. We detail all model and application parameters in Section 3.

We point out that the addition of the CLOUD is a sine-qua-non to use total carbon cost as a metric. To see why, consider an overloaded system: some jobs will not be able to match their deadline on the edge servers. How do we account for (the carbon cost) of these failed jobs? The addition of the CLOUD nicely answers the question: some jobs will be executed at a very high cost on the CLOUD, but at the end of the day all jobs will have executed successfully, and total carbon cost becomes a fair metric.

Now, for scheduling algorithms, the de-facto greedy standard is *Earliest Completion Time* (ECT): schedule each job when it is released, and assign it the edge server that will allow for the earliest completion time, given already taken decisions. Note that: (i) if the job is not assigned on its *origin*, transfer times are taken into account into the completion time; and (ii) if the job cannot match its deadline, it is executed on the CLOUD. ECT does not account for *green* and *brown* intervals to make decisions, and one can envision several variants to explore. A first heuristic is to give priority to *locality* and assign a job to its *origin* server, but while aiming at using as much *green* periods as possible before the job deadline. A second heuristic is to give priority to carbon cost and to assign the job on the edge that maximizes the green fraction of its execution, if transfer costs are not too high, and still ensuring that the job deadline is met.

In addition, more sophisticated algorithms like *Earliest Deadline First* (EDF) may revisit previous scheduling decisions: upon release of a new job, the priority of all the jobs that have been scheduled already but have not actually started execution yet, can be re-evaluated, and the priority ordering used by the scheduler to map jobs can be updated. Several combinations can be designed, and we refer to Section 4 for a complete description.

Altogether, our first contribution is a realistic yet tractable model for this important schedul-

ing problem. Our second contribution is to assess the complexity of several problem instances. Our third contribution is the design of an optimal algorithm for the offline version of the problem with a single edge server, which is used as a key building block for the general online problem with multiple servers. Our fourth contribution is the design and analysis of novel scheduling algorithms that dramatically decrease carbon cost when compared to standard carbon-unaware competitors. Indeed, we evaluate the performance of several carbon-aware algorithms on a variety of problem instances arising from experimental traces and report that an important fraction of carbon consumption can be avoided. This is good news at a time where computing consumes a larger and larger fraction of energy resources!

The rest of the paper is organized as follows. Section 2 surveys related work. In Section 3, we detail the framework for platforms, jobs, resource variability and optimization objectives, and we provide complexity results. In Section 4, we introduce our novel scheduling algorithms, whose experimental assessment is conducted in Section 5. Finally, we give concluding remarks and hints for future work in Section 6.

2 Related work

A range of applications of edge computing is given in [5], and these include e-health, disaster recovery, autonomous vehicles or flying drones. Although no specific model is given, [5] provides a list of objectives and constraints that have been studied, such as the delay, the bandwidth, the energy, QoS-assurance, etc. Some challenges of edge computing are also discussed in [25]. An extensive body of research investigates workload scheduling for reducing carbon emissions and maximizing green energy use. This section examines key techniques employed in prior studies.

Temporal and spatial workload shifting can provide significant carbon savings by exploiting variations in carbon intensity across time and geographic location. Temporal shifting, which has been extensively studied in prior literature [35, 15, 16, 40, 20, 30, 21], leverages the delay-tolerant properties of workloads to shift computation to periods of lower carbon intensity. Prior studies typically enforce constraints on the maximum allowable delay, most often 24 hours from the job's submission. This formulation of job deadlines disproportionately affects shorter-duration jobs.

Shifting compute workloads to geographic regions with lower carbon intensity, based on time zone differences and regional fuel mixes, has been studied in [43, 44, 9, 32, 42]. A combination of both temporal and spatial shifting techniques for lower carbon emissions is studied in [26, 19, 38]. Our work utilizes both temporal and spatial workload shifting techniques. We impose tight deadlines on jobs, allowing them temporal flexibility. Additionally, jobs can be spatially transferred to other edge servers before the start of their execution, to exploit green availability in other edge nodes or to ensure that job deadlines are met.

Resource scaling and job speedup is another common load adaptation technique that increases energy consumption when abundant renewable generation is available. GreenPar [22] increases the resource allocations of jobs and uses job speedup profiles to reduce runtimes, allowing them to maximize green energy consumption while meeting SLAs. CarbonScaler [20] dynamically scales applications in response to carbon intensity signals. [41] proposes a scheduler that uses DVFS to allocate jobs in cloud data centers, reducing energy consumption.

Several studies have investigated strategies for managing data centers powered by renewable energy. Some leverage batteries to store surplus renewable energy and discharge it when renewable generation is unavailable, allowing jobs to complete using green energy [29, 15, 22]. Others integrate supplemental power sources, such as the electric grid and backup generators, to maintain stable resource capacity [15, 22, 17, 16]. While our evaluation assumes only a grid connection, our algorithms are generalizable to any power model.

Predictions of renewable energy availability could inform scheduling decisions on edge nodes. Previous studies have used statistical and machine learning techniques to predict grid-level renewable availability to inform scheduling decisions [16, 17, 22, 1]. Radovanović et al. [35, 19] assume exact knowledge of renewable availability and compute carbon-aware Virtual Capacity Curves (VCCs) for Google's datacenter clusters. We also assume full knowledge of future renewable availability and make scheduling decisions based on the system's current and future state, reassigning jobs if needed.

Scheduling independent jobs on edge-cloud systems has been studied with various platform models; a good overview is provided by [28], which includes multiple cloud-assisted edge servers. While energy constraints are present in some models, to the best of our knowledge, no work has yet focused on edge servers operating under different carbon-cost intervals. On the theoretical side for edge-cloud servers without carbon-aware considerations, [8] derived approximation algorithms for a simple platform with 1 edge server and 1 cloud, while [8] studied the complexity of minimizing the maximum stretch in an online setting.

Checkpointing is widely used for resilience in HPC and other workflow and batch-oriented workloads [10, 23]. It involves periodically saving the system's state, enabling recovery in the event of a failure. However, checkpointing introduces overhead due to data storage and retrieval. In contrast, our work relies on the pause and resume technique [27], which allows job execution to be temporarily halted when green energy is no longer available and resumed without the need for continuous state-saving mechanisms. We acknowledge that for our platform model, the decision to allow pause-resume but neither migration once a job is started nor checkpointing, was influenced by [2, 3], where it is shown that allowing migration does not affect the efficiency of the algorithms that minimize the average response time or the average stretch[1].

3 Framework

This section first details the framework (target platform in Section 3.1, and applications in Section 3.2), the scheduling rules (Section 3.3), and the objective function (Section 3.4). Then, we provide complexity results in Section 3.5, proving that the problem is difficult even on a single edge server.

3.1 Target platform

The platform consists of a completely connected set of n edge servers e_i, $1 \leq i \leq n$, each with identical speed. W.l.o.g, we assume unit speed for the edge servers. The execution horizon is a (long) interval of T seconds, which is partitioned into *green* and *brown* intervals on each edge. Specifically, edge e_i has u_i intervals:

$$I_1^{(i)} = [0, \tau_1^{(i)}[, I_2^{(i)} = [\tau_1^{(i)}, \tau_2^{(i)}[, \ldots, I_{u_i}^{(i)} = [\tau_{u_i-1}^{(i)}, \tau_{u_i}^{(i)}[, \text{ where } \tau_{u_i}^{(i)} = T.$$

Each interval $I_j^{(i)} = [\tau_{j-1}^{(i)}, \tau_j^{(i)}[$ is either *green*, with carbon cost 0, or *brown*, with carbon cost k per second. As stated in Section 2, we assume full a priori knowledge of the green and brown intervals.

The bandwidth between two edge servers is b_{trans}, while the carbon cost of the transfer is k_{trans} per second.

The platform is complemented by a powerful Cloud server with high processing capacity, which has the capacity to process all submitted jobs in parallel.

[1] The stretch of a job is the ratio of the response time by its duration. The response time is the time elapsed from release to completion.

3.2 Jobs, execution times and carbon cost

Jobs are submitted online, at any time in $[0, T[$, for a total of m jobs during the whole horizon. Job J_j, $1 \leq j \leq m$ is submitted to its *origin* edge server o_j and is characterized by several parameters and variables:

- parameter: release time r_j;
- parameter: deadline d_j;
- parameter: duration on an edge ℓ_j, expressed in seconds since we have unit edge speed;
- parameters: data/communication input volume f_j^{in} and output volume f_j^{out}, expressed in bits;
- variable: execution duration $t_j^{(local)}$ and carbon cost $C_j^{(local)}$ when executed locally on its *origin* o_j, which both depend upon the final schedule;
- variable: execution duration $t_j^{(transfer)}$ and carbon cost $C_j^{(transfer)}$ when transferred to and executed on another edge server $e_i \neq o_j$, which both depend upon the final schedule;
- parameters: execution time $t_j^{(cloud)}$ and carbon cost $C_j^{(cloud)}$ when delegated to the cloud.

In a nutshell, execution durations and carbon costs may vary on the edge platform as a function of the schedule, for both a local and a remote execution. However, because the Cloud can process all jobs in parallel, $t_j^{(cloud)}$ and $C_j^{(cloud)}$ are constants that only depend on the job, not on the schedule. Note that we need to enforce

$$t_j^{(cloud)} \leq d_j - r_j \text{ for each job } J_j, 1 \leq j \leq m \tag{1}$$

to ensure that all submitted jobs can be processed successfully on the Cloud (if needed).

As for carbon costs on the edge platform, they are computed as follows:

- Local execution: job J_j is executed on its *origin* edge server o_j. Assume that the job starts executing at time $t \geq r_j$ and completes its execution at time t', where $t + \ell_j \leq t' \leq d_j$. While the duration is $t_j^{(local)} = t' - t$, the job actually executes during exactly ℓ_j seconds within the interval $[t, t'[$ and is frozen the rest of the time. Recall that the job can be frozen one or several times but cannot be migrated once started. Letting $\alpha_j \leq 1$ be the amount (in seconds) of the execution time ℓ_j spent in *brown* intervals, the carbon cost of the execution is
$$C_j^{(local)} = \alpha_j k$$
and the amount spent in *green* intervals comes for free. Note that the edge server must be free during the whole duration interval $[t, t'[$ and cannot accommodate any other job until completion time t'.
- Remote execution: job J_j is transferred to another edge server $e_i \neq o_j$. Assume that the job starts executing at time t and completes its execution at time t'. Since the execution starts at time t, the scheduler initiates the input communication from o_j to e_i ALAP (As Late As Possible), i.e., at time $t - t_j^{(comm-in)}$, where $t_j^{(comm-in)} = \frac{f_j^{in}}{b_{trans}}$ is the time needed for that communication. We must have $t - t_j^{(comm-in)} \geq r_j$. Note that we assume that there is no delay nor contention for communications. Similarly to a local execution, the job executes during exactly ℓ_j seconds within the interval $[t, t'[$, and we let $\alpha_j \leq 1$ be the amount (in seconds) of the execution time ℓ_j spent in *brown* intervals on edge e_i. The return communication is initiated ASAP (As Soon As Possible) at time t' and lasts for $t_j^{(comm-out)} = \frac{f_j^{out}}{b_{trans}}$ seconds, without delay nor contention. We must have $t' + t_j^{(comm-out)} \leq d_j$. The whole duration is $t_j^{(transfer)} = t' - t + t_j^{(comm-in)} + t_j^{(comm-out)}$ and the carbon cost is proportional to the volume of data transferred:
$$C_j^{(transfer)} = \alpha_j k + (f_j^{in} + f_j^{out}) k_{trans}.$$

As before, note that the edge server e_i must be free during the whole duration of interval $[t, t'[$ and cannot accommodate any other job until completion time t'. But e_i can terminate the execution of another job during the input communication, as well as initiate the execution of another job during the return communication.

We observe that the model is equipped with many parameters but includes some simplification regarding communications between edge servers. In a nutshell, the communication model is unbounded multiport [4, 24], which means that all communications can take place in parallel without contention. It is easy to change the communication model and, for instance, to move to a bounded multiport model (where the network card of a server is the bottleneck) or even to a one-port model including latencies, but this requires the introduction of several additional parameters and complicates the description of the scheduling algorithms.

It remains to describe how the scheduler decides where to map each incoming job and at which time to start its execution. This is a complicated process that we describe below.

3.3 Scheduling rules

Online scheduling problems are known to be difficult. The release of a new job may lead the scheduler to re-evaluate its decisions. For short, the release of a new job is called an *event*. At each event, there are two possible states for the jobs that have been previously submitted and whose execution is not yet complete:

- state *started*: execution has already started: then the scheduler cannot re-assign the job (no migration in our model) but can change the current plan for the remaining of the execution, which includes changing the use of *green* and *brown* intervals, and idling (freezing) the job. The aim of re-evaluating the current plan for a job on a server is to benefit from a larger *green* portion globally for all jobs scheduled on that server. Of course, any change of plan must enforce that the totality of the job is executed before its deadline. Note that at any instant, at most one job can be *started* on each server;
- state *planned*: execution is scheduled but has not started yet: then the scheduler can change the whole assignment. In addition to enforcing that the totality of the job is executed before its deadline, the scheduler must account for the input communication in case of a transfer to another edge.

Basically, the scheduler takes ALAP decisions. For instance, say there was an event at time t_1, namely the release of some job J_j and that the scheduler plans to execute it at time t_2 on a remote edge $e_i \neq o_j$. Hence, at time t_1, job J_j is *planned*; the current schedule has the input communication of job J_j from o_j to e_i start at time $t_2 - t_j^{(comm-in)}$, ALAP to start execution at t_2. Now say there is a new event (a new job $J_{j'}$ released) at time t_3 where $t_1 < t_3 < t_2$. At time t_3, the state of job J_j is still *planned*, even if the input communication has been initiated, i.e., if $t_3 \geq t_2 - t_j^{(comm-in)}$. The scheduler may well re-evaluate its plan and decide to assign job J_j somewhere else, for instance to another edge than e_i and o_j, or even to the Cloud. Note that J_j is still *planned* at time t_3 unless it starts its execution immediately, either locally or on the Cloud. Note also that if the input communication of job J_j from o_j to e_i has started at time t_3, then it will (uselessly) proceed until completion, and the corresponding carbon cost will be paid for.

In this scenario, J_j may be re-assigned at time t_3 upon release of job $J_{j'}$, but it is not the only one! The scheduler may well re-evaluate its whole plan and re-assign all the jobs that are in state *planned*. As mentioned, it can also update the execution of the jobs that are in state *planned* and scheduled to execute on some edge server to try and benefit from a larger *green* portion globally on that server. Finally, a job J_j *planned* to execute on the Cloud will always be started ALAP, i.e., at time $d_j - t_j^{(cloud)}$.

Admittedly, the rules are complicated! But the intuition is simple: take any decision ALAP in order to better be able to react to new job releases, and re-evaluate the schedule at each event, completely for *planned* jobs, and partially for *started* jobs, which are pinned to their resource but whose execution may be shifted if it leads to a lower total cost for all jobs.

Now, to design a scheduling algorithm, we only need to detail the scheduling policy: at each event, the scheduler will order the jobs, both *planned* and *started*, according to some priority rule, and will schedule them according to some criterion. We discuss various priority ordering and assignment criteria in Section 4.

3.4 Objective function

The objective is to minimize the total carbon cost, with the constraint that all submitted jobs must have been successfully executed before the end of the horizon T. Owing to the addition of the CLOUD and to Equation (1), there is always a solution to this problem.

In practical scenarios, it is likely that executing on the CLOUD will be faster but more carbon costly for all jobs ($t_j^{(cloud)} \leq t_j^{(local)}$ and $C_j^{(cloud)} \geq C_j^{(local)}$ for each job J_j). But if the edge platform is overloaded, some jobs will have to be delegated to the CLOUD, and it is crucial to determine which ones.

3.5 Complexity

The offline version of the edge scheduling problem resembles a classical scheduling problem, namely scheduling with limited machine availability [36]. The main difference is that the objective is to minimize carbon cost instead of makespan. The problem is difficult even with a single edge server:

Definition 1. *We let ONEEDGESCHED be the problem of minimizing the total carbon cost with m jobs and a single edge server (complemented with a CLOUD server).*

Theorem 1. *With a single edge, the decision problem ONEEDGESCHEDDEC associated to ONEEDGESCHED (with a bound on the total carbon cost C_{max}) is NP-complete in the strong sense.*

Proof. First, ONEEDGESCHEDDEC belongs to NP, a certificate being the schedule and carbon cost of each job. For the completeness, consider an instance $\mathcal{I}_1$ of the 3-PARTITION problem, which is NP-complete in the strong sense [14]: given $3n$ positive integers a_i, $1 \leq i \leq 3n$ and a positive bound B with $\sum_{i=1}^{3n} a_i = nB$, does there exist a partition of these $3n$ integers into n triplets, each of sum B? We consider the following instance $\mathcal{I}_2$ of ONEEDGESCHEDDEC with one edge server e_1:

- the execution horizon is the interval $[0, T[$ with $T = 3nB$, partitioned into 2 intervals. The first interval $I_1^{(1)} = [0, 2nB[$ is of length $2nB$ and is *green*. The second interval $I_2^{(1)} = [2nB, 3nB[$ is of length nB and is *brown* with carbon cost $k = 1$ per second
- there are $m = 4n$ jobs which we denote as follows for the intuition: there are $3n$ *flexible* jobs $\mathcal{A}_j$ of length a_j, release date 0, and deadline T, for $1 \leq j \leq 3n$; in addition, there are n *rigid* jobs $\mathcal{B}_j$ of length B, release date $(2j-1)jB$ and deadline $2jB$ for $1 \leq j \leq n$. The *rigid* job $\mathcal{B}_j$ must execute during the interval $[(2j-1)B, 2jB[\subset I_1^{(1)}$: no flexibility and zero carbon cost. On the contrary, jobs $\mathcal{A}_j$ can freely be shifted around the whole scheduling window due to their flexibility
- we let $C_{max} = 0$. Finally, we let the carbon cost of the CLOUD be $C_j^{(cloud)} = 1$ for any job $\mathcal{A}_j$ or $\mathcal{B}_j$, so that no job can be executed on the cloud while matching the bound.

The size of $\mathcal{I}_2$ is polynomial (even linear) in the size of $\mathcal{I}_1$. Obviously, there is a solution to $\mathcal{I}_1$ if and only if the $3n$ jobs $\mathcal{A}_j$ perfectly fit into the n sub-intervals $[0, B[$, $[2B, 3B[$, $[4B, 5B[$, $\ldots$, $[(2n-2)B, (2n-1)B[$ left available by the rigid jobs, in which case they can be scheduled with zero total carbon cost. Otherwise, some jobs $\mathcal{A}_j$ must (at least partially) execute during the *brown* interval $I_2^{(1)}$, with a non-zero cost. This is equivalent to say that there is a solution to $\mathcal{I}_1$ if and only if $\mathcal{I}_2$ has a solution of cost C_{max}. □

Corollary 1. *Unless P=NP, there is no constant approximation algorithm for* OneEdgeSched.

Proof. Assume that there exists a λ-approximation to OneEdgeSched and consider the proof of Theorem 1. Starting with instance $\mathcal{I}_1$ of 3-PARTITION, we construct the same instance $\mathcal{I}_2$ but without any a priori bound on the total carbon cost. There is a solution to $\mathcal{I}_1$ if and only if the total cost is 0 in $\mathcal{I}_2$, which the λ-approximation must find because it cannot be worse than λ times the optimal cost. Hence P=NP, which contradicts the assumption. □

Even though the problem is already difficult with a single edge, we propose in the next section heuristic algorithms targeting the original problem with several edge servers, denoted by EdgeSched.

4 Algorithms

This section is devoted to the design of algorithms to solve the EdgeSched problem. In Section 4.1, we present a set of greedy algorithms that schedule jobs as soon as possible when they are released, and do not re-evaluate the choices made previously upon a new release. Then in Section 4.2, we introduce an offline algorithm to schedule an ordered set of jobs on a single edge server, and we prove its optimality. In Section 4.3, we use this algorithm as a building block to design more sophisticated algorithms, some of them re-evaluating decisions taken for jobs that are either *started* (but not yet completed) or *planned*.

Throughout this section, for the study of the complexity of the algorithms, we denote by l the average job length, and by a the average elapsed time between the release date and the deadline of a job. Each job spans over an average of pa intervals, where $p = \frac{\sum_{i=1}^{n} u_i}{nT}$ is the inverse of the average interval length.

4.1 Greedy baseline algorithms

The five algorithms presented in this section serve as baselines, since they only take basic greedy decisions each time a new job arrives in the system. Depending on its priority, a heuristic decides where to execute the newly released job, without reconsidering decisions that had been taken before. The first three algorithms (AllCloud, Local and ECT) are not aware of *green* intervals and just decide where to execute the job (cloud server, local edge server, or another edge server), while LocalGreen and ECTGreen do account for *green* intervals.

AllCloud: This baseline sends all jobs to the cloud upon their release; hence, there is no execution on edge servers. The complexity is in $O(1)$ per job.

Local: This algorithm favors locality: a newly released job is scheduled on its *origin* edge server, as soon as possible after the last job that is already scheduled on it, if it is possible to finish its execution before its deadline. Otherwise, the job is immediately sent to the cloud server. The complexity is again in $O(1)$ per job.

ECT: This *Earliest Completion Time* algorithm schedules the new job on the edge server on which it will complete first, taking into account the time needed to transfer the job (for a non-local execution). If no server is able to complete the job before its deadline, the job is sent to the cloud server. The complexity is in $O(n)$ per job, since we try all possible n edge servers.

LocalGreen: This algorithm aims at a local execution similarly to Local, but it also cares about *green* intervals. Hence, the new job is scheduled on its *origin* edge server, after the last job that is already scheduled on it, but using as much *green* intervals as possible while finishing before its deadline. If not enough *green* intervals are available, it completes the execution with the latest *brown* intervals before its deadline. If the deadline cannot be met locally, the job is sent to the cloud server. For each job, we consider all edge intervals until its deadline; hence, a complexity of $O(pa)$ per job.

ECTGreen: In this algorithm, the new job is scheduled on an edge server that can execute it before its deadline (similarly to ECT), and among these, the algorithm chooses the edge server that has a *green* interval at the earliest time, taking into account transfer time. If no server can execute the job before its deadline, the job is sent to the cloud. ECTGreen allocates the job, if possible, only on *green* intervals. Otherwise, it completes the execution with the latest *brown* intervals before its deadline, similarly to LocalGreen. The complexity is $O(npa)$ per job, since we now consider all edge intervals until the job's deadline, but on all edges.

4.2 OfflineGreenest– Optimal carbon cost for ordered jobs in one edge

In this section, we present OfflineGreenest (see Algorithm 1), which runs on a single edge server, in offline mode, on an ordered list of m jobs $J_j, 1 \leq j \leq m$. It executes jobs in the given order on that edge server, maximizing the amount of *green* seconds used. Furthermore, if there is a solution for allocating all these jobs, then OfflineGreenest will find the optimal solution that uses the server as soon as possible for each job. This property becomes especially useful when the algorithm is used in an online setting because, then, when a new job arrives on the system, a good fraction of the load will already have been processed.

During its initialization phase, OfflineGreenest computes *restricted release dates* and *restricted deadlines*. The restricted release date rr_j of a job J_j, computed at Line 1, is its earliest possible starting time, defined as the minimum between its release date and the earliest possible completion time of the preceding job J_{j-1}. Similarly, the restricted deadline rd_j of job J_j, computed at Line 2, is its latest possible completion time, defined as the minimum between its deadline and the latest possible starting time of the succeeding job, J_{j+1}. To ease the writing of the algorithm, we have assumed that restricted release dates and deadlines only occur at the extremities of the green and brown intervals $[\tau_i, \tau_{i+1}]$. If this is not the case, one just needs to subdivide these intervals, increasing their number by at most $2m$.

Because the jobs must be executed in the given execution order, any schedule is completely defined by stipulating when the server is active and processes jobs. The OfflineGreenest algorithm defines the amount of time each *green* and *brown* interval will be used to process jobs (during the remainder of these intervals, the edge server will be idle).

OfflineGreenest works in two passes. In the first pass (Lines 4 through 13), from time 0 through time T, it computes which amount AmountUsed$_i$ of each *green* interval $[\tau_i, \tau_{i+1}[$ will be used in the final schedule. This computation is done through the use of two variables: Work, which is the total size of the *active* jobs, that is of jobs which have been released but whose

Algorithm 1: OFFLINEGREENEST

```
1  rr_1 ← r_1;  for j = 2 to m do rr_j ← max{r_j, rr_{j-1} + ℓ_{j-1}}
2  rd_m ← d_m;  for j = m − 1 downto 1 do rd_j ← min{d_j, rd_{j+1} − ℓ_{j+1}}
3  θ_1, ..., θ_N ← SORT({τ_i | 1 ≤ i ≤ u} ∪ {rr_j | 1 ≤ j ≤ m} ∪ {rd_j | 1 ≤ j ≤ m})
4  RESERVEDGREEN^(1) ← 0
5  for x = 1 to N                    /* Compute amount of green intervals to use */
6  do
7      if θ_x = rd_j then
8          WORK ← WORK − ℓ_j
9          RESERVEDGREEN^(1) ← max{0, RESERVEDGREEN^(1) − ℓ_j}
10     if θ_x = rr_j then WORK ← WORK + ℓ_j
11     if θ_x = τ_i and [τ_i, τ_{i+1}] is green then
12         AMOUNTUSED_i ← min{τ_{i+1} − τ_i, WORK − RESERVEDGREEN^(1)}
13         RESERVEDGREEN^(1) ← RESERVEDGREEN^(1) + AMOUNTUSED_i
14 RESERVEDGREEN^(2) ← 0
15 for x = N downto 1                /* Compute amount of brown intervals to use */
16 do
17     if θ_x = τ_i and [τ_i, τ_{i+1}] is green then
18         RESERVEDGREEN^(2) ← RESERVEDGREEN^(2) + AMOUNTUSED_i
19     if θ_x = rr_j then
20         if ℓ_j ≤ RESERVEDGREEN^(2) then
21             RESERVEDGREEN^(2) ← RESERVEDGREEN^(2) − ℓ_j
22         else
23             Allocate greedily an amount of brown intervals of size
                ℓ_j − RESERVEDGREEN^(2) during [rr_j; min{rd_j, rr_{j+1}}]
24             RESERVEDGREEN^(2) ← 0
```

deadline has not yet be reached; and RESERVEDGREEN$^{(1)}$, which records the amount of *green* intervals that has been allocated to the execution of active jobs. Each time we encounter a restricted deadline (Line 7), we update these two variables: WORK is decreased by the duration of the job (Line 8), and RESERVEDGREEN$^{(1)}$ by the amount that was pre-allocated to that job (Line 9). Each time we encounter a restricted release date, we increase WORK by the duration of the newly released job (Line 10). Each time we encounter a *green* interval, we allocate as much of it as possible to the processing of jobs (Line 12) and we update RESERVEDGREEN$^{(1)}$ accordingly (Line 13).

At the end of this first pass, we have reserved as much of the *green* intervals as was possible. This was done using the earliest possible *green* intervals. Note, however, that we have not made any allocation; deciding which job is using which *green* interval is done during the second pass. In this second pass (Lines 14 through 24), we consider intervals in reverse order, from time T through time 0, and identify which amount of each *brown* interval will be used in the final schedule. This computation is done using a single additional variable, RESERVEDGREEN$^{(2)}$, which records the amount of already encountered *green* intervals that has been reserved for job execution in the first pass, and that has not already been allocated to jobs. Each time we encounter a *green*

interval, we increase ReservedGreen$^{(2)}$ by the amount of that interval that was reserved in the first pass (Line 18). Each time we encounter a restricted release date (Line 19), we have two cases to consider. If the amount of *green* intervals reserved is no smaller than the length of the job, all this job will be executed using *green* intervals and we decrease ReservedGreen$^{(2)}$ accordingly (Line 21). Otherwise, part of the job must be executed using *brown* intervals. To do this, we just take a slice of size $\ell_j -$ ReservedGreen$^{(2)}$ of the first *brown* intervals starting from time rr_j (Line 23), and we then set ReservedGreen$^{(2)}$ to 0, as we have exhausted all the reserved *green* (Line 24).

Theorem 2. *Given a set of m ordered jobs $J_j, 1 \leq j \leq m$, if a valid schedule of these jobs exist,* OfflineGreenest *successfully executes them while optimally minimizing the carbon-cost. Furthermore, among all such executions, it completes each job at the earliest possible time.*

Proof. We have to prove that OfflineGreenest builds a valid schedule (if one exists), that it maximizes the amount of *green* seconds used (which is equivalent to the carbon-cost optimality), and that each job is completed at the earliest possible time.

First, we note that in any valid schedule, a job must be executed in the interval defined by its restricted release date and its restricted deadline. Furthermore, there exists a valid schedule if and only if for each job j, $rd_j - rr_j \geq \ell_j$.

By induction on the dates considered by its first pass, we establish that OfflineGreenest maximizes the amount of *green* seconds used. This is initially true. Let us consider a new date θ_x, which is the start of a *green* interval $[\tau_i, \tau_{i+1}]$. If AmountUsed$_i = \tau_{i+1} - \tau_i$, the result is obvious. Otherwise, this means that the total size of the active jobs (up to time τ_{i+1}) is smaller than the amount of *green* intervals already reserved plus the size of interval $\tau_{i+1} - \tau_i$. Therefore, all active jobs will be executed using solely *green* intervals and the amount of used *green* intervals is also maximized.

To prove that OfflineGreenest builds a valid schedule, we must prove that the second pass is always successful. The only potential problem is with the allocation of (parts of) *brown* intervals. However, job J_j is only allocated *brown* energy during the interval $[rr_j; \min\{rd_j, rr_{j+1}\}]$ and these intervals do not intersect. Furthermore, we have $rr_{j+1} \geq rr_j + \ell_j$ by definition, and the existence of a valid schedule implies $rd_j \geq rr_j + \ell_j$. Therefore, the interval $[rr_j; \min\{rd_j, rr_{j+1}\}]$ is at least of length ℓ_j. Furthermore, the amount of *green* intervals in $[rr_j; \min\{rd_j, rr_{j+1}\}]$ is at most ReservedGreen$^{(2)}$. Indeed, ReservedGreen$^{(2)}$ is the amount of *green* intervals reserved in the time interval $[rr_j; T]$ and not allocated to jobs $J_{j+1}, ...J_m$. Furthermore, all the jobs in $J_1, ...J_j$ must have completed by the time rd_j and, thus, ReservedGreen$^{(2)}$ is the amount of *green* intervals reserved in the time interval $[rr_j; rd_j] \supset [rr_j; \min\{rd_j, rr_{j+1}\}]$ and not allocated to jobs $J_{j+1}, ...J_m$. Hence, there is at least an amount $\ell_j -$ ReservedGreen$^{(2)}$ of *brown* intervals during that interval.

Finally, the greedy allocation of Line 23 directly gives that each job is completed at the earliest possible date (because *green* intervals are also used as early as possible). □

The overall complexity of OfflineGreenest is $O(u + m)$, where u is the total number of *green* and *brown* intervals on the edge server.

4.3 Algorithms building on OfflineGreenest

In this section, we present sophisticated algorithms that rely on three mapping strategies and two job priorities to schedule jobs; once an ordered list of jobs is specified for each server, the algorithms use OfflineGreenest to obtain an optimal schedule that, in addition, uses the server as soon as possible for each job. We first detail the three mapping strategies:

InPlace: We assign, if possible, jobs on their *origin* server. If there is no feasible schedule, we resort to the LowCarb strategy for the current job.

LowCarb: We assign jobs on servers so that total carbon cost is minimized. If there is no feasible schedule, we delegate jobs to the cloud.

NoCarbComm: This is a strategy similar to LowCarb, except that the carbon cost of transfers is ignored when designing the schedule, with the hope of favoring early starts on remote servers. Of course, the actual total carbon cost, including transfers, is computed in the end. If there is no feasible schedule, we delegate jobs to the cloud.

A first approach is to apply these strategies without re-evaluating previous decisions: upon a job release at time t, we do not update the schedule of planned jobs on each edge, but instead we insert the incoming job into the schedule of the edge server chosen by the strategy. To insert job J_j into the schedule of server e_i, we use OfflineGreenest on each period $[t, d_j[$, where t is the current time, that can accommodate J_j. Recall that some planned jobs may be frozen and restarted, so we are looking for all periods of length at least ℓ_j during which edge e_i is completely idle. We use OfflineGreenest to compute the schedule for J_j in a given period and keep the schedule with lowest carbon cost; using OfflineGreenest for a single job is not an overkill, because at most two passes are needed to find the earliest optimal schedule for J_j.

This first approach leads to greedy heuristics GreedyInPlace, GreedyLowCarb and GreedyNoCarbComm. The InPlace rule ensures that we favor a local execution, while LowCarb aims at minimizing the carbon cost (which may induce more communications). Finally, by pretending to ignore transfer costs, NoCarbComm aims at starting a job as soon as possible. On each edge, the complexity is in $O(pa)$; hence, a total complexity for these greedy heuristics in $O(npa)$, to tentatively schedule the job on each edge server.

Much more ambitiously, a second approach re-evaluates previous scheduling decisions upon release of each new job. There is some flexibility because *planned* jobs may be completely re-scheduled, e.g., on other edge servers, and *started* jobs may have their execution frozen and restarted differently. We consider two priority functions when re-evaluating scheduling decisions:

Looseness: Jobs are prioritized according to the time remaining before their deadline, weighted by their size, in non-decreasing order; at time t, the looseness of job J_j is $\frac{d_j - t}{\ell_j}$;

EDF: Jobs are prioritized according to the time remaining before their deadline, in non-decreasing order (*Earliest Deadline First*).

We combine the three mapping strategies with the two priority rules, hence, obtaining six algorithms performing reallocation, denoted

ReallocInPlaceLooseness,
ReallocInPlaceEDF,
ReallocLowCarbLooseness,
ReallocLowCarbEDF,
ReallocNoCarbCommLooseness,
ReallocNoCarbCommEDF.

At each new job release, these six algorithms reconsider all previous decisions for *planned* jobs. They all sort *planned* jobs and the new job according to the priority function, and schedule them one by one, in this order. To this purpose, they follow the target strategy. Initially, we keep in the local list of each edge server only the job that has already started its execution, if such a job exists, since it will always remain the first job on the ordered list of the server. As the schedule is being rebuilt, new jobs are assigned to the list of each server. To assign a new job on a server, we call OfflineGreenest for all the jobs currently in the list, including the first one (for the remainder of its execution). Altogether, this second approach is much more costly,

b_{trans}	$\{10, 100, 500, 1000\}$ Mbit/s
k_{trans}	$\{1, 10, 100, 1000\}$ units of carbon/Mbit
k	180 units of carbon/second
Edge servers	10
Solar only	all powered by solar generation with a grid connection
Wind only	all powered by wind generation with a grid connection
Solar and Wind	all powered by solar and wind generation with a grid connection
Mix	30% powered by solar generation, 30% by wind generation, 30% by solar and wind, and all with a grid connection
Job duration	Right skewed in [0.4,340] minutes with mean 60 minutes
Job data volume	Uniformly distributed in [2, 200] Gbit
Load	{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Looseness	{2, 4, 6} ±10%
Uniform workload	workload uniformly distributed across all edge servers
Clustered workload	30% of the edge servers receive 90% of the workload
Mall workload	10% of the edge servers receive 80% of the workload
T	30 days
CLOUD speed	10 times edge speed
CLOUD carbon cost	10 times edge carbon cost
CLOUD bandwidth	250 Mbit/s
CLOUD transfer cost	1000 units of carbon/Mbit

Table 1: Values of the different parameters for the experiments.

because it computes a new schedule, job after job, potentially considering each edge server, and each time applying OFFLINEGREENEST. However, we expect dramatic cost savings!

We conclude with a technical exception to the general rule above: for the REALLOCINPLACE variants, if a job cannot be executed locally, we do not assign it to another edge server until all the remaining planned jobs have been assigned. Once all jobs that could be assigned locally have been scheduled, we perform a second pass on the jobs that need to be sent to another edge server, following the LOWCARB strategy.

The complexity of running these algorithms is in $O(pa\overline{m}(\overline{m}+m))$ at each release date, where $\overline{m}$ is the maximum number of overlapping jobs at any time ($\overline{m} = \max_{0 \leq t \leq T} \left\{ |\{J_j \mid r_j \leq t < d_j\}| \right\}$).

5 Experiments

In Section 5.1, we describe the simulation settings. All parameters are summarized in Table 1. Simulation results are discussed in Section 5.2.

5.1 Methodology

5.1.1 Modeling edge resources

We model an edge location with multiple edge servers, and each edge server can have different on-site renewable generation sources. We consider four edge server models: (1) with solar generation, (2) with wind generation, (3) with both solar and wind generation, and (4) with no on-site renewable generation (with only an electric grid connection). When renewable generation is not available, the edge server consumes power from the electric grid. Each edge server is connected to the cloud to ensure that job deadlines can be met even with restricted renewable availability.

The carbon intensity of computation at the edge is determined by the carbon intensity of the edge server's power source. Renewable generation produces *green* intervals (carbon intensity = 0). However, renewable generation is only intermittently available, requiring the electric grid for power supply when unavailable. The electric grid is a mix of both renewable (solar, wind) and non-renewable (gas, coal, nuclear) generation sources. Therefore, consuming power from the grid produces *brown* intervals. Figure 2 illustrates *green* and *brown* intervals across the edge server models over a one-week period. Solar only, wind only, and solar and wind have *green* intervals whenever renewable generation is available and switch to *brown* generation when unavailable. The grid-only edge model consumes power directly from the grid, producing a *brown* period. The cloud consumes power from the same grid than the edge servers.

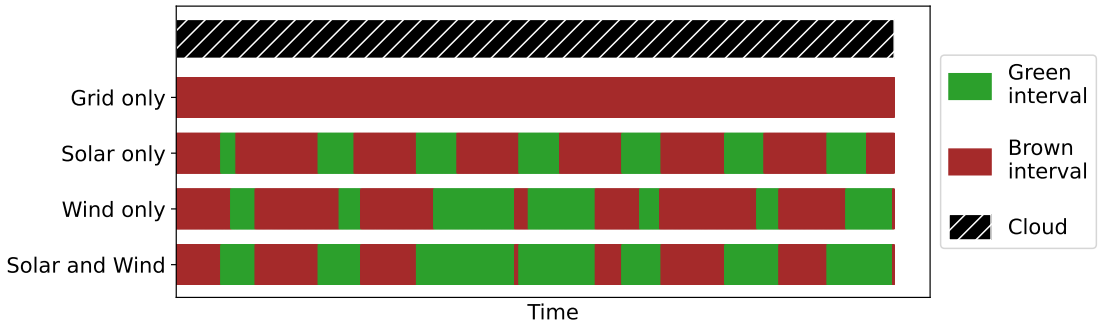

Figure 2: Green and brown intervals across edge server models over a one-week period.

We assume an edge location in California, USA, and model the electric grid's carbon intensity based on the CAISO grid [7]. We use data from [39] from August 2023 to August 2024 at 5-minute granularity. The carbon intensity of *brown* intervals is modeled as a constant $k = 180$ units of carbon/second, based on CAISO's average carbon intensity over the considered period.

To model on-site solar generation, we use CAISO's grid-wide solar generation. Since the CAISO grid is vertically elongated, its solar generation is representative of an edge server's on-site generation. We account for the grid's capacity factor of 25% [6, 37], which reflects actual generation output, significantly lower than the theoretical maximum capacity. An edge server is considered to have solar generation available at time t if the CAISO grid's solar generation at that time exceeds its 75th percentile value (i.e., 1 - solar capacity factor). Thus, we model on-site solar generation availability at an edge server at time t as: $solarGen(t, CAISO) > P75(solarGen(CAISO))$.

We build a similar model for wind generation at the edge. We consider the LZ_WEST zone in the ERCOT grid (Texas, USA) due to its substantial wind penetration [34]. We obtain zone-level wind generation from [12] and similarly account for its capacity factor of 40% [11]. Wind generation is considered available at an edge server at time t when: $windGen(t, LZ\_WEST) >$

$P60(windGen(LZ\_WEST))$. In the wind and solar generation edge server model, a green interval occurs if either of the above conditions is met.

We consider four configurations of edge locations, each with a total of 10 edge servers:

- *Solar only*: 10 edge servers powered by on-site solar generation, with a grid connection.
- *Wind only*: 10 edge servers powered by on-site wind generation, with a grid connection.
- *Solar and Wind*: 10 edge servers powered by both on-site solar and wind generation, with a grid connection.
- *Mix*: 3 edge servers powered by on-site solar generation, 3 edge servers powered by on-site wind generation, and 3 edge servers powered by both solar and wind generation, and one edge server with no renewable generation, with a grid connection.

5.1.2 Modeling edge workloads

We generate synthetic edge workloads based on the properties of realistic edge workloads. Each job is assumed to consume all cores on an edge server, with job durations that are right skewed in $[0.4, 340]$ minutes, with mean 60 minutes.

The *Load* of the system is the ratio of the total length of jobs ($\sum_{j=1}^{m} \ell_j$) to the number of edges by the execution horizon (nT). We consider a range of system load factors, varying from 0.1 to 1 in 0.1 increments. Each hour, *Load* $\times n$ core-hours are released, (assuming 1 core per server), with job release times uniformly distributed throughout the hour.

The *Looseness* of a job J_j is the ratio of the length of its execution window ($d_j - r_j$) relative to its length ℓ_j. This essentially models the workload's temporal flexibility, and we consider looseness values of 2, 4, and 6, with a random noise of $\pm 10\%$ in each.

Job arrivals at the edge are not necessarily uniformly distributed among edge servers. In practice, some edge servers may receive more requests than others due to geographical factors, such as higher demand in densely populated areas. We consider three job arrival models, with randomly selected edges nodes that receive higher load:

- *Uniform*: the workload is distributed uniformly across all edge servers.
- *Clustered*: 30% of the edge servers receive 90% of the workload.
- *Mall*: 10% of the edge servers receive 80% of the workload.

We randomly generate 10 workloads for each triplet of parameters (job arrival model; *Load*; *Looseness*) and assign each job an *origin* server edge according to the chosen job model, a deadline according to looseness, and a data/communication volume drawn uniformly between 2 and 200 Gbit. We impose the same data/communication volume for input and for output ($f_j = f_j^{in} = f_j^{out}$). The number of jobs of a set depends on the *Load* and of the execution horizon ($T = 30$days).

5.1.3 Model parameters

We list below the values of the different parameters:

- The bandwidth between edge servers, b_{trans}, takes values in $\{10, 100, 500, 1000\}$ Mbit/s.
- The carbon cost of communication between edge servers, k_{trans}, takes values in $\{1, 10, 100, 1000\}$ units of carbon/Mbit.
- The cloud executes jobs ten times faster than the edge servers and its carbon cost per time unit is ten times larger than that of edge servers. The bandwidth between the cloud and the edge servers is set at 250 Mbit/s and the carbon cost of these communications is set at 1000 units of carbon/Mbit. Hence, the execution time of job J_j on the cloud is $t_j^{(cloud)} = \ell_j/10 + 2(f_j/250)$ and its carbon cost is $C_j^{(cloud)} = 180\ell_j + 2 \times 1000 f_j$. In addition, we impose that the time needed to send and execute a job on the cloud is never larger than

its execution time on its origin server. This is equivalent to limiting the communication volume of a job as a function of its length ($f_j \leq 125.5\ell_j$).

5.1.4 Metrics

Our performance measure is the total carbon cost of the solutions produced by the different scheduling algorithms. However, the carbon cost of the optimal solution may vary by order of magnitudes depending on the choice of the simulation parameters. Therefore, we cannot directly compare the raw performance of different schedules on different instances. To circumvent this problem, the first solution is to use an ORACLE that is supposed to know, for each instance, which algorithm is providing the best solution. Then, for each algorithm, we compute its *RatioOracle*, that is, for each instance, the ratio of its carbon cost to the carbon cost found by ORACLE. The *RatioOracle* is always greater than 1, and the smaller the better. We can then build statistics on the *RatioOracle*, like its (geometric) mean.

Using an omniscient oracle as baseline enables to determine the apparent overhead of always using the same algorithm to build solution. Another approach would be to use as baseline a simple algorithm. We then build the relative performance *RatioLocal* using as baseline LOCAL. If the average *RatioLocal* of an algorithm is 0.2, this means that this algorithm has a geometric average cost equal to 20% of that of LOCAL; in other words, using it rather than LOCAL divides the carbon cost by 5 (on average).

5.2 Results

We now report the simulation results. We start by comparing the overall performance of all algorithms before focusing on a comparison between the greedy baseline and the best algorithms. We conclude the section by assessing the reduction in carbon cost provided by the best solutions.

Combining the possible values of all the parameters and then generating several instances for each combination would have led to a prohibitively large amount of simulations to perform. Therefore, we opted for a randomized approach: we generated 20,000 instances by randomly drawing an instance for each of the parameters and models.

5.2.1 Statistics on all algorithms

Table 2 presents statistics on the *RatioOracle* performance of all algorithms, where algorithms are sorted with respect to their geometric mean.

Algorithm REALLOCLOWCARBEDF performs much better than other algorithms: it has a mean of 1.06 while other algorithms have a mean greater than 1.27. Furthermore, this algorithm has a standard deviation close to 1; hence, its performance is very stable. Moreover, in half of the instances, REALLOCLOWCARBEDF found the solution with lowest cost, when REALLOCINPLACEEDF also found it for half of instances but achieving larger mean and standard deviation. Finally, if we allow a carbon overhead of 10% with respect to the performance of the best solution, once again REALLOCLOWCARBEDF achieves the best performance: it achieves a *RatioOracle* no larger than 1.1 in 79% of instances.

Each algorithm with re-evaluation and EDF priority achieves a significantly better performance average than its LOOSENESS counterpart. Therefore, in the remainder of this section, we focus on the algorithms with re-evaluation and EDF priority and on the greedy baseline algorithms.

5.2.2 Baselines vs. re-evaluation algorithms with EDF priority

Figure 3 presents the influence of the load (on the left) and of the carbon cost of transfer (per Mbit of data/communication volume of the job, on the right) for the main algorithms for the *RatioOracle* metric, using a logarithmic scale.

First, we observe that the heavier the load (or the higher the carbon cost of transfers), the lower the cost of AllCloud. Hence, the improvement obtained by the best algorithms is less significant. The curve of AllCloud gives us an idea of the difficulty of the instances or, conversely, the room for improvement. Therefore, regardless of algorithm decisions, the relative gain will be lower when the system is overloaded and/or with a high carbon cost of transfers.

Most conclusions drawn from the comparison of the performance of the greedy algorithms

Algorithms	*Mean*	*SD*	*Best*	10%
AllCloud	37.737	2.932	0	0
LocalGreen	11.821	3.073	0	0
Local	3.915	2.879	2	7
ECTGreen	2.228	2.219	4	21
ECT	2.050	1.852	1	8
GreedyNoCarbComm	1.911	2.169	2	31
ReallocNoCarbCommLooseness	1.894	2.173	2	31
ReallocNoCarbCommEDF	1.787	2.190	13	45
GreedyInPlace	1.587	1.736	6	28
ReallocInPlaceLooseness	1.558	1.707	6	27
GreedyLowCarb	1.367	1.484	12	44
ReallocInPlaceEDF	1.272	1.690	49	68
ReallocLowCarbLooseness	1.272	1.335	11	47
ReallocLowCarbEDF	1.059	1.115	48	79

Table 2: Statistics on 20,000 random instances. The geometric means and standard deviations of the *RatioOracle* of the different algorithms are reported in the *Mean* and *SD* columns. The *Best* column contains the percentage of instances for which the algorithm found a solution with lowest cost. Column 10% presents the percentage of instances for which the algorithm found a solution whose cost was within 10% of the best solution.

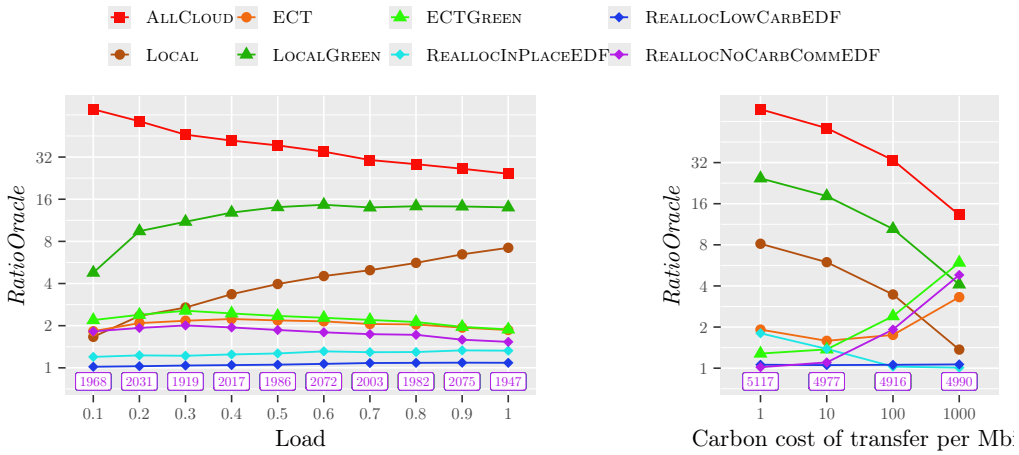

Figure 3: Impact of the load (on the left) and of the carbon cost of transfer (on the right) on the *RatioOracle* performance of algorithms (with a logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

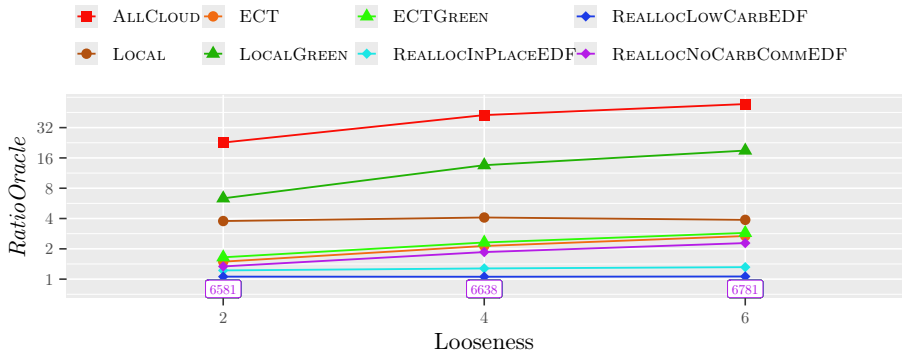

Figure 4: Impact of the looseness on the *RatioOracle* performance of algorithms (with a logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

were expected. ECT generally finds better results than Local (respectively ECTGreen than LocalGreen): because Local and LocalGreen do not transfer jobs, they end up using the cloud more often to meet deadlines, which penalizes them (on average, respectively 13% and 37% of jobs are executed on the cloud by Local and LocalGreen, compared to 1.8% for ECT and 2.8% for ECTGreen, see Table 3 in Appendix A). This trend is reversed when the carbon cost of transfers becomes too large. Because ECT and ECTGreen have no control over transfers, they allow transfers to happen even if this is more expensive than running jobs on the Cloud. There is one potentially surprising phenomenon: LocalGreen, which is aware of *green* intervals, obtain significantly worse results than Local, which is unaware of them. This can be explained: LocalGreen sends more jobs to the cloud, because rather than executing jobs immediately it waits for *green* intervals and, thus, wastes time, leaving less room for subsequent jobs which then have to run on the Cloud. Its lower usage of *brown* intervals does not compensate for its overutilization of the Cloud. ECTGreen does not achieve better performance than ECT because it performs more transfers whose cost is not balanced by their benefit: this is exemplified when the carbon cost of transfer is at least 100.

ReallocNoCarbCommEDF achieves better overall performance than ECT, performing slightly more job transfers but using less the Cloud. ReallocNoCarbCommEDF becomes competitive when the load is very high and very competitive when the carbon cost of transfer is negligible. On the contrary, it is useless when this transfer cost becomes very high, which explains its poor ranking in Table 2. The performance of ReallocInPlaceEDF is good in general but rather poor when the carbon cost of transfer is small because it does not use this opportunity to transfer jobs. Finally, the performance of ReallocLowCarbEDF is excellent and consistent, as expected from Table 2. It is the best algorithm, except when the carbon cost of transfer becomes negligible, a case in which it is very slightly underperforming.

The parameters other than the load and the carbon cost of transfer have little impact on the relative performance of the different algorithms, see Appendix A. However, when the job *Looseness* increases (see Figure 4), the performance of ReallocLowCarbEDF and ReallocInPlaceEDF improves with respect to the other algorithms: as the looseness increases, there are more opportunities of optimization which these algorithms succeed to benefit from. Also, we can see on Figure 8 of Appendix A that the performance of ReallocInPlaceEDF is identical to that of ReallocLowCarbEDF, except when the configuration of edge locations is *Mix*. Hence, the average performance of these two algorithms only differ because of the inclusion of the *Mix* configuration in our simulation settings.

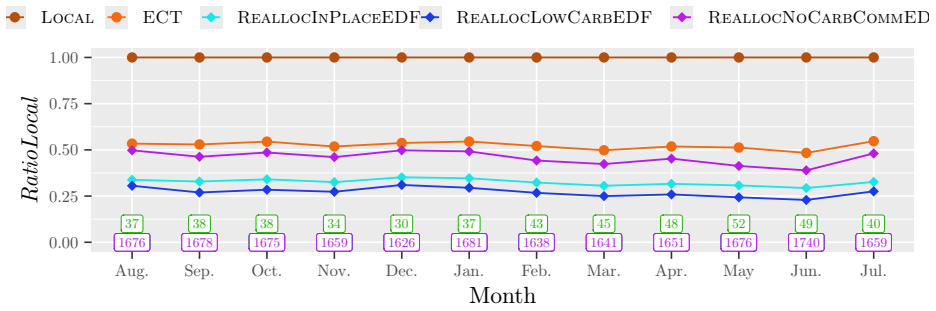

Figure 5: Impact of the month on the *RatioLocal* metric. Numbers in green indicate the average percentage of *green* for the month.

5.2.3 Comparison of algorithms with re-evaluation to Local

Here, we compare the performance of ECT and of the three re-evaluation algorithms with EDF priority using the *RatioLocal* metric. Figure 5 presents the influence of months on the performance of algorithms. We obtained this figure by grouping the results obtained from all simulations, especially those using different edge models. Hence, the percentage of green for a given month corresponds to the average percentage of green in the *Solar only*, *Wind only*, and *Solar and Wind* models. We can note that, in May, edge servers are powered 52% of the time by renewable energies, compared with only 30% in December (looking only at the *Solar only* model, the amount of renewable energy is 4% in December but 43% in May). Hence, there is a significant variation in the proportion of renewable energy depending on the month. Therefore, one could have expected that the month of the year would have had an impact on the relative performance of the algorithms. It is immediately apparent that this is not at all the case.

Overall, the carbon cost of ECT is only 52% of the cost of Local. This cost falls to 45% for ReallocNoCarbCommEDF, to 32% for ReallocInPlaceEDF and to 27% ReallocLowCarbEDF. When comparing their carbon cost to that of ECT, ReallocNoCarbCommEDF still gets a saving of 13%, ReallocInPlaceEDF, a saving of 38%, and ReallocLowCarbEDF, a saving of 48%.

6 Conclusion

We have studied classical greedy, simple carbon-aware, and carbon-aware with transfer scheduling approaches applied to the edge green scheduling problem. The carbon-aware algorithms consider additional dimensions of carbon variation as well as the cost of job transfer (communication carbon costs). The results show that carbon-aware schedulers can robustly reduce carbon emissions for workload, across a variety of load and communication carbon costs. The best two of the more sophisticated algorithms that consider reallocation consistently outperform these simple carbon-aware schedulers. These results are also robust to seasonal variations in edge site carbon content of power, and workload flexibility (looseness).

Future work will extend OfflineGreenest to a more general case with c different costs for the intervals, with $2(c-1)$ passes likely needed to obtain an optimal solution. We will also investigate scenarios where brown and green intervals are not completely known in advance, which would require the scheduling algorithms to dynamically re-act and reallocate jobs to cope with sudden (unexpected) variations.

References

[1] B. Aksanli, J. Venkatesh, L. Zhang, and T. 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.

[2] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. *SIAM J. on Computing*, 31(5):1370–1382, 2002.

[3] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Average stretch without migration. *J. Comput. Syst. Sci.*, 68:80–95, 02 2004.

[4] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[5] A. Brogi, S. Forti, C. Guerrero, and I. Lera. How to place your apps in the fog: State of the art and open challenges. *Software: Practice and Experience*, 50(5):719–740, 2020.

[6] California Energy Commission. Building energy efficiency standards for residential and nonresidential buildings. https://www.energy.ca.gov/sites/default/files/2021-11/CEC-200-2021-010.pdf, 2021.

[7] California ISO team. California Independent System Operator (CAISO). https://www.caiso.com, 2025. Accessed: 2025-03-12.

[8] X. Chen. Job scheduling on edge and cloud servers. In *2022 IEEE 8th Intl Conference on Big Data Security on Cloud (BigDataSecurity)*, pages 87–91, 2022.

[9] A. A. Chien, L. Lin, H. Nguyen, V. Rao, T. Sharma, and R. Wijayawardana. Reducing the carbon impact of generative ai inference (today and in 2035). In *Proceedings of the 2nd workshop on sustainable computer systems*, pages 1–7, 2023.

[10] J. Dongarra, T. Herault, and Y. Robert. *Fault tolerance techniques for high-performance computing*. Springer, 2015.

[11] Electric Reliability Council of Texas (ERCOT). Ercot zonal reliability study report, January 2023. Accessed: March 12, 2025.

[12] Electric Reliability Council of Texas (ERCOT). Ercot data products - historical carbon intensity data, 2024. Accessed: March 12, 2025.

[13] European Union. The AI@EDGE H2020 Project. https://aiatedge.eu/, 2020.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[15] Í. Goiri, W. Katsak, K. Le, T. D. Nguyen, and R. Bianchini. Parasol and greenswitch: Managing datacenters powered by renewable energy. *ACM SIGPLAN Notices*, 48(4):51–64, 2013.

[16] Í. Goiri, K. Le, M. E. Haque, R. Beauchea, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. Greenslot: scheduling energy consumption in green datacenters. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[17] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. Greenhadoop: leveraging green energy in data-processing frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 57–70, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Google. Google 2024 Environmental Report. https://sustainability.google/reports/google-2024-environmental-report/, 2024.

[19] S. Hall, F. Micheli, G. Belgioioso, A. Radovanović, and F. Dörfler. Carbon-aware computing for data centers with probabilistic performance guarantees. *arXiv preprint arXiv:2410.21510*, 2024.

[20] W. A. Hanafy, Q. Liang, N. Bashir, D. Irwin, and P. Shenoy. Carbonscaler: Leveraging cloud workload elasticity for optimizing carbon-efficiency. In *Abstracts of the 2024 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS/PERFORMANCE '24, page 49–50, New York, NY, USA, 2024. Association for Computing Machinery.

[21] W. A. Hanafy, Q. Liang, N. Bashir, A. Souza, D. Irwin, and P. Shenoy. Going green for less green: Optimizing the cost of reducing cloud carbon emissions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 479–496, 2024.

[22] M. E. Haque, I. Goiri, R. Bianchini, and T. D. Nguyen. Greenpar: Scheduling parallel high performance applications in green datacenters. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 217–227, 2015.

[23] T. Herault, Y. Robert, A. Bouteiller, A. Arnold, K. B. Ferreira, G. George, and J. Dongarra. Checkpointing strategies for shared high-performance computing platforms. *International Journal of Networking and Computing*, 9(1):28–52, 2019.

[24] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[25] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin, and J. Wan. Energy aware edge computing: A survey. *Computer Communications*, 151, 2020.

[26] A. Lechowicz, N. Christianson, B. Sun, N. Bashir, M. Hajiesmaili, A. Wierman, and P. Shenoy. Carbonclipper: Optimal algorithms for carbon-aware spatiotemporal workload management. *arXiv preprint arXiv:2408.07831*, 2024.

[27] A. Lechowicz, N. Christianson, J. Zuo, N. Bashir, M. Hajiesmaili, A. Wierman, and P. Shenoy. The online pause and resume problem: Optimal algorithms and an application to carbon-aware load shifting. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(3), Dec. 2023.

[28] K. Li. Scheduling independent tasks on multiple cloud-assisted edge servers with energy constraint. *Journal of Parallel and Distributed Computing*, 184, 2024.

[29] Y. Li, A.-C. Orgerie, and J.-M. 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.

[30] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser. Renewable and cooling aware workload management for sustainable data centers. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 175–186, 2012.

[31] Microsoft. Microsoft 2024 Environmental Sustainability Report. https://blogs.microsoft.com/on-the-issues/2024/05/15/microsoft-environmental-sustainability-report-2024/, 2024.

[32] J. Murillo, W. A. Hanafy, D. Irwin, R. Sitaraman, and P. Shenoy. CDN-Shifter: Leveraging Spatial Workload Shifting to Decarbonize Content Delivery Networks. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 505–521, New York, NY, USA, 2024. Association for Computing Machinery.

[33] NVIDIA. NVIDIA and SoftBank Corp. Accelerate Japan's Journey to Global AI Powerhouse. https://investor.nvidia.com/news/press-release-details/2024/NVIDIA-and-SoftBank-Corp.-Accelerate-Japans-Journey-to-Global-AI-Powerhouse/default.aspx, 2024.

[34] Potomac Economics. 2023 State of the Market Report for the ERCOT Electricity Markets. https://www.potomaceconomics.com/wp-content/uploads/2024/05/2023-State-of-the-Market-Report_Final.pdf, May 2024. Accessed: March 12, 2025.

[35] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, et al. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2022.

[36] G. Schmidt. Scheduling with limited machine availability. *European Journal of Operational Research*, 121(1):1–15, 2000.

[37] S&P Global Market Intelligence. 2022 monthly us solar capacity factors underscore winter doldrums, 2022. Accessed: March 12, 2025.

[38] T. Sukprasert, A. Souza, N. Bashir, D. Irwin, and P. Shenoy. On the limitations of carbon-aware temporal and spatial workload shifting in the cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 924–941, 2024.

[39] University of Chicago. Right Place, Right Time (RiPiT) Carbon Emissions Service. http://ripit.uchicago.edu/Part1/ripit-part1.html, n.d. Accessed: 2025-03-12.

[40] P. Wiesner, I. Behnke, D. Scheinert, K. Gontarska, and L. Thamsen. Let's wait awhile: How temporal workload shifting can reduce carbon emissions in the cloud. In *Proceedings of the 22nd International Middleware Conference*, pages 260–272, 2021.

[41] C.-M. Wu, R.-S. Chang, and H.-Y. Chan. A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters. *Future Generation Computer Systems*, 37:141–147, 2014.

[42] L. Wu, W. A. Hanafy, A. Souza, K. Nguyen, J. Harkes, D. Irwin, M. Satyanarayanan, and P. Shenoy. Carbonedge: Leveraging mesoscale spatial carbon-intensity variations for low carbon edge computing. *arXiv:2502.14076*, 2025.

[43] M. Xu and R. Buyya. Managing renewable energy and carbon footprint in multi-cloud computing environments. *Journal of Parallel and Distributed Computing*, 135:191–202, 2020.

[44] J. Zheng, A. A. Chien, and S. Suh. Mitigating curtailment and carbon emissions through load migration between data centers. *Joule*, 4(10):2208–2222, 2020.

A Additional figures

Algorithms	*NbLocal*	*CostLocal*	*NbExtern*	*CostTransfer*	*CostExtern*	*NbCloud*	*CostCloud*
AllCloud	0.0	0.0	0.0	0.0	0.0	100.0	100.0
Local	86.9	25.1	0.0	0.0	0.0	13.1	74.9
ECT	71.0	37.5	27.2	36.1	11.5	1.8	14.9
LocalGreen	63.0	5.0	0.0	0.0	0.0	37.0	95.0
ECTGreen	46.7	22.2	50.5	47.8	12.7	2.8	17.4
ReallocInPlaceEDF	90.3	58.9	6.8	14.8	1.6	2.9	24.7
ReallocLowCarbEDF	79.4	50.8	17.9	21.6	4.3	2.7	23.3
ReallocNoCarbCommEDF	54.7	27.7	44.0	48.2	11.7	1.3	12.4

Table 3: Statistics on 20,000 random instances. The percentage of jobs executed locally, on another edge server, on Cloud are reported respectively in the *NbLocal*, *NbExtern* and *NbCloud* columns. The associated costs (in percent) are reported respectively in columns *CostLocal*, *CostTransfer* (for transferred data cost) and *CostExtern* (for external execution), and *CostCloud*.

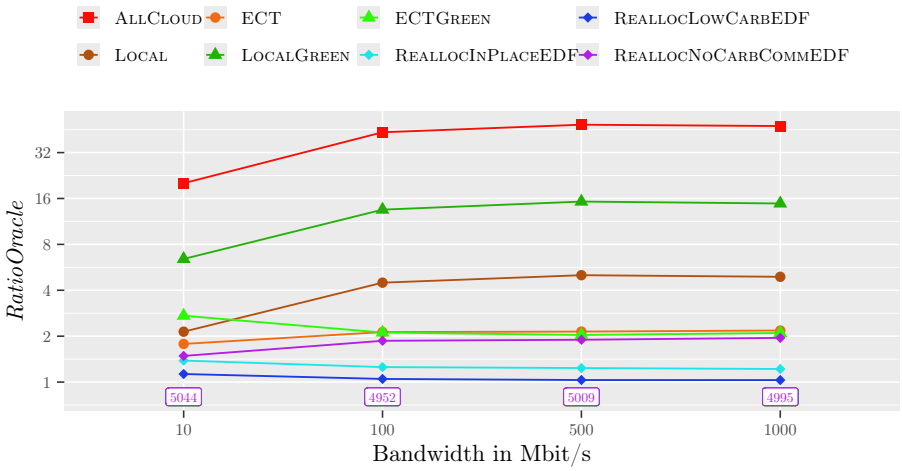

Figure 6: Impact of the bandwidth on the *RatioOracle* performance of algorithms (with a logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

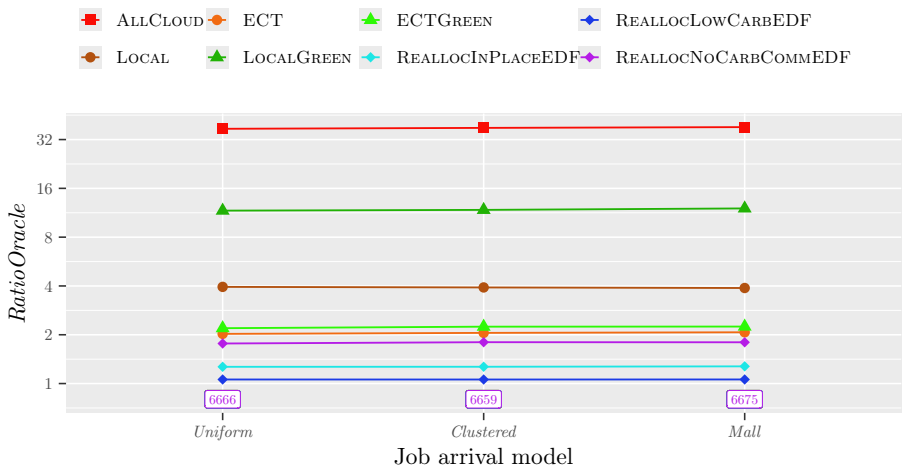

Figure 7: Impact of the job arrival model on the *RatioOracle* performance of algorithms (with a logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

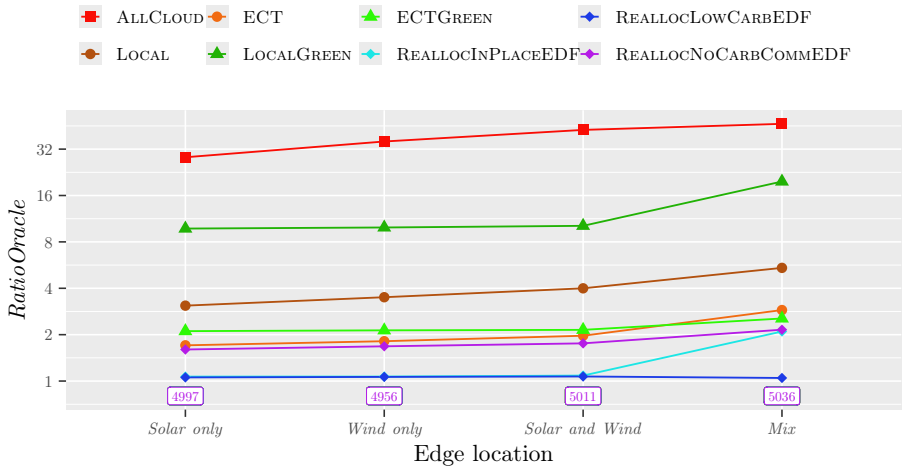

Figure 8: Impact of the edge location on the *RatioOracle* performance of algorithms (with a logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399