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 (Corresponding Author)[1][0009−0004−7636−860X],
Rajini Wijayawardana[2][0009−0006−0947−6277],
Anne Benoit[1,3][0000−0003−2910−3540], Yves Robert[1][0000−0003−2361−055X],
Frédéric Vivien[1][0000−0002−0663−6152], Andrew A. Chien[2][0000−0002−1204−206X]

[1] CNRS, Inria, ENS Lyon, UCBL, LIP UMR 5668, France
[2] University of Chicago, IL, USA
[3] Institut Universitaire de France and IDEaS, Georgia Tech, USA
{joachim.cendrier, anne.benoit, yves.robert,
frederic.vivien}@ens-lyon.fr,
{rajini, acchien}@uchicago.edu}

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. 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. This paper introduces several new algorithms that use sophisticated scheduling policies to efficiently decrease carbon cost; these algorithms maximize the use of green energy 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 42% over standard approaches.

1 Introduction and Related Work

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 [4]; 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 [6,10,5]. 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 [3]. 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. We provide in this section a high-level overview of the problem 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 Fig. 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[4]. Each job is submitted online to a local edge server, called its *origin*, and has several parameters: length, release time, deadline, and data volume. When a job is submitted, there are three possibilities: the job can be (i) executed locally, (ii) transferred to another edge server, or (iii) delegated to the CLOUD. Each job should meet its deadline, and the schedules must follow rules that are detailed in Section 2.

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

[4] Having zero carbon cost for *green* is not a restriction: what matters is the cost difference between *green* and *brown*.

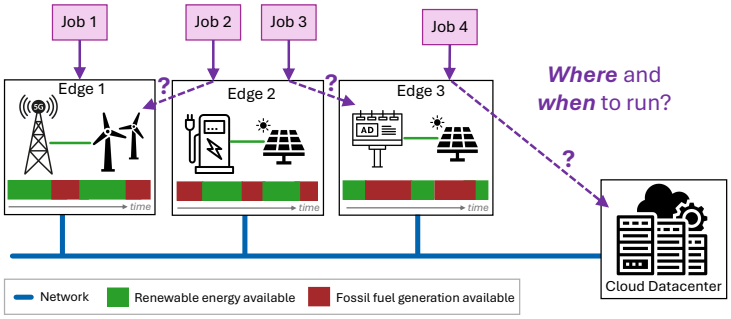

Fig. 1. The edge green scheduling problem: minimizing carbon emissions from computing, while balancing job deadlines, carbon costs of communication, and distributed resources. All in the presence of varying carbon emissions from power.

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, while still ensuring that the job deadline is met. In the latter assignment, the carbon cost of transfer must be included in the comparison.

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 3 for a complete description.

Altogether, our first contribution is a realistic yet tractable model for this important scheduling problem, and a complexity analysis (Section 2). Our second 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 design of novel scheduling algorithms with multiple servers (Section 3). We also perform a thorough evaluation 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 (Section 4). This is good news at a time where computing consumes a larger and larger fraction of energy resources! Due to lack of space, extensive discussion of related work, as well as additional results, can be found in the companion research report [2].

2 Framework

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: $I_1^{(i)} = [0, \tau_1^{(i)}[, I_2^{(i)} = [\tau_1^{(i)}, \tau_2^{(i)}[, \ldots, I_{u_i}^{(i)} = [\tau_{u_i-1}^{(i)}, \tau_{u_i}^{(i)}[$, 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. Like some related work [5], we assume full a priori knowledge of *green* and *brown* intervals, for instance through predictions based on statistical and machine learning techniques. The bandwidth between two edge servers is b_{trans}, while the carbon cost of the transfer is k_{trans} per Mbit. The platform is complemented by a powerful CLOUD server with high processing capacity, which has the capacity to process all the jobs present in the system if needed (which is unlikely).

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: release time r_j, deadline d_j, job duration on an edge ℓ_j (in seconds), data/communication input volume f_j^{in} and output volume f_j^{out} (in bits), execution time $t_j^{(cloud)}$ and carbon cost $C_j^{(cloud)}$ when delegated to the cloud. Note that $t_j^{(cloud)}$ and $C_j^{(cloud)}$ are constants that only depend on the job, not on the schedule, since the Cloud can process all jobs in parallel. Furthermore, we enforce that $t_j^{(cloud)} \leq d_j - r_j$ for all jobs, to ensure that each job can be executed on the Cloud within its deadline. A job that is executed on an edge server cannot be migrated during execution: once the execution has started on some edge server, it has to finish on that server. However, 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. Hence, the execution duration and carbon cost on an edge server (the counterparts of $t_j^{(cloud)}$ and $C_j^{(cloud)}$) depend on the intervals used by the job, the time when the job was eventually frozen, and whether the execution is local or remote. A detailed description on how to account for transfer times and carbon costs is available in [2].

There 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.

Scheduling Rules. Online scheduling problems are known to be difficult. The release of a new job, called *event*, may lead the scheduler to re-evaluate its decisions. 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 reassign 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 decisions ALAP (As Late As Possible). 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 plans the input communication ALAP so that execution starts 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. The scheduler may well re-evaluate its plan and decide to assign job J_j somewhere else, for instance to another edge or to the Cloud. Note that J_j is still *planned* at time t_3 unless it starts its execution immediately. Also, 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 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. A job J_j *planned* to execute on the Cloud is always 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 3.

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, 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. 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.

Complexity. The offline version of the edge scheduling problem resembles the classical scheduling problem of scheduling with limited machine availability [8], whose objective is to minimize makespan instead of carbon cost. We prove that this problem is difficult even with a single edge server: it is NP-complete in the strong sense, and unless P=NP, there is no constant approximation algorithm. To prove the NP-completeness, the reduction is done from 3-partition, using a set of jobs with tight deadlines that partition the execution horizon. Due to lack of space, the detailed proof can be found in the companion research report [2].

3 Algorithms

This section is devoted to the design of algorithms to solve the EdgeSched problem. Further details and the pseudo-code of the OfflineGreenest algorithm are available in the companion research report [2].

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.

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.

OfflineGreenest– Optimal carbon cost for ordered jobs in one edge. The OfflineGreenest algorithm 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 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 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}.

OfflineGreenest then works in two passes. It first goes through the intervals from time 0 to T, and reserves as much time on the *green* intervals as possible, given the restricted release dates and deadlines of jobs, without yet deciding which job to execute on which interval. The second pass is done in reverse order, from time T to 0, and it identifies the amount of *brown* intervals that will be used in the final schedule. While the first pass is enough to guarantee the optimality by using as much *green* as possible, the second pass ensures that jobs are completed as early as possible. Please refer to [2] for details. We can then prove the following theorem:

Theorem 1. *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.*

The detailed proof is available in [2]. We first show the optimality of the choices of *green* by induction on the dates considered during the first pass. We then prove that OfflineGreenest builds a valid schedule by showing that the second pass is always successful.

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

Algorithms Building on OfflineGreenest. We designed 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. The three mapping strategies are:

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, and 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, 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

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

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, 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

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 ($\overline{m} = \max_{0\leq t\leq T}\left\{|\{J_j \mid r_j \leq t < d_j\}|\right\}$).

4 Experiments

We briefly describe the simulation settings; all parameters are summarized in Table 1 and a detailed description is available in the companion research report [2]. Simulation results are then discussed.

Modeling Edge Resources. 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). 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. Fig. 2 illustrates *green* and *brown* intervals across the 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 [1]. We use data from [9] from August

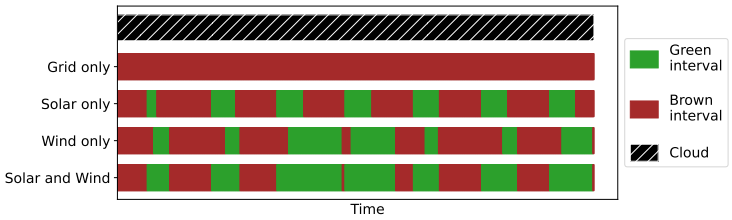

Fig. 2. Green and brown intervals across edge server models over a one-week period.

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, while we consider the LZ_WEST zone in the ERCOT grid (Texas, USA) for wind generation, due to its substantial wind penetration [7]. Model details are in [2]. Four configurations of 10-server platforms are considered, see Table 1 (*Solar only*, *Wind only*, *Solar and Wind*, and *Mix*).

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). 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. Parameter values are given in Table 1, as well as three different job arrival models (*Uniform*, *Clustered*, and *Mall*). These models account for the fact that some edge servers may receive more requests than others due to geographical factors, such as higher demand in densely populated areas.

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 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).

Metrics. The performance measure is the total carbon cost of the solutions produced by the different scheduling algorithms. In order to fairly compare different schedules on different instances, we use an Oracle that knows, 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 (*RatioOracle* ≥ 1, 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 solutions. We also consider *RatioLocal*, using Local as a baseline. If the average *RatioLocal* of an algorithm is 0.2, it 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).

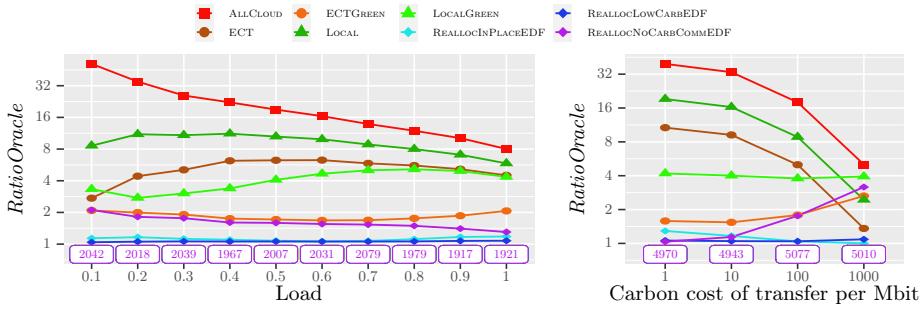

Fig. 3. Impact of the load (on the left) and of the carbon cost of transfer (on the right) on the *RatioOracle* performance of algorithms (with a logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

Results. We now report the simulation results. To avoid combining all possible values of all parameters, we generated 20,000 instances by randomly drawing an instance for each of the parameters and models.

• *Statistics on all algorithms.* Table 2 presents statistics on the *RatioOracle* performance of all algorithms. The best algorithm is ReallocLowCarbEDF, closely followed by ReallocInPlaceEDF. All the other heuristics perform at least 50% worse than ReallocLowCarbEDF in the average. In addition, ReallocLowCarbEDF has a standard deviation close to 1; hence, its performance is very stable. And for half of the instances, ReallocLowCarbEDF found the solution with lowest cost, similarly to ReallocInPlaceEDF also found but achieving a lower 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 78% 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.

• *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.

Table 2. Statistics on 20,000 random instances. Geometric means (*Mean*) and standard deviations (*SD*) of *RatioOracle*. Percentage of instances for which the algorithm achieves the lowest cost (*Best*), or with a solution whose cost is within 10% of the best solution.

Algorithms	*Mean*	*SD*	*Best*	10%
AllCloud	18.511	3.602	0	0
LocalGreen	9.040	3.166	0	0
Local	5.071	3.291	1	3
ECTGreen	3.974	2.461	0	2
GreedyNoCarbComm	2.665	2.141	0	5
ReallocNoCarbCommLooseness	2.341	1.981	0	6
GreedyLowCarb	2.014	1.818	0	10

Algorithms	*Mean*	*SD*	*Best*	10%
GreedyInPlace	1.895	1.683	0	8
ECT	1.843	1.657	0	2
ReallocLowCarbLooseness	1.620	1.496	1	14
ReallocNoCarbCommEDF	1.606	1.956	18	37
ReallocInPlaceLooseness	1.590	1.435	0	9
ReallocInPlaceEDF	1.118	1.261	49	70
ReallocLowCarbEDF	1.060	1.090	34	78

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 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 Technical Report [2]. 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 costs are not compensated 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 Technical Report [2]. 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.

• *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

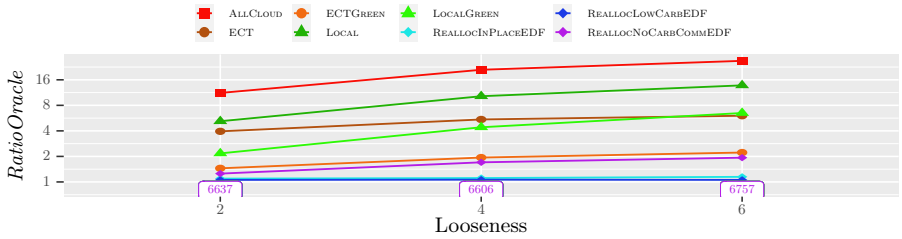

Fig. 4. Impact of looseness on the *RatioOracle* performance of algorithms (logarithmic scale). Numbers in purple indicate the number of instances corresponding to each case.

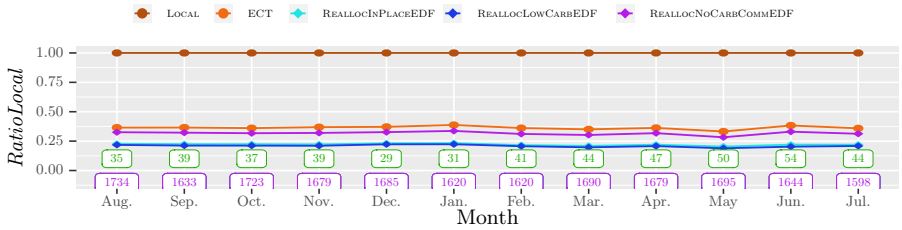

Fig. 5. Impact of the month on the *RatioLocal* metric. Numbers in green indicate the average percentage of *green* for the month.

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 June, edge servers are powered 54% of the time by renewable energies, compared with only 29% in December (looking only at the *Solar only* model, the amount of renewable energy is 5% in December but 44% in June). 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 36% of the cost of Local. This cost falls to 32% for ReallocNoCarbCommEDF, to 22% for ReallocInPlaceEDF and to 21% ReallocLowCarbEDF. When comparing their carbon cost to that of ECT, ReallocNoCarbCommEDF still gets a saving of 13%, ReallocInPlaceEDF, a saving of 39%, and ReallocLowCarbEDF, a saving of 42%.

5 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. It would also be interesting to adapt the algorithms to a communication model with contention.

Acknowledgements: We thank the anonymous reviewers for their insightful feedback. This work is supported in part by NSF Grants CNS-1901466, CNS-2325956, the VMware University Research Fund, and the UChicago-CNRS joint PhD Program.
Disclosure of Interests: The authors have no competing interests to declare that are relevant to the content of this article.

References

1. California ISO team. California Independent System Operator (CAISO). https://www.caiso.com, 2025. Accessed: 2025-03-12.
2. J. Cendrier, R. Wijayawardana, A. Benoit, Y. Robert, F. Vivien, and A. A. Chien. Green scheduling on the edge. Research report 9580, INRIA, 2025. https://inria.hal.science/hal-04994586.
3. European Union. The AI@EDGE H2020 Project. https://aiatedge.eu/, 2020.
4. Google. Google 2024 Environmental Report. https://sustainability.google/reports/google-2024-environmental-report/, 2024.
5. 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.
6. 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.
7. 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.
8. G. Schmidt. Scheduling with limited machine availability. *European Journal of Operational Research*, 121(1):1–15, 2000.
9. 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.
10. 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.