CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency

WALID A. HANAFY, University of Massachusetts Amherst, USA
QIANLIN LIANG, University of Massachusetts Amherst, USA
NOMAN BASHIR, University of Massachusetts Amherst, USA
DAVID IRWIN, University of Massachusetts Amherst, USA
PRASHANT SHENOY, University of Massachusetts Amherst, USA

Cloud platforms are increasing their emphasis on sustainability and reducing their operational carbon footprint. A common approach for reducing carbon emissions is to exploit the temporal flexibility inherent to many cloud workloads by executing them in periods with the greenest energy and suspending them at other times. Since such suspend-resume approaches can incur long delays in job completion times, we present a new approach that exploits the elasticity of batch workloads in the cloud to optimize their carbon emissions. Our approach is based on the notion of "carbon scaling," similar to cloud autoscaling, where a job dynamically varies its server allocation based on fluctuations in the carbon cost of the grid's energy. We develop a greedy algorithm for minimizing a job's carbon emissions via carbon scaling that is based on the well-known problem of marginal resource allocation. We implement a `CarbonScaler` prototype in Kubernetes using its autoscaling capabilities and an analytic tool to guide the carbon-efficient deployment of batch applications in the cloud. We then evaluate CarbonScaler using real-world machine learning training and MPI jobs on a commercial cloud platform and show that it can yield i) 51% carbon savings over carbon-agnostic execution; ii) 37% over a state-of-the-art suspend-resume policy; and iii) 8% over the best static scaling policy.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Hardware → Renewable energy**; • **Social and professional topics → Sustainability**.

Additional Key Words and Phrases: Carbon efficiency; Sustainable computing; Auto scaling

ACM Reference Format:
Walid A. Hanafy, Qianlin Liang, Noman Bashir, David Irwin, and Prashant Shenoy. 2023. CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 3, Article 57 (December 2023), 28 pages. https://doi.org/10.1145/3626788

1 INTRODUCTION

Data centers worldwide consume over 200TWh of energy each year—comprising roughly 1% of global electricity usage [46]—and are poised to grow to 3-13% of global electricity demand by 2030 [4, 37]. The growth of hyper-scale cloud providers is fueling this rapid increase in energy use, resulting in a significant environmental impact by increasing carbon and greenhouse gas (GHG) emissions [30, 47]. For the past two decades, cloud providers have relentlessly focused on improving

Authors' addresses: Walid A. Hanafy, University of Massachusetts Amherst, Amherst, MA, USA, 01002, whanafy@cs.umass.edu; Qianlin Liang, University of Massachusetts Amherst, Amherst, MA, USA, 01002, qliang@cs.umass.edu; Noman Bashir, University of Massachusetts Amherst, Amherst, MA, USA, 01002, nbashir@umass.edu; David Irwin, University of Massachusetts Amherst, Amherst, MA, USA, 01002, irwin@ecs.umass.edu; Prashant Shenoy, University of Massachusetts Amherst, Amherst, MA, USA, 01002, shenoy@cs.umass.edu.

2476-1249/2023/12-ART57 $15.00
https://doi.org/10.1145/3626788

their data centers' energy-efficiency to reduce their operational costs—by driving down their power usage effectiveness (PUE) close to the optimal value of 1. As a result, optimizations, such as server consolidation, open-air cooling, and power infrastructure improvements, have yielded significant energy-efficiency gains. However, energy-efficiency improvements alone are insufficient to satisfy cloud data centers' aggressive sustainability goals, since even energy-efficient data centers may generate significant carbon emissions from their energy use. This has led to a new emphasis on carbon-efficient operations that directly target reducing data centers' overall carbon emissions [14].

Carbon efficiency can be achieved through supply-side or demand-side methods. Supply-side methods include power purchase agreements (PPAs) from renewable generation sources, such as solar, wind, and hydro, which *indirectly* offset a cloud data center's carbon emissions. Such optimizations yield net-zero operation [28, 48, 51] over a long period, such as a year, but offsets by themselves do not eliminate the instantaneous direct emissions at all times [10]. Consequently, supply-side optimizations must be combined with demand-side methods to reduce a cloud data center's instantaneous direct carbon emissions. Demand-side optimizations exploit the fact that the carbon intensity of grid-supplied electricity varies both temporally and geographically. A common demand-side optimization is *time shifting* delay-tolerant workloads to periods with the "greenest" electricity supply. Although not all cloud workloads are delay tolerant, many types of batch workloads exhibit significant temporal, performance, and even geographic flexibility.

One approach for leveraging the temporal flexibility above is to use *suspend-resume* mechanisms [2, 19, 59, 73], where a scheduler suspends a job when electricity's carbon intensity rises (e.g., above some threshold) and resumes it when it drops (e.g., below the threshold). For example, Google recently adopted carbon-aware time-shifting in its Carbon-Intelligent Computing System [59]. While suspend-resume temporal shifting policies can reduce the carbon emissions of delay-tolerant workloads [73], they suffer from two drawbacks. First, the carbon intensity of grid-supplied electricity changes slowly, and there may be long periods (e.g., many hours) of high carbon periods where jobs remain suspended and make no progress. Such suspensions cause substantial delays in completion time, with 7-10× increases in completion times in some cases [65]. Second, when batch jobs have limited temporal flexibility and thus cannot be significantly shifted, the effectiveness of these methods is significantly reduced.

To overcome these drawbacks, we present `CarbonScaler`, a new approach that exploits the *resource elasticity* of cloud workloads to dynamically vary the amount of resources allocated to applications in response to fluctuations in the carbon cost[1] of their energy supply. Our "carbon scaling" approach is analogous to cloud autoscaling, where the number of servers allocated to a cloud application varies dynamically over time [6]. However, while cloud autoscalers generally respond to variations in applications' workload demand, often for request-based services, our "carbon scaling" approach responds to the carbon dynamics of electricity. In essence, carbon scaling scales up the servers allocated to an application when the carbon cost is low and gracefully scales them down when the cost increases. In contrast to the static allocation of suspend-resume approaches, carbon scaling enables faster progress during low carbon periods, which can potentially eliminate delays in job completion times while also reducing carbon emissions.

Designing cloud carbon scaler requires addressing two key design challenges: *how much* to scale each application up or down and *when*. Since different applications exhibit different scaling characteristics with respect to the number of allocated servers, a carbon scaler must take this scaling behavior into account when determining how much to scale up each application during low carbon periods. For example, an embarrassingly parallel job can opportunistically scale up significantly

[1]We use the terms carbon intensity and carbon cost of electricity interchangeably.

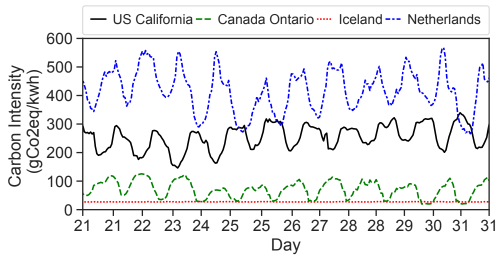


Fig. 1. *Grid's carbon intensity shown over a 10 days period varies spatially and temporally.*

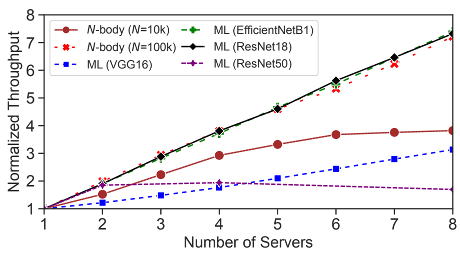


Fig. 2. *Scaling characteristics of common MPI jobs and machine learning training frameworks.*

without increasing its overhead (thus increasing its carbon efficiency), while applications with scaling bottlenecks should scale up more judiciously. To maximize carbon savings, `CarbonScaler` relies on its knowledge of the energy's future carbon intensity, application scalability profile, job length, and other execution constraints to compute a schedule to decide when to perform such scaling operations. However, carbon intensity forecasts, profile estimates, and the expected length are error-prone, requiring carbon scaling decisions to be robust to such errors.

In designing, implementing, and evaluating `CarbonScaler`, we make the following contributions.

- We introduce the notion of carbon scaling for cloud applications, which maps to the well-known problem of marginal resource allocation for which greedy optimal solutions exist [22]. `CarbonScaler` builds on these ideas to develop a greedy autoscaling algorithm that minimizes individual application's emissions by scaling the resources up and down in response to carbon cost variations. Further, CarbonScaler can substantially reduce or even eliminate the completion time delays seen in suspend-resume approaches.
- We implement a full prototype of `CarbonScaler` in Kubernetes and use it to leverage a cloud application's elasticity and temporal flexibility to reduce its carbon footprint. We also implement our algorithm in `CarbonScaler`'s CarbonAdvisor tool, which enables analysis and simulated execution of cloud applications to evaluate their carbon savings before being deployed in the cloud. CarbonAdvisor enables system designers to understand better how to minimize their carbon cost based on job characteristics, geographic region, and different run-time parameters.
- We evaluate `CarbonScaler` against multiple baselines using numerous real-world batch applications, including machine learning training and MPI-based scientific jobs. Our results show that `CarbonScaler` can yield up to i) 51% carbon savings over a carbon-agnostic execution, ii) 37% over the state-of-the-art suspend-resume policy, and iii) 8% over the best static scaling policy.

2 BACKGROUND

This section provides background on sustainable data centers and carbon-aware scheduling.

2.1 Sustainable Data Centers

In addition to their long-standing emphasis on improving energy efficiency by reducing their PUE,[2] cloud data centers have recently begun to focus on reducing their carbon footprint [2, 59]. This can be achieved by reducing operational carbon emissions measured in gCO2eq/kWh, a.k.a. Scope 2 emissions [34], resulting from electricity use, as well as by reducing embodied carbon—Scope 3 emissions—that arise during the manufacturing of data center hardware (e.g., servers). Our work focuses on reducing Scope 2 emissions. Cloud platforms have little direct (Scope 1) emissions, and optimizing embodied carbon of computing workloads is beyond the scope of this paper.

[2]Power Usage Effectiveness (PUE) is the ratio of the total energy used by a data center to the energy used for computing.

2.2 Carbon Intensity of Electricity

To reduce Scope 2 emissions, cloud data centers must track the carbon cost of their electricity supply and modulate their electricity consumption over time. The carbon cost of electricity depends on the source of generation. For example, a unit of energy generated by a coal plant will have a high carbon cost (i.e., emissions in terms of gCO2eq/kWh), while energy generated from a hydroelectric plant will have no emissions. The electricity the grid supplies is produced by a mix of generation sources, and the resulting carbon cost is a weighted average of the corresponding sources. Importantly, the generation mix varies from one region to another—based on the local power plants in each region—and also varies over time since the generation mix changes based on demand, the relative cost of generation, and intermittent generation from renewable sources.

Figure 1 depicts how the carbon cost differs by country/region and how it exhibits diurnal variations daily. In this case, Ontario tends to have a low but variable carbon cost because its energy mix consists of a large fraction of carbon-free nuclear and hydroelectric energy combined with some coal plants, which results in non-zero carbon intensity, and solar, which causes the diurnal fluctuations. California is similar but has a higher fraction of solar, which results in larger fluctuations, but also a higher fraction of coal plants, which elevates the average carbon intensity. The Netherlands also shows diurnal variation but with a higher average as it relies more on fossil-based electricity generation. By contrast, the carbon intensity of electricity in Iceland is nearly zero and flat due to its unique abundance of carbon-free geothermal energy.

2.3 Carbon-aware Cloud Scheduling

Many cloud workloads have both temporal flexibility and resource elasticity, which enables exploiting the temporal and spatial variations in energy's carbon intensity, as demonstrated in recent work [19, 29, 59, 68, 73]. To facilitate such efforts, commercial services, such as electricityMap [45] and WattTime [71], have emerged that aggregate data from grids in different parts of the world and expose grid energy's current and forecasted carbon intensity to cloud providers and users in real-time. Researchers, in turn, are exploiting this data to design carbon-aware schedulers that dynamically shift workloads across time and space to reduce emissions.

As mentioned above, temporal shifting involves moving delay-tolerant batch workloads to periods of low carbon intensity. In Figure 1, for instance, rather than running a batch job continuously in a *carbon-agnostic* manner, *suspend-resume* approaches execute the job in the "valleys", where the carbon cost is low, and suspend the job during peak periods. This technique has been explored in recent work [19, 59, 68, 73]. Threshold-based suspend-resume scheduling policies suspend jobs whenever the carbon cost rises above a certain threshold, while deadline-based methods choose the n lowest carbon cost periods between the arrival time and the deadline to execute the job. Importantly, a key drawback of *suspend-resume* methods, whether threshold-based or deadline-based, is that the carbon savings depend on the amount of time the user is willing to wait for their job to complete—a higher delay tolerance yields higher savings, but also a longer completion times.

Geographic or spatial shifting, in contrast, migrates jobs or workloads to regions with the greenest electricity grid [19, 52, 76, 77]. However, batch jobs often cannot exploit geographic shifting due to data privacy regulations, such as GDPR, that impose regional restrictions. Even when possible, spatially shifting jobs can incur high migration costs if it requires moving substantial state or data associated with the job. Since `CarbonScaler` focuses on batch jobs, spatial shifting is outside the scope of this paper. We discuss related work in spatial shifting in Section 7.

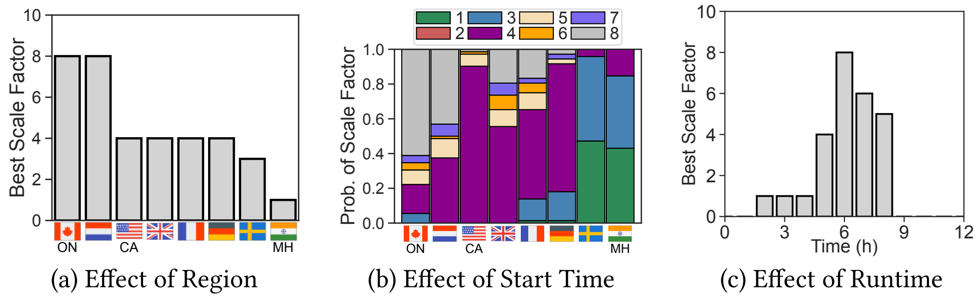


Fig. 3. *Best static scale factor varies across (a) geographical regions, (b) job start times, and (c) job execution.*

2.4 Carbon-Aware Autoscaling

Cloud workloads fall into two broad classes: interactive and batch. Since interactive workloads are latency-sensitive, they are not amenable to temporal shifting optimizations, and scaling is only beneficial in response to demand variations. Hence, our work focuses on distributed batch workloads, such as machine learning jobs, data analytics, and scientific computing simulations, which run on multiple machines. Given its benefits, elastic execution mechanisms are now built into many machine learning frameworks, such as Pytorch [55], data processing frameworks, such as Spark [5], as well as scheduling frameworks [25, 40, 61].

Although autoscaling can be applied from a cluster or cloud service provider perspective, our work focuses on the cloud *application's* perspective of carbon scaling, similar to cloud autoscaling. A typical autoscaler adjusts the number of servers based on the application demand, where higher demand can be measured in latency or average utilization of provisioned resources [6, 26]. However, `CarbonScaler` adjusts the number of servers based on the carbon intensity of electricity. In both cases, the cloud application operates under an abstract view of the underlying servers and can allocate as many as needed. We discuss carbon scaling from a cloud providers' perspective in Section 6.

Elastic scaling capabilities have enabled designing policies that scale resources based on electricity's carbon intensity [65]. For example, a policy might scale up an application's resources when carbon is low and either halt or scale down when carbon is high. However, unlike traditional autoscalers, a distributed batch application often has communication bottlenecks that vary widely across applications and govern the scaling benefits. Figure 2 depicts the scaling behavior of four deep learning training jobs, which use Horovod and PyTorch for elastic scaling and two MPI tasks that perform scientific computations. As shown, ResNet18 training and the larger N-body MPI computation show a linear increase in throughput as the number of servers increases, indicating linear scaling behavior. In contrast, the smaller N-body MPI computation exhibits diminishing growth in throughput with increased server allocation. Finally, VGG18 and ResNet50 training tasks exhibit a slower increase in throughput due to scaling bottlenecks. These differences in scaling behavior, as well as the variability in carbon intensity and execution constraints (e.g., start time and deadline), should be considered by a carbon scaling approach when optimizing for carbon savings.

3 CARBONSCALER DESIGN

This section motivates `CarbonScaler` in the context of prior work, formulates the carbon scaling problem, and then presents `CarbonScaler`'s design.

3.1 Motivation

The *suspend-resume* and temporal shifting policies proposed in prior work can reduce the carbon emissions of delay-tolerant workloads [73]. However, they suffer from two drawbacks. First, the carbon intensity of grid-supplied electricity changes slowly, and thus, there may be arbitrarily long intervals (e.g., many hours) of high carbon periods where jobs remain suspended and make no progress. Such suspensions delay completion times, with a 7-10× increase in completion times in some cases [65]. Second, when batch jobs have limited temporal flexibility and cannot be significantly shifted, the effectiveness of these methods is significantly reduced.

To overcome the drawbacks above, our paper presents *CarbonScaler*, a new approach that exploits the *resource elasticity* of cloud workloads to dynamically vary the amount of resources allocated to applications in response to fluctuations in the carbon cost of their energy supply. `CarbonScaler`'s key insight is that *the scale which yields the minimum carbon consumption, not only depends on the application characteristics but also the variations in carbon intensity across geographical regions, application start times within a given region, and the runtime of an application following a specific start time.* Importantly, current approaches for selecting an application's scale factor do not apply directly to this context, necessitating a new approach. Specifically, analytic performance models of an application, such as those used in cloud auto-scaling approaches, only account for application performance characteristics and do not consider the impact of time-varying carbon intensity on the scale factor. Similarly, the state-of-the-art approach for leveraging workload elasticity demonstrates that this scale factor varies across applications [65] but does not provide an algorithm for choosing this scale factor or show how carbon intensity variations should be considered when doing so.

To demonstrate the impact of application characteristics and temporal variations in carbon intensity on the scale factor, we consider an *oracle* approach for choosing the best static scale factor for a 24hr job on a per-region, per start time, and per-timeslot for ML (ResNet 18). Figure 3(a) shows that the best static scale factor for a given application varies significantly, from 1× to 8×, across geographical regions, as different regions exhibit different variations in carbon intensity. Figure 3(b) presents the distribution of best static scale factors across all the possible start times for various regions for one of them. We observe that there is no single static scale that works for a given region due to the differences in their carbon intensity profiles. In addition, the static scale must also be adapted depending on when an application executes. Finally, as shown in Figure 3(c), the best static scale factor can even vary during application execution time, where the lowest carbon consumption is achieved by running the application with *five different* scaling factors. Further, neither application performance models, which are inherently carbon-oblivious, nor state-of-the-art carbon-aware techniques, such as Ecovisor [65] or Wait Awhile[73], can realize this oracle approach.

The dynamicity of choosing the best static scale factor motivates the design of `CarbonScaler`, which adapts the operating scale factor for each application depending on where and when it executes. `CarbonScaler` avoids computing the best static scale factor across application runs in an exhaustive brute-force manner and instead computes a carbon-aware schedule using a greedy approach. We next formulate the problem and present our dynamic scaling algorithm.

3.2 Problem Formulation

Similar to cloud autoscalers that scale each application *independently*, a carbon scaler operates independently on each cloud application that wishes to optimize its carbon emissions. When a new batch application arrives at time t, it specifies (i) the minimum number of servers, m, that it needs to run, where $m \geq 1$, and (ii) the maximum number of servers M that can be allocated to it, $M \geq m$. The carbon scaler can then vary the servers allocated to the application between m and M. Suppose that l is the estimated job length when executing on the baseline allocation of m servers.[3] By default, we assume that the desired job completion time is $T = t + l$, which means that jobs

[3]The job length l can be estimated using profiling and modeling [57, 63] or using prior execution history. For example, [72] reports that 65% of batch jobs see repeated execution at least five times within a two-month period.

should complete “on time” with no delays. Although T must be at least $t + l$ for all jobs, some delay tolerant jobs have significant temporal flexibility and can *optionally* specify a longer completion time T such that $T{>}t + l$. The value $T - (t + l)$ represents the slack available when executing the job. This slack captures the willingness of users to wait in order to increase their carbon savings. The default case of $T = t + l$ assumes on-time completion and zero slack.

The completion time T specifies the *temporal flexibility* (delay tolerance) available to the job, while the maximum server allocation M specifies the *resource elasticity* of the job. The parameters T and M can be specified differently to obtain a range of carbon scaling behaviors. For example, when $T = t + l$, the application has no temporal flexibility and cannot be subjected to suspend-resume mechanisms. In this case, the job can only exploit resource elasticity by scaling up to M workers during low carbon periods and must be completed on time with no delays. In contrast, when $M = m$, the job has no resource flexibility, and the carbon scaler is limited to performing only suspend-resume optimizations with a static number of servers, m, while also ensuring it completes the job by the specified completion time T. Of course, when $T > t + l$ and $M > m$, a carbon scaler has the most flexibility and can exploit both resource elasticity and temporal shifting via suspend-resume. Our goal is to design a carbon scaler that minimizes a job's carbon emissions subject to the available resource elasticity and temporal flexibility.

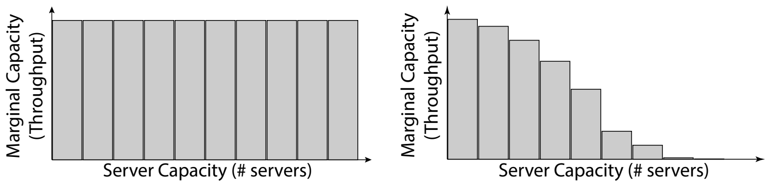
3.3 Basic Design

When a new batch job arrives, our system, which we refer to as `CarbonScaler`, computes an *initial* schedule for executing the job through completion. The execution schedule specifies how many servers to allocate to the batch job over time and when to dynamically change the allocation in response to variations in carbon cost. This initial schedule is based on a forecast of future carbon cost, as well as the expected progress of the job over time based on its resource allocation. As the job executes, `CarbonScaler` adjusts its schedule periodically if it encounters forecast errors or deviations in the job's expected progress — to ensure completion by the specified completion time T. Observed deviations can occur due to profiling errors, from network and locality interference [36], or resource procurement denials. We discuss these issues further in §5.7.

`CarbonScaler` assumes that carbon cost forecasts are available; commercial services [43, 71] provide such forecasts for up to four days with high accuracy in most locations. Since the application specifies its temporal flexibility (in terms of completion time T) and its resource elasticity (in terms of the varying server allocation from m to M), `CarbonScaler`'s schedule responds to fluctuations in forecasted carbon cost by scaling down or completely suspending the job when the carbon cost is high and opportunistically scaling up when the carbon cost is low.

Different clustered batch applications will have different scaling behaviors, as shown in Figure 2, which should be considered when scaling an application's server capacity between the specified range of m to M. As noted in Figure 2, applications' throughput either increases sub-linearly or increases somewhat linearly initially and then shows diminishing returns with an additional increase in server capacity. This behavior is a direct consequence of Amdahl's law [3], which states that the speedup of a parallel application is limited by the amount of sequential code within it — adding server capacity only speeds up the parallel component of the application. Software bottlenecks, such as synchronization overheads, also limit the ability to scale up.

`CarbonScaler` considers this scaling behavior in terms of a *marginal capacity curve*, shown in Figure 4, which captures the incremental increase in application capacity (i.e., throughput) for each unit increase in server capacity. The ideal case of linear scaling translates to a *flat* marginal capacity curve where each additional server results in a unit increase in (normalized) application capacity (see Figure 4(a)). Most applications will have a diminishing marginal capacity curve, where marginal capacity decreases monotonically with an increase in the server capacity (see Figure 4(b)).



(a) Linear scaling (b) Diminishing scaling

Fig. 4. *Example marginal capacity curves.*

The marginal capacity curve and the carbon intensity curve can then be used to scale the application up or down in a carbon-efficient manner. To do so, the marginal capacity curve is normalized by the forecasted carbon cost in each time step to compute the *marginal capacity per unit carbon* — the marginal work done per unit carbon. CarbonScaler then adds server capacity to the time slots that maximize the work done per unit of carbon. By doing so, CarbonScaler allocates *more server resources when the carbon cost is low since more marginal work can be done at a lower carbon cost.* CarbonScaler will incrementally add servers to various time slots until sufficient server capacity has been added to complete the job within the desired completion time T, thereby yielding a carbon-efficient execution schedule that optimizes the carbon emissions.

In practice, each application can have multiple marginal capacity curves, each representing a different phase of its execution. For example, a MapReduce job can have different scaling behaviors and marginal capacity curves for its map and reduce phases. For ease of exposition, our discussion below assumes a single marginal capacity curve per application. However, our approach generalizes to multiple marginal capacity curves by considering the appropriate scaling curve in each time slot that corresponds to the current phase of the application's execution.

3.4 Carbon Scaling Algorithm

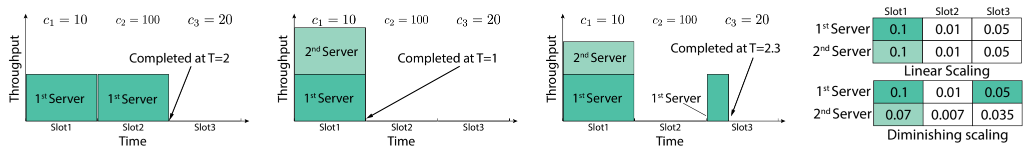
CarbonScaler relies on the knowledge of application scalability profile, carbon intensity forecast, and other job constraints to decide when to i) horizontally scale resources up or down or ii) suspend execution to ensure minimum carbon consumption. As noted earlier, when a new job arrives at time t, it specifies a *desired* completion time (i.e., a "deadline") of time T. We also assume that the marginal capacity curve of the application is obtained by profiling the application offline (see Section 4.1) and is known at arrival time. Finally, the algorithm takes the carbon cost forecast c, which we assume to be correct. We analyze the impact of inaccurate forecasts in Section 5.7.

We assume that the interval $[t, T]$ is discretized into smaller fixed-length intervals (e.g., 15 minutes or an hour), and the number of servers allocated to the job can be changed at the start of each interval. Suppose that there are n time intervals between $[t, T]$, $n \geq 1$. Let $c_1, c_2, ..., c_n$ denote the forecasted carbon cost in each interval $i, i \in [t, T]$. Suppose that the marginal capacity curve is denoted by $MC_m, MC_{m+1}, ..., MC_M$, where MC_j is the marginal capacity increase after allocating the j-th servers, $j \in [m, M]$. Since the estimated job length is l when executing with minimum server capacity m, the total work the job needs to perform is $W = l \cdot MC_m$. Our algorithm must compute a schedule where the aggregate server capacity allocated to the job over $[t, T]$ can perform this work before the completion time T, minimizing carbon emissions.

The aforementioned carbon scaling problem is a marginal allocation problem of discrete resources, which is known to yield an optimal solution in many cases [22]. Our greedy Carbon Scaling Algorithm, detailed in Algorithm 1, builds on the algorithm and theoretical results in [22]. We provide the requirements and the optimality proof of our greedy Carbon Scaling Algorithm in appendix A. The Algorithm, first computes the *marginal capacity per unit carbon* in each time

Algorithm 1: `Carbon Scaling Algorithm()`

```
    Input: Marginal capacity ($MC$), time slots $[t, T]$, carbon cost forecast ($c$), total work ($W$)
    Output: Execution Schedule $S$
1   $S \leftarrow [0..0]$;
2   $L \leftarrow [\,]$ ;
3   for $i \in [t, T]$ do
4       for $j \in [m, M]$ do
5           $L$.append($i, j, MC_j/c_i$);
6   $L \leftarrow$ Sort($L$) ; // w.r.t. Norm. Marginal Cap.
7   $w \leftarrow 0$ ;
8   while $w < W$ do
9       $i, j, * \leftarrow L.pop()$; // next highest $MC_j/c_i$
10      $S[i] = j$; // increase allocation in slot $i$
11      $w.update(S)$ ;
12  return $S$
```



(a) Carbon-agnostic (b) Linear scaling (c) Diminishing scaling (d) MC per unit carbon

Fig. 5. *An illustrative example of our carbon scaling algorithm at work.*

interval i by normalizing the MC curve with carbon cost c_i in that interval (line 5).[4] That is, the marginal capacity per unit carbon in time interval i is $MC_m/c_i, MC_{m+1}/c_i, ..., MC_M/c_i$. The greedy algorithm then iteratively and incrementally allocates server capacity to various time slots in order of decreasing *marginal capacity per unit carbon* (lines 6-11). For each iteration, the algorithm chooses the interval i from [1, n] such that allocating incremental server capacity to that time slot maximizes the work done per unit carbon (i.e., chooses the interval with the greatest MC_j/c_i across all intervals). After allocating server capacity to that interval, it iteratively determines the next interval where allocating additional server capacity yields the next highest work done per unit of carbon.

Note that our greedy algorithm may allocate additional capacity to the same interval as the previous iteration of its marginal work done per unit carbon continues to be the highest across all intervals. Otherwise, a new time interval with the next highest marginal work done per unit is chosen for allocating server capacity. Also, when a time interval is initially chosen by the greedy algorithm for capacity allocation, it must be allocated the minimum requirement of m servers, after which the allocation can be increased incrementally by one in subsequent steps. Similarly, if a time slot reaches the maximum allocation of M servers, it is not considered further by the greedy algorithm. The process continues until sufficient capacity has been allocated across the n time intervals to complete W units of work. This yields an initial schedule where each time interval has either a zero allocation (causing the job to be suspended in that period) or a non-zero allocation between m and M, with the server allocation potentially changing at interval boundaries.

Example. To illustrate our carbon scaling algorithm, consider a job of length 2 that arrives at $t = 0$ and needs to finish by $T = 3$. Suppose that the job needs to execute on at least one server ($m = 1$) and at most two servers ($M = 2$). Carbon-agnostic execution will run the job as soon as it arrives, and it will complete at time 2, as shown in Figure 5(a). Suppose that the forecasted carbon cost in

[4]If an application has multiple marginal capacity curves, we select the one for the execution phase in time slot i.

time slots 1, 2, and 3 is $c_1 = 10$ ("low"), $c_2 = 100$ ("high"), and $c_3 = 20$ ("medium"), respectively. First, assume that the job has ideal scaling behavior and a flat marginal capacity curve of $MC_1 = 1$ and $MC_2 = 1$. The algorithm simply allocates two servers to the job in slot 1, since it has the lowest carbon cost and the highest marginal capacity per unit carbon. As shown in Figure 5(b), such a job runs with two servers and terminates at the end of slot 1.

Next, assume a job with a diminishing marginal capacity curve, given by $MC_1 = 1$ and $MC_2 = 0.7$. Figure 5(d) shows the marginal capacity per unit cost table (MC_j/c_i) for all three slots. The greedy algorithm allocates the first server to slot 1, since it has the highest marginal capacity per unit cost of 0.1. In the next iteration, the greedy algorithm allocates a second server to slot 1 as it still has the highest marginal capacity per unit cost ($MC_2/c_1 = 0.07$). Although two servers have been allocated, the total work done by these two servers is only 1.7 ($MC_1 + MC_2$), which cannot complete the job of length 2 ($W = 2$). The algorithm then allocates another server to slot 3, which has the next highest marginal capacity per unit cost ($MC_1/c_3 = 0.05$). This yields a schedule where the job is given 2 servers in slot 1, zero in slot 2, and one server in slot 3. The job only runs for one-third of slot 3 before it completes. The example also illustrates a tradeoff where CarbonScaler reduces the emissions compared to carbon-agnostic execution (from 110 to 40 carbon units) but increases cloud costs by 15% due to the need for a third server. The tradeoff between carbon saving and cost overheads is fundamental to carbon-aware computing, as demonstrated by prior work [29].

Periodic Schedule Recomputation. Once the algorithm computes an initial schedule, CarbonScaler can begin execution of the job by auto-scaling it up or down, or suspending it, in each time slot as per the schedule. CarbonScaler continuously monitors the work done ("job progress") and the emissions of the job over the course of its execution. Recall that the initial schedule is computed based on a *forecasted* carbon cost and an *estimated* marginal capacity curve derived from profiling, both of which may have errors in their estimates. Similar to weather forecasts, carbon forecasts can have errors, especially over the period of multiple days [43, 44]. Similarly, the marginal capacity curves may not be exact since production environments may differ somewhat from the profiling environment [36, 57, 63]. These errors can cause deviations in the expected work done or the expected carbon emissions as estimated by the initial schedule.

To be robust to carbon prediction or profile estimation errors, CarbonScaler compares the expected work and carbon emissions to the estimates in the schedule at the end of each time interval. If the deviations exceed a threshold, it recomputes the schedule for the remainder of the job's execution from the current time t' to the completion time T. When doing so, CarbonScaler can use an updated carbon forecast if available, since such forecasts are often updated every few hours, similar to weather forecasts. Thus, if the progress deviates from the plan (e.g., due to profiling errors), CarbonScaler will recompute the schedule to ensure the highest carbon savings. Since some batch jobs can execute for days [70], such schedule adjustments provide robustness to prediction errors and ensure timely job completion while minimizing carbon emissions.

Run Time Complexity. In Algorithm 1, the time complexity of computing the marginal capacity per unit carbon (steps: 3-5) is $O(n.M)$, list sorting is $O(nM \log nM)$, and computing the schedule is $O(nM)$ (steps: 8-11). The total complexity is $O(nM + nM \log nM) \approx O(nM \log nM)$.

4 CARBONSCALER IMPLEMENTATION

This section describes CarbonScaler's implementation, which optimizes the carbon emissions of distributed batch cloud workloads. Our system comprises three main components: (1) Carbon Profiler, which uses offline profiling to estimate marginal capacity (MC) curves and energy usage of jobs, (2) Carbon AutoScaler is our cloud-based carbon scaling system implemented in Kubernetes [41], and (3) Carbon Advisor, which simulates the execution of the jobs to estimate

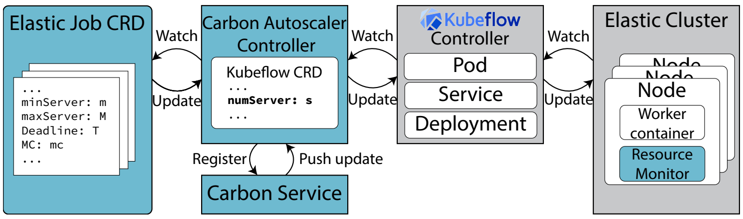


Fig. 6. *An overview of Carbon AutoScaler.*

carbon reduction under different deployment configurations. `CarbonScaler` is implemented in Go using ~2.5*k* SLOC. The code is available at https://github.com/umassos/CarbonScaler.

4.1 Carbon Profiler

`CarbonScaler` requires the marginal capacity curve of a job for carbon-aware scaling. `Carbon Profiler` performs a one-time offline profiling of a new job to derive its marginal capacity curve. To do so, it runs the job with server allocations ranging from the job-specified minimum number of servers, m, to the maximum number of servers, M, and records the work done at each allocation. To minimize the profiling overhead, `Carbon Profiler` runs the job for a small, configurable amount of time α (up to a few minutes) and varies the resource allocation with a granularity β, which depends on M. If $\beta > 1$, `Carbon Profiler` interpolates the recorded measurements to obtain a complete marginal capacity curve. Finally, the marginal capacity curves are valid for a computing environment identical to the profiling environment. The scaling behavior and the expected savings may change if the environment is significantly different, necessitating environment-specific profiling or an online update of the capacity curves. `CarbonScaler` also allows substituting `Carbon Profiler` with alternative workload profiling approaches from prior work [13, 38, 54, 56–58, 63].

4.2 Carbon AutoScaler

Figure 6 shows an overview of `Carbon AutoScaler` that uses Kubeflow [40] to implement our `Carbon Scaling Algorithm` from §3.4. The incoming elastic batch applications use Kubernetes' Custom Resource Definition (CRD), written in `.yaml` format. `Carbon AutoScaler` follows Kubernetes standards in defining its user-facing interface. In this case, the user extends the normal job specification by adding extra `Carbon AutoScaler`-specific maps that provide scaling and scheduling information, including minimum m and maximum M number of servers, completion time T, and an estimated job length l. The user also specifies methods for obtaining the marginal capacity curve, where the current default is profiling. The user then submits the jobs to `Carbon AutoScaler` using standard using Kubernetes APIs such as `kubectl`.

We implement `Carbon AutoScaler` as a controller that sits on top of the Kubeflow training operator and leverages its core resource management functionality for clustered batch jobs, such as ML training and MPI. `Carbon AutoScaler` first runs the `Carbon Scaling Algorithm` to compute the initial schedule for each job. To do so, `Carbon AutoScaler` tracks carbon intensity using a dedicated service that provides the instantaneous and forecasted carbon intensity. Then, `Carbon AutoScaler` informs the Kubeflow training operator to execute the schedule by modifying the Kubeflow job specification to scale the resources allocated to the job, such as the number of replicas. `Carbon AutoScaler` is also in charge of maintaining the job status of the Kubeflow operator. `Carbon AutoScaler` implements resource-level and application-level monitoring. `Carbon AutoScaler` implements additional Kubernetes services to monitor resource usage, energy usage, and carbon usage over time. We track CPU usage using Kubernetes Metrics Server [64], CPU energy usage using Running Average Power Limiting (RAPL) [17] interfaces and PowerAPI [11], and GPU energy usage using NVIDIA Data Center GPU Manager (DCGM) [53]. The resource and

Name	Implementation	Epochs	BatchSize	Power (W)
N-Body Simulation (10,000)	MPI	138000	NA	CPU (60)
N-Body Simulation (100,000)	MPI	1500	NA	CPU (60)
Resnet18 (Tiny ImageNet)	Pytorch	173	256	CPU+GPU (210)
EfficientNetB1 (ImageNet)	Pytorch	45	96	CPU+GPU (210)
VGG16 (ImageNet)	Pytorch	31	96	CPU+GPU (210)

Table 1. *Details of elastic workloads in evaluation. Epochs represent the number of epochs needed for a 24hr job.*

power monitoring can include other resources such as storage and network. `Carbon AutoScaler` monitors application-level metrics such as progress and throughput via application-level interfaces.

Finally, `Carbon AutoScaler` registers a reconcile callback function, which is called when the carbon intensity changes and when applications report their progress. This enables `Carbon AutoScaler` to detect divergence in progress, throughput, or carbon intensity. `CarbonScaler` then recomputes the schedule as explained in §3.4.

4.3 Carbon Advisor

`Carbon Advisor` enables pre-deployment analysis of the carbon scaling algorithm in an environment that simulates the operation of `Carbon AutoScaler`. `Carbon Advisor` takes, as input, a carbon trace, job start time, deadline, job length, and `CarbonScaler`-specific parameters, such as range of server allocations $[m, M]$ and marginal capacity curve. The fidelity of `Carbon Advisor` depends on the accuracy of the marginal capacity profile for the application. In Section 5, we demonstrate the high fidelity of `Carbon Advisor` in estimating the carbon savings from different carbon-aware scaling policies. The `Carbon Advisor` simulates the running of the job and reports savings for carbon-aware scaling policies. Additionally, the `Carbon Advisor` enables simulating various kinds of errors to ensure the robustness of the predictions, as described in Section 5.7. The simple plug-and-play nature of the tool allows application developers to perform what-if scenarios and explore a wide range of parameters before actual deployment. For example, users can explore the benefits of extending their waiting time and its impact on carbon savings. `Carbon Advisor` also enables key high-level analysis by default, such as computing the distribution of carbon savings across different start times of the year. Finally, to facilitate initial exploration, we plan to provide carbon traces and marginal capacity curves used in the paper alongside the tool.

5 EXPERIMENTAL EVALUATION

This section evaluates the performance of `CarbonScaler` using our prototype implementation, described in Section 4. We augment the prototype evaluation results with additional large-scale analysis that leverages `Carbon Advisor`.

5.1 Experimental Setup

Workload. Table 1 describes the elastic workloads we use for evaluating `CarbonScaler` and their specifications. The workloads span both CPU- and GPU-intensive applications such as the N-body problem [1] implemented using MPI [24] and machine learning models, including ResNet [31], EfficientNet [69], and VGG [69] implemented using Pytorch [55]. The table shows the base configurations and power measurements for jobs that need 24hrs to finish. The chosen workloads have a wide-range of scaling characteristics (shown in Figure 2), configurations, and energy requirements.
Infrastructure. We deployed `CarbonScaler` in two different settings to demonstrate its adaptability to the underlying infrastructure. For CPU-intensive workloads, we used a local computing cluster consisting of 8 servers, each equipped with a 16-core Xeon CPU E5-2620, connected through a 10G network. For GPU-intensive workloads, we deployed `CarbonScaler` on Amazon Web Services (AWS) using 8 `p2.xlarge` instances, each equipped with NVIDIA K80 GPU.

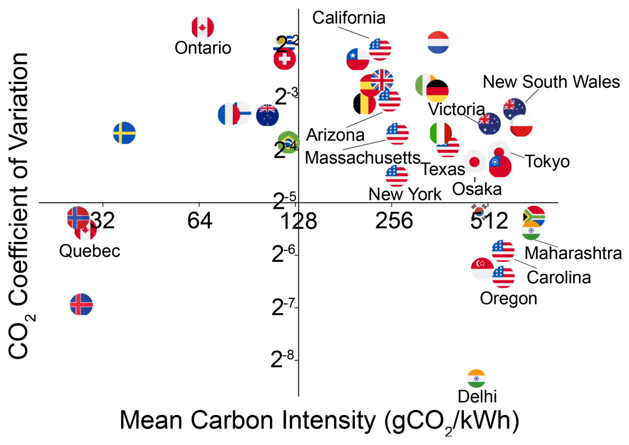


Fig. 7. *Most cloud regions globally have a high carbon cost, but also show significant daily variations, providing an opportunity for CarbonScaler to optimize carbon emissions.*

Carbon Traces. We collected carbon traces for different geographical locations using electricityMap [45], an online service that provides real-time and archival carbon intensity information. We use average carbon intensity values, measured in grams of carbon dioxide equivalent per kilowatt-hour (gCO2eq/kWh), provided at hourly granularity. The collected carbon traces span from January 2020 to December 2022, we specify the duration for each trace where it is used.

To choose representative regions for our evaluation, we analyzed the average carbon intensity and the coefficient of variation (computed as standard deviation over mean) for different AWS regions. Figure 7 shows the results for 37 regions. Most regions have high carbon intensity but also show high daily variations, while some have low carbon intensity with similarly high daily variations. Since `suspend-resume` and `CarbonScaler` rely on these high variations to reduce emissions, the figure indicates that both techniques will be effective in the majority of low-carbon as well as high-carbon cloud regions. A few cloud regions have stable carbon costs (i.e., low variations), including low carbon regions such as Iceland and Sweden, and high carbon regions such as India and Singapore. The effectiveness of `suspend-resume` and `CarbonScaler` is diminished in such cloud regions as changing the execution time and scale does not alter the carbon intensity. Still, such regions are a small minority of the total cloud regions in a global cloud platform such as AWS. Based on this analysis, we choose Netherlands (🇳🇱) as a representative high carbon region and Ontario, Canada (🇨🇦) as an example of a low carbon region for our subsequent experiments. Nonetheless, we evaluate the potential savings across regions in Section 5.6.

Baselines Policies. We evaluate the performance of `CarbonScaler` against three baseline policies: `carbon-agnostic`, `suspend-resume`, and `static-scale`. The `carbon-agnostic` is a simple policy that runs a job without considering carbon emissions and represents the status quo. The `suspend-resume` policy is inspired by prior work [19, 73]. As mentioned in §2.3, `suspend-resume` can be implemented in two ways: threshold-based, which uses a carbon threshold to suspend-resume a job in a deadline-unaware manner, and deadline-based, which chooses the k lowest carbon periods before the specified deadline for execution. In this case, `suspend-resume` defaults to `carbon-agnostic` policy when the completion time equals the job length ($T = l$), i.e., no slack, since execution cannot be deferred. This policy acts as a baseline for temporal shifting scenarios where we assume a job has a completion time higher than the job length ($T > l$). Finally, `static-scale` is another policy inspired by prior work [65], where an application picks the lowest carbon intensity points and runs with a certain *static* scale factor to utilize the carbon intensity variations better. This is our default baseline for scenarios where we evaluate `CarbonScaler` for its ability to leverage

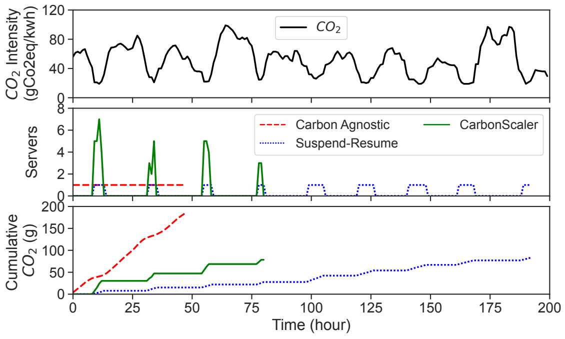


Fig. 8. *CarbonScaler in action for a 48hrs long N-body MPI job (N=100k), where $T = 2 \times l$.*

workload elasticity and scaling. Unless stated otherwise, we report the mean across 15 runs for our cloud experiments and 100 runs for `Carbon Advisor`'s simulated executions.

Carbon Advisor Fidelity. To demonstrate the fidelity of the simulator, we compare the carbon savings estimates from `Carbon Advisor` with the results from various real experiments in the evaluation. `Carbon Advisor` estimates have an average error of less than 5%, demonstrating the high accuracy of our simulation results based on `Carbon Advisor`.

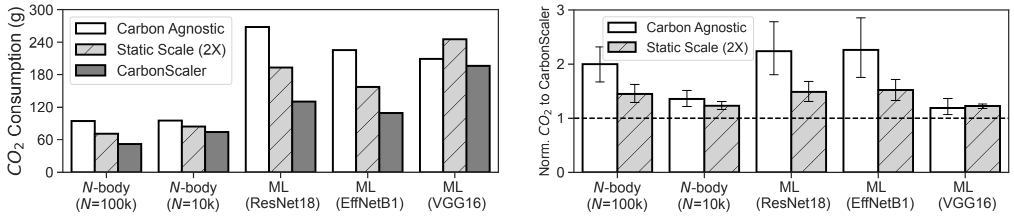
5.2 CarbonScaler in Action

To show `CarbonScaler` in action, we ran a 48hr N-body MPI job on our CPU cluster and compared its execution to the threshold-based `suspend-resume` (deadline-unaware) and `carbon-agnostic` policies. As shown in Figure 8, the `carbon-agnostic` policy starts the job as soon as it arrives and finishes in 48hrs at the cost of 184g of CO_2 emissions. The `suspend-resume` policy *suspends* the job during high carbon intensity periods and waits for the carbon intensity to fall below a threshold (25th percentile in this case) to *resume* the job. By leveraging temporal flexibility, `suspend-resume` saved 45% carbon compared to the `carbon-agnostic` policy but increased the job completion time by 4×. Finally, we set the desired completion time T to be 96hrs, i.e., $T = 2 \times l$, and execute our proposed `CarbonScaler` policy. `CarbonScaler` scales the number of servers depending on the application's scaling properties and the carbon cost at a given time. As a result, `CarbonScaler` achieves 42% carbon saving over `carbon-agnostic` policy. `CarbonScaler` achieves comparable savings with `suspend-resume` while also reducing the job completion time to 2× of `carbon-agnostic` policy.

5.3 Impact of Workload Elasticity

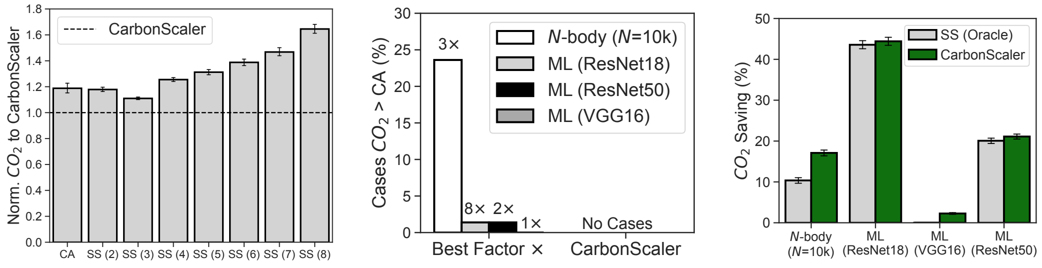
The two key aspects that impact carbon savings from `CarbonScaler` are temporal flexibility and workload elasticity. While prior work necessitates temporal flexibility for carbon savings, `CarbonScaler` can achieve significant savings by leveraging workload elasticity even when no temporal flexibility is available. The extent of savings depends on the scalability properties of the workload: a highly scalable job (with flat or close to flat marginal capacity) can achieve higher savings, as illustrated for the simple workload in Figure 5. To demonstrate the elasticity effect, we limit the job completion time to the job length, i.e., $T = l$, which means no temporal flexibility is available. We run 24hrs long jobs for various applications in Table 1 using `carbon-agnostic` policy, `static-scale` (2×), and `CarbonScaler`.

Figure 9 shows the performance of the three policies for different workloads. Figure 9(a) compares the absolute carbon footprint of the three policies and shows that the `CarbonScaler` highest savings are for highly scalable workloads. For example, for N-body (N=100k) and ML (ResNet18), `CarbonScaler` saves up to 140 and 63 (gCO2eq) compared to `carbon-agnostic` and `static-scale` (2×), respectively. To demonstrate the superiority of `CarbonScaler`, independent of the task and start-time dependent carbon consumption, we compare the normalized carbon savings of different policies to `CarbonScaler`. Figure 9(b) compares the performance of all policies to `CarbonScaler`,



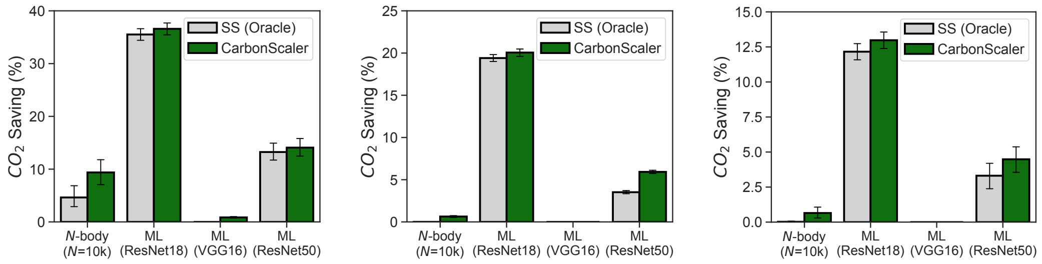
(a) Carbon footprint (b) Performance w.r.t CarbonScaler

Fig. 9. *Carbon footprint and performance of different workloads scheduled under carbon-agnostic, static-scale (2×), and* CarbonScaler*, in Ontario, Canada, where $T = l$ (i.e., no slack and on-time completion).*



(a) (b) (c)

Fig. 10. CarbonScaler *vs. the best static scale (SS) factor in Ontario, Canada. Carbon emissions of various static scale factors compared to* CarbonScaler *(a), percentage of start times when a policy consumes more carbon than* carbon-agnostic *(b), and static scale oracle against* CarbonScaler *for multiple applications (c).*

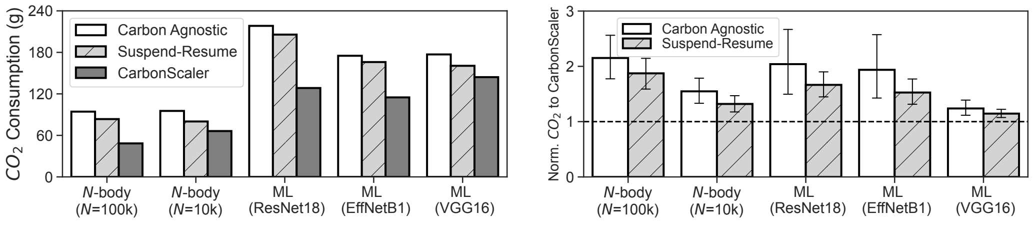


(a) Netherlands (b) California, US (c) Texas, US

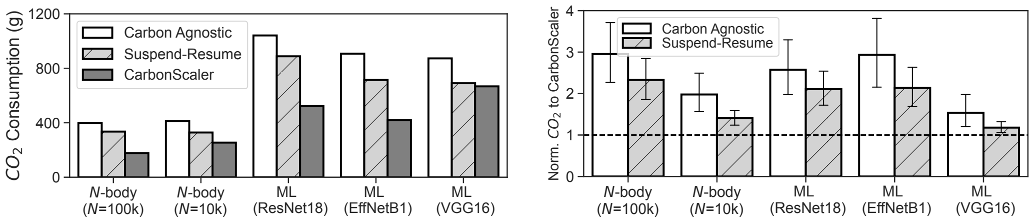
Fig. 11. *Comparing* CarbonScaler *with static scale oracle in multiple regions.*

where the whiskers represent the 95th percentile confidence interval and the horizontal line represents the performance of CarbonScaler. The figure shows that, aside from the saving, workloads, and start times, CarbonScaler demonstrates the ability to outperform all other policies. In particular, CarbonScaler uses 33% and 20% less carbon than carbon-agnostic and static-scale (2×), respectively. The figure also shows that, since the static-scale does not consider the job's scalability properties, it can instead *increase* the carbon consumption for some workloads by as much as 20% by scaling the job beyond a single *optimal* scale factor. On the other hand, CarbonScaler is cognizant of scaling behavior and picks a different scale at each time slot that has the highest work done per unit carbon cost, yielding minimum carbon consumption.

To further demonstrate CarbonScaler benefits over the best static scale factor, we use Carbon Advisor to compare CarbonScaler against oracle-based static scale factors. Figure 10(a) shows the performance of all scale factors and CarbonScaler for N-body (N=10k). The static scaling consumes 17-65% more carbon than CarbonScaler. While the static-scale policy can reduce



(a) Ontario, Canada



(b) Netherlands

Fig. 12. *Carbon footprint and normalized performance of different workloads and policies, where $T = 1.5 \times l$.*

carbon emissions compared to `carbon-agnostic` for scale factors 2 and 3, it can consume more carbon at higher scale factors due to the non-linear scalability of the workloads. The potential increase in carbon consumption is not only true for an arbitrary non-optimal scale factor; even the best scale factor for each start time can consume more carbon than `carbon-agnostic`. Figure 10(b) shows the probability that the best scale factor (on top of each bar) yields a higher consumption than the `carbon-agnostic` operation. As shown, certain instances always exist where this best scale factor performs worse than `carbon-agnostic`. Perhaps the only exception is ML (VGG16), a non-scalable application, where the best scale factor is the `carbon-agnostic` (1×).

As opposed to `CarbonScaler`, the best static scale factor may not be optimal for all the time slots during the execution of a job, resulting in higher carbon emissions. In Figure 10(c), we show the additional savings from adapting the scale factor during the execution of a job for multiple applications. As demonstrated, `CarbonScaler` outperforms the static scale oracle by 1.2% to 8%, depending on the job's scalability characteristics. Figure 11 extends the evaluation of 10(c) and shows how `CarbonScaler` outperforms the oracle `static-scale` in different regions, even when carbon savings are limited. However, it is worth noting that static state oracle is the artifact of our implementation. Neither application performance models, which are inherently carbon-oblivious, nor state-of-the-art carbon-aware techniques, such as Ecovisor [65] or Wait Awhile[73], can realize this optimal oracle approach.

Key Takeaway. *CarbonScaler better leverages the workload elasticity by choosing dynamic scale factors depending on the job scalability characteristics and carbon intensity for each start time for the job and each time slot during a job's execution.*

5.4 Impact of Temporal Flexibility

In addition to workload elasticity, temporal flexibility can be an important source of carbon savings for delay-tolerant jobs. We evaluate the impact of temporal flexibility by running workloads from Table 1 using `carbon-agnostic` policy, `suspend-resume` policy, and `CarbonScaler` with extended completion times where $T > l$. To ensure that the `suspend-resume` respects the job-specified completion time, we use the deadline-aware version of the `suspend-resume` policy [73]. Figure 12 shows the carbon consumption (left) and performance of different policies (right) when *running* the workloads with 24 hrs length l, and 36 hrs as completion time T, $T = 1.5 \times l$, across two locations. `CarbonScaler` is better at exploiting the temporal flexibility and outperforms the

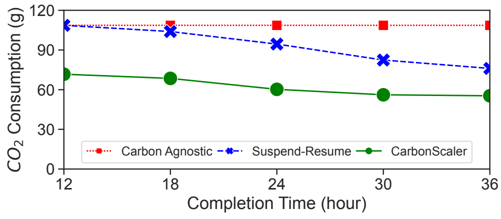


Fig. 13. *Effect of completion time on the carbon footprint of a 12hrs long ResNet18 job in Ontario, Canada.*

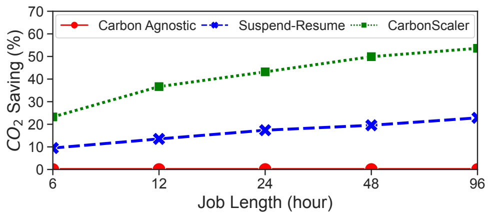


Fig. 14. *Effect of job length on CO_2 savings for an N-Body (N=100k) job in Ontario, Canada, $T = 1.5 \times l$.*

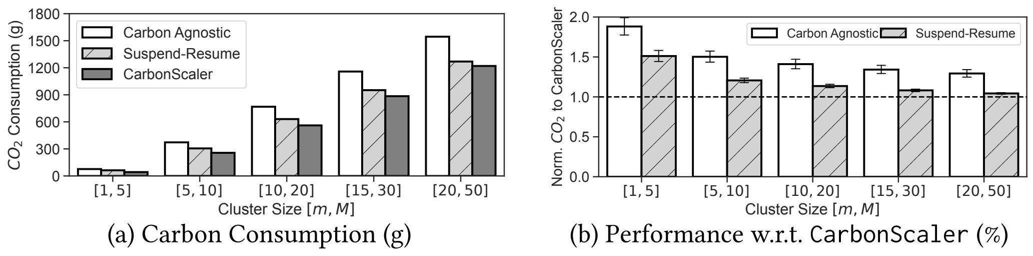


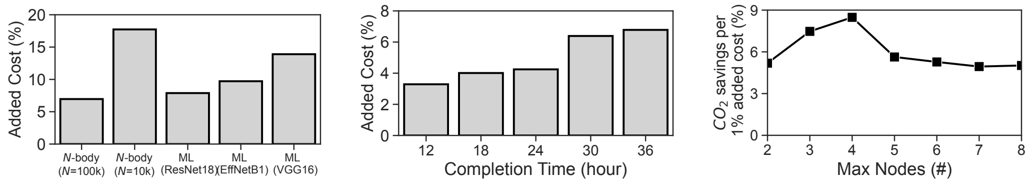
Fig. 15. *Carbon consumption and normalized performance of 24-hour N-body (N=100k) MPI job with different cluster sizes in Ontario, Canada, where $T = 1.5 \times l$.*

suspend-resume policy for all workloads. As shown, CarbonScaler is able to save 36% and 22% compared to carbon-agnostic and suspend-resume in Ontario, Canada, and 51% and 37% in the Netherlands for the highly scalable ML (ResNet18). On the other hand, for less scalable workloads such as ML(VGG16), most of the carbon savings of CarbonScaler stem from time-shifting, yielding comparable savings to suspend-resume. The right column of the figure also demonstrates the superiority of CarbonScaler aside from the carbon savings, which is start-time and task dependent.

Effect of Completion Time. Prior results have demonstrated that temporal flexibility can yield significant savings. Figure 13 evaluates the gain in carbon savings with increasing temporal flexibility (higher desired completion time T). We run a 12hrs ML training job (ResNet18) and configure it to complete in 12hrs ($T = l$) up to 36hrs ($T = 3 \times l$). For higher completion times, more low carbon slots become available, which allows CarbonScaler and suspend-resume to reduce the carbon consumption by 30-45% and 0-32%, respectively. CarbonScaler achieves higher savings by using a higher scale factor during the lowest carbon slots and only picks a higher carbon slot if it gives a better marginal work done per unit carbon. For very high completion times, the savings of CarbonScaler over suspend-resume diminish, since it begins to prefer job suspensions over high scale factors to avoid the impact of non-linear scaling behavior.

Effect of Job Length. The length of a job is another key factor in determining carbon savings. As the job length increases, more low-carbon slots become available as the grid's carbon intensity generally has a diurnal pattern. To evaluate the impact of job length, we varied the job length from 6 hours to 96 hours and used our Carbon Advisor to analyze the estimated carbon savings. Figure 14 shows the carbon savings of different policies, against a carbon-agnostic baseline, for the N-body(N=100k) MPI workload when $T = 1.5 \times l$. CarbonScaler outperforms suspend-resume and carbon-agnostic over various job lengths. The carbon savings increase with job length since there are more low-carbon time slots to choose from, providing opportunities for greater savings. Overall, CarbonScaler achieves 30% more savings than suspend-resume for long batch jobs.

Effect of Cluster Size. Our experiments thus far have used a lower bound of 1 server ($m = 1$) and an upper bound of 8 servers ($M = 8$) for workloads due to cluster size and cloud cost constraints.



(a) Workloads (Fig. 9) (b) Deadlines (Fig. 13) (c) Carbon-Cost tradeoff

Fig. 16. *Monetary cost overhead of CarbonScaler over carbon-agnostic execution for different scenarios.*

However, larger batch jobs execute on larger clusters, with larger m and M. For example, certain HPC and ML training applications run on tens or even hundreds of servers in the cloud [18, 35] and can only be executed on a large number of servers $m \gg 1$. To evaluate the efficacy of `CarbonScaler` for large clusters, we extrapolated the marginal capacity curve for the current N-body($N = 100k$) job. Then, we use `Carbon Advisor` to estimate how carbon savings change when running progressively bigger jobs on increasing cluster sizes while keeping the job length unchanged at 24hrs.

Figure 15(a) compares the carbon consumption across cluster sizes. The figure shows that, although the savings percentages diminish with larger cluster sizes as they are less dynamic, the absolute carbon savings increase. Figure 15(b) depicts the relation between policies aside from the size-dependent carbon consumption. As shown, `CarbonScaler` can obtain 30–42% additional savings than `carbon-agnostic`, and `suspend-resume` achieves the same savings of 17% over `carbon-agnostic` policy across all cluster sizes. The `suspend-resume` achieves this static saving since it suspends the job in the same high carbon periods regardless of the cluster size. Lastly, the figure shows that the savings difference between `CarbonScaler` and `suspend-resume` reduces as the cluster size increases since the marginal capacity curve shows diminishing gains for larger cluster sizes.

Key Takeaway. *CarbonScaler exploits temporal flexibility to outperform suspend-resume policy across regions with different carbon costs and over different job lengths, completion times, and cluster sizes.*

5.5 Monetary Cost Overhead

As discussed in Section 3.4, for the workloads with diminishing marginal capacity curves, `CarbonScaler` can potentially incur extra cloud costs quantified as the additional cloud compute-hours needed compared to the `carbon-agnostic` policy. In Figure 16, we present the effect of workload scalability, extended completion time, and degree of flexibility on the added cost of `CarbonScaler`. Figure 16(a) shows that the highly scalable workloads such as N-body ($N = 100$) and ML (ResNet18) that yield the highest savings under `CarbonScaler` cost only 5-10% higher than a `carbon-agnostic` policy. The less scalable workloads incur higher costs for the same carbon savings. It is worth noting that the `static-scale` would also incur similar overheads as the cost depends on the scaling properties of the workload [29]. For ML (ResNet18) workload, Figure 13 demonstrates that as the job completion time increases, the added cost increases up to 7% and then plateaus with a further increase in completion time. This is because, at higher job completion times, there are more low-carbon slots available where `CarbonScaler` can scale higher. Importantly, across both scenarios, the added cost never increases beyond 18%. Finally, in figure 16 (c), we leverage `Carbon Advisor` to highlight the tradeoff between carbon savings and cost overheads across different degrees of flexibility for ML (ResNet18). The figure shows that there exists a flexibility degree that yields the highest carbon savings of almost 9% per each % of added cost.

Key Takeaway. *The cloud cost overhead of CarbonScaler is small but depends on the scalability properties of the workloads (the higher the scalability, the lower the cost overhead). Furthermore, there may be a sweet spot across various dimensions that yields the highest savings per unit of added cost.*

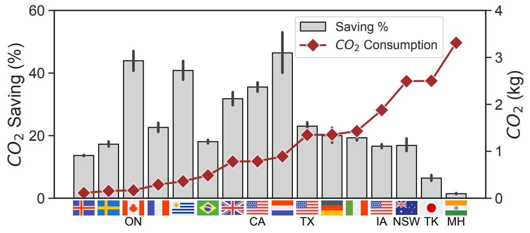


Fig. 17. *Carbon consumption (kg) and savings (%), for an ML (ResNet18) job, where $T = l$, across geographical regions (carbon intensity increases from left to right).*

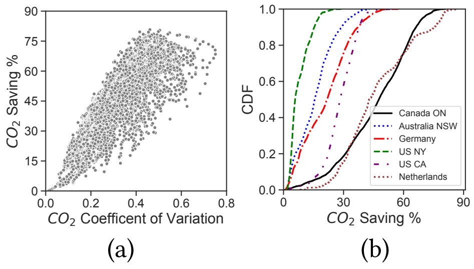


Fig. 18. *Effect of variation (a) and location (b) on* realized *savings for an ML (ResNet18) job ($T = l = 24$hrs).*

5.6 Impact of Carbon Cost Dynamics

Since achievable carbon savings depend on the temporal characteristics of the carbon costs within a cloud region, which significantly vary across regions, we next evaluate the impact of regions and carbon intensity variability on carbon savings.

Carbon Savings Across Cloud Regions. To assess the effect of regions on carbon savings, we use `Carbon Advisor` to compute carbon savings achieved by a 24hrs long ML (ResNet18) job, with $T = l$, across 16 different AWS cloud regions. Figure 17 provides several insights about the average relative and absolute carbon savings compared to the `carbon-agnostic` policy. First, the figure shows that the carbon emissions of the same job can vary by an *order of magnitude* depending on which cloud region is used to execute it. Second, `CarbonScaler` is able to achieve significant carbon savings (in %) for most regions, with a median and average savings of 16% and 19%, respectively. So long as the carbon costs exhibit diurnal variations, `CarbonScaler` can reduce the job's emissions over the `carbon-agnostic` policy regardless of whether it runs in a low or high carbon region. Finally, Figure 17 shows that India's (🇮🇳) region is an exception: while it has high absolute carbon costs, its low hourly variations prevent `CarbonScaler` from generating much savings.

Effect of Variability. As noted earlier, regions with variable carbon cost tend to generate higher carbon savings. This is because the high variations in such regions provide more low carbon periods to exploit for carbon reductions. We use the coefficient of variation, standard deviation divided by mean, as a metric to quantify the variability of the region. Figure 18(a) shows the carbon savings, for each starting point of the year, for a 24hrs ML (ResNet18) job with no excess time for Ontario, Canada using `Carbon Advisor`. The carbon savings are highly correlated with the coefficient of variation, with a Pearson coefficient of 0.82. However, even a highly variable location like Ontario has a small fraction of hours when savings are less than 20%, a fraction that will vary depending on the region. Figure 18(b) presents the distribution of carbon savings and compare regions with different average coefficient of variation. Note that the curves on the right are better as they lead to high carbon savings most of the time. The regions represented by the curves are also strictly ordered by their coefficient of variation, which means that a coefficient of variation can be used to rank regions, when mean carbon cost is comparable, for their carbon saving potential.

Key Takeaway. *CarbonScaler achieves carbon savings for most cloud regions regardless of their absolute carbon cost. In addition, higher diurnal variations in carbon cost translate to greater savings.*

5.7 Robustness to Errors

In prior experiments, we assumed that the carbon forecasts are perfect and applications are profiled on an environment similar to what they eventually run on, yielding highly accurate marginal capacity curves. However, in practice, these assumptions may not always be true, and we evaluate the impact of deviation from the ground-truth for these two factors.

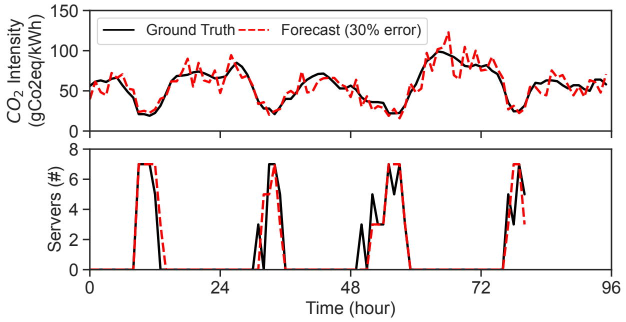


Fig. 19. *Illustrative example of error in carbon forecasts for an N-Body (N = 100k) workload.*

Effect of Carbon Forecast Error. Carbon forecasts are easily available through online tools and services such as [43, 44, 71], with a reported mean accuracy of 6.4%. More importantly, the fidelity of `CarbonScaler` does not depend on the actual magnitude of the carbon forecast and instead relies on correctly identifying the hills (high carbon slots) and valleys (low carbon slots) in the carbon trace, which can be predicted with high accuracy. To illustrate this effect, we generate carbon traces with forecast errors of up to 30% by adding a uniformly random error in the range of -X% to X% for an error of X%. Figure 19(top) shows an example ground-truth and forecasted (X = 30% error) carbon cost time-series. While an erroneous forecast deviates from ground-truth at certain points, it still retains the hills and valleys, leading to harmonious schedules in both cases.

To further quantify the effect of forecast errors, we compare the performance of `CarbonScaler` with perfect carbon forecast to an error-agnostic variant of `CarbonScaler` that is oblivious to forecast errors, and `CarbonScaler` that recompute the schedule when the realized forecast error exceeds 5%. Figure 20 shows the carbon overhead over the perfect forecast scenario. The results highlight the resiliency of `CarbonScaler` to forecast errors, as a 30% forecast error resulted in merely 4% added carbon at 95^{th} percentile.

Effect of Profiling Errors. The marginal capacity curves generated by the `Carbon Profiler` can become erroneous if the environment characteristics, such as network bottlenecks [36], change during the execution. This can impact the carbon savings of a given job if scaling behavior changes due to deviation from actual marginal capacity curves. To evaluate the effect of erroneous profiles, we added uniformly random errors to the marginal capacity curves and measured the carbon consumption using `Carbon Advisor`. Figure 21 shows the carbon overhead over `CarbonScaler` with accurate marginal capacity profiles. The results show that the magnitude of error depends on the application power consumption and scalability behavior, e.g., the N-body job is less affected by errors as it has low power consumption and scales somewhat linearly. Additionally, we only show the results for the initial phase of execution, where errors persist. `CarbonScaler`'s error-handling mechanism of updating marginal capacity curves, when they deviate, corrects the errors, and net overhead over the entire execution of the workload would be considerably small.

Impact of Server Procurement Denial. Since `CarbonScaler` dynamically scales each job independently, similar to cloud autoscalers, many jobs may request cloud servers during low carbon periods, creating a high demand for servers during such periods. Thus, jobs may end up competing with one another for additional servers, which can cause the cloud platform to deny some requests for new instances, to avoid failures. For example, it is not uncommon to see denials for popular GPU instances during work hours, even in the absence of carbon scaling. To evaluate the effect of such denials, we run a 24hr job with 48hr completion time, ($T = 2 \times l$), with different probabilities of random procurement denials. In such cases `CarbonScaler` keeps retrying its request and then recomputes the schedule to mitigate the impact of denials on job completion. Figure 22 illustrates

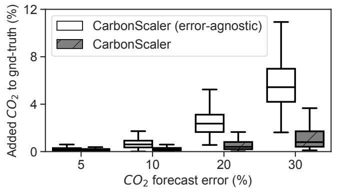


Fig. 20. *Effect of carbon forecast errors for an N-body (N=100k) job.*

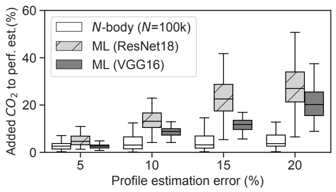


Fig. 21. *Effect of errors in profiled marginal capacity curves.*

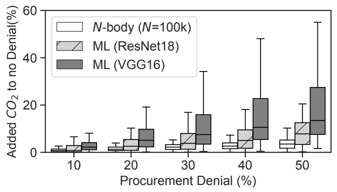


Fig. 22. *Carbon overhead of the server procurement denial.*

that the carbon overhead, compared to a no-denial scenario, increases as the denial percentage increases. The overhead's magnitude depends on a job's scalability behavior. For example, a highly scalable N-body job incurs 5% overhead, while a non-scalable ML job (VGG16) incurs up to 15% overhead compared to the best schedule.

Key Takeaway. *CarbonScaler only depends on carbon cost trends, and simple recomputations achieve savings comparable to the perfect estimation. The potential overheads of profiling error can be overcome by updating marginal capacity curves as they start to deviate. Finally, resource availability can impact the achievable savings, but the magnitude depends on the scalability properties of the workloads.*

5.8 System Overheads

CarbonScaler incurs two types of systemic overheads in its execution. First, CarbonScaler incurs switching overhead, which is the overhead of scaling or suspending, as the number of resources changes over time. The scaling overhead is a function of the application state size (e.g., the number of parameters in ML models). Although CarbonScaler did not account for this overhead in its scheduling decisions, in our experiments, the scaling overhead was between 20-40 seconds. We note that suspend-resume incurs similar overheads as the state is scale-independent. The second source of overhead is the time needed by Carbon Profiler to obtain marginal capacity curves. As mentioned in §4.1, profiling time can be configured using profile duration α at each allocation level, and granularity β of allocations profiled. We use $\alpha = 1$ minute, and $\beta = 1$, i.e., we profile across all possible allocation levels. Thus, the one-time profiling took 40 minutes, where each workload in Figure 9 took only 8 minutes.

Key Takeaway. *CarbonScaler's systemic overheads are small, configurable, and generally occur once.*

6 DISCUSSION

CarbonScaler takes an application-centric approach to reduce the carbon footprint of cloud workloads. While addressing potential second-order effects is outside the scope of CarbonScaler, we discuss the implications for cloud operators when customers operate in a carbon-aware manner.

Capacity Constraints. Cloud operators have different optimization goals and constraints than their tenants. The conflicts are handled through the *pay-as-you-go* pricing model, which hides the underlying constraints, objectives, and potential second-order effects from customers. Additionally, datacenters are designed for peak demand to handle workloads that exhibit diurnal patterns, where they increase at certain times of the day and are correlated between customers. As a result, they typically have low utilization, usually between 40-60% [7, 8, 62], providing enough headroom to handle peaks from carbon-aware demand shifting. Carbon savings are achieved by aligning the demand with the carbon intensity. However, as more and more customers try to increase their carbon efficiency, the compute and power demand will increase at certain periods beyond the datacenter capacity. This will require cloud operators to handle such spikes by adopting a dynamic pricing model, enforce fair sharing limits, or by denying resource acquisition requests if needed. The modeling of such dynamic pricing and carbon-aware fair shares and how CarbonScaler will respond is outside the scope of this paper. For acquisition denials, we demonstrate that CarbonScaler is robust to such denials (see Figure 22).

Datacenter Energy Optimizations. CarbonScaler is an application-centric approach to reducing the carbon footprint of executing workloads in the cloud, which can be used by organizations that are setting ambitious goals for reducing the carbon footprint of their operations [12, 33, 67]. While cloud customers' behavior impacts the cloud datacenter operation, the pay-as-you-go model hides that from the customer. Internally, cloud operators can deploy several optimizations, such as forecasting demand and putting servers into a deep sleep or turning them off completely, offering resources at a discounted price, and procuring location-specific renewable energy. Many cloud operators are already experimenting with such optimizations. Examples include variable capacity computing at Google [59, 75], spot VMs offered by AWS [20], and 24/7 renewable energy procurements by all the major cloud providers [21, 28]. Considering the impact of such operator-side optimizations is outside the scope of a customer-oriented approach like CarbonScaler.

Holistic Emissions Reduction. A datacenter's carbon emissions arise from manufacturing hardware like servers (embodied emissions) and operating these resources (operational emissions). While both emission types are important, they require distinct optimizations [9]. For instance, cutting embodied emissions involves extending device lifespan and choosing low operational carbon suppliers [2, 42]. How cloud operators and customers leverage such techniques for optimizing embodied carbon is outside the scope of this paper. Instead, in this paper, we focus on reducing operational carbon emissions by modulating how and when we execute our workloads.

7 RELATED WORK

Batch Scheduler. HPC schedulers have focused on achieving high utilization and performance efficiency. Traditional batch schedulers such as Slurm [74] and Torque [66] focus on fixed-sized clusters and employed multiple policies to optimize turnaround [16], utilization [60], and energy[27]. Recent schedulers such as Borg, Kubernetes, and Mesos [32, 41] have utilized the elasticity of cloud resources while considering the monetary cost. In both cases, sustainability concerns have influenced operational and scheduling decisions, leading to optimization objectives such as reducing carbon consumption. In the rest of this section, we discuss recent research on carbon-aware scheduling.

Energy Accounting. Reporting carbon consumption depends on a cluster's ability to account for an individual tenant's energy consumption. CarbonScaler currently focuses on CPU and GPU resources as they are 1) highly correlated with total energy consumption [39], 2) software tools such as (RAPL) [17] and `nvidia-smi` [53], are available on modern processors and GPUs. However, our accounting methods can be generalized to other server resources as shown in [11, 15, 23, 39]. Such accounting techniques are vital for holistic carbon optimization since cloud service providers such as Microsoft [50], and AWS [49] are starting to offer basic carbon management capabilities.

Temporal Shifting. Temporal shifting by delaying execution of batch jobs from high carbon slots to lower carbon slots has been explored in [19, 59, 73]. The Let's wait-a-while [73] approach uses temporal shifting to reduce the carbon footprint of batch workloads using threshold and deadline-based methods and by exploiting overnight or weekend hours to extract savings. In [19], the authors highlight the implications of scheduling AI workloads in different settings and suggest temporal shifting to minimize the carbon footprint. Finally, in [59], the authors employ a virtual limit on resources when carbon cost is high to force the scheduler to shift workloads to lower carbon periods. As noted in §1, a limitation of temporal shifting approaches is that they delay job completion times and may also require users to specify deadlines for jobs. In contrast, CarbonScaler employs resource elasticity to scale and complete jobs in a timely manner and can additionally exploit temporal flexibility whenever available.

Spatial Shifting. Prior work has studied spatial shifting to select the region with the lowest carbon footprint to execute newly arriving jobs. The authors of [2, 19, 68, 76] explore the spatial selection to achieve lower carbon cost. The authors of [2] explore data center and power upgrade plans to allow more carbon-efficient execution, while [19, 68] explore cloud data center regions and potential carbon savings. Lastly, [76] exploits migration to avoid energy curtailment. While we study the benefits of using different geographic regions to run carbon scaling jobs in §5.6, a full analysis of combining spatial shifting with carbon scaling is outside the scope of this paper.

8 CONCLUSION

Many compute-intensive cloud workloads, such as ML training and scientific computations, have inherent resource elasticity and temporal flexibility that can be leveraged to optimize carbon emission reductions. To exploit this opportunity, we propose CarbonScaler that judiciously scales up or down an application, based on its scalability behavior and carbon cost, to minimize its carbon emissions. We implement CarbonScaler as a cloud-based autoscaler implemented using Kubernetes and a simulation-based advisory tool to facilitate pre-deployment analysis. We demonstrate the efficacy of CarbonScaler in reducing carbon emissions for various workloads, job configurations, and cloud regions. We demonstrated that using real-world machine learning training and MPI jobs on a commercial cloud platform, CarbonScaler can yield i) 51% carbon savings over carbon-agnostic execution, ii) 37% over a suspend-resume policy, and iii) 8% over the best static scaling policy. In the future, we plan to extend CarbonScaler into a cluster-wide scheduler to address the challenges of resource heterogeneity, resource pressure, priorities, and power management.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Anshul Gandhi, for their valuable comments, which improved the quality of this paper. This research is supported by NSF grants 2211302, 2211888, 2213636, 2105494, US Army contract W911NF-17-2-0196, VMware, and Amazon Web Services.

REFERENCES

[1] J. Sverre Aarseth. 1985. 12 - Direct Methods for N-Body Simulations. In *Multiple Time Scales*. Academic Press, 377–418. https://doi.org/10.1016/B978-0-12-123420-1.50017-3

[2] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. 2023. Carbon Explorer: A Holistic Framework for Designing Carbon Aware Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 118–132. https://doi.org/10.1145/3575693.3575754

[3] Gene M Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the Spring Joint Computer Conference.*

[4] Anders S. G. Andrae and Tomas Edler. 2015. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges* 6, 1 (2015), 117–157. https://doi.org/10.3390/challe6010117

[5] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. Scaling Spark in the Real World: Performance and Usability. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1840–1843. https://doi.org/10.14778/2824032.2824080

[6] AWS. 2022. AWS Auto Scaling. https://aws.amazon.com/autoscaling/.

[7] Luiz André Barroso and Urs Hölzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Springer Nature, Europe. 189 pages.

[8] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the Limit: Peak Prediction-driven Resource Overcommitment in Datacenters. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 556–573. https://doi.org/10.1145/3447786.3456259

[9] Noman Bashir, David Irwin, and Prashant Shenoy. 2023. On the Promise and Pitfalls of Optimizing Embodied Carbon. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems (HotCarbon)*. ACM, New York, NY, USA, 6 pages.

[10] Noman Bashir, David Irwin, Prashant Shenoy, and Abel Souza. 2022. Sustainable Computing – Without the Hot Air. In *HotCarbon: Workshop on Sustainable Computer Systems Design and Implementation*. ACM, New York, NY, USA, 7 pages.

[11] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. Powerapi: A Software Library to Monitor the Energy Consumed at the Process-level. *ERCIM News* (2013).

[12] Seán Boyle and Casey Junod. 2023. Accelerating our climate commitments on Earth Day. https://blog.twitter.com/en_us/topics/company/2022/accelerating-our-climate-commitments-on-earth-day.

[13] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. Neuralpower: Predict and Deploy Energy-efficient Convolutional Neural Networks. In *Asian Conference on Machine Learning*.

[14] A. Chien. 2021. Driving the Cloud to True Zero Carbon. *Communication of the ACM* 64, 2 (February 2021).

[15] Maxime Colmant, Mascha Kurpicz, Pascal Felber, Loïc Huertas, Romain Rouvoy, and Anita Sobe. 2015. Process-Level Power Estimation in VM-Based Systems. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2741948.2741971

[16] Renato L.F. Cunha, Eduardo R. Rodrigues, Leonardo P. Tizzei, and Marco A.S. Netto. 2017. Job placement advisor based on turnaround predictions for HPC hybrid clouds. *Future Generation Computer Systems* 67 (2017), 35–46. https://doi.org/10.1016/j.future.2016.08.010

[17] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[19] Jesse Dodge, Taylor Prewitt, Remi Tachet des Combes, Erika Odmark, Roy Schwartz, Emma Strubell, Alexandra Sasha Luccioni, Noah A. Smith, Nicole DeCario, and Will Buchanan. 2022. Measuring the Carbon Intensity of AI in Cloud Instances. In *2022 ACM Conference on Fairness, Accountability, and Transparency (FAccT '22)*.

[20] EC2 2022. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[21] EPA. 2023. Green Power Partnership Long-term Contracts. https://www.epa.gov/greenpower/green-power-partnership-long-term-contracts

[22] Awi Federgruen and Henri Groenevelt. 1986. The Greedy Procedure for Resource Allocation Problems: Necessary and Sufficient Conditions for Optimality. *Oper. Res.* 34, 6 (dec 1986), 909–918.

[23] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. 2020. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 479–488. https://doi.org/10.1109/CCGrid49817.2020.00-45

[24] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. USA.

[25] William Fox, Devarshi Ghoshal, Abel Souza, Gonzalo P. Rodrigo, and Lavanya Ramakrishnan. 2017. E-HPC: A Library for Elastic Resource Management in HPC Environments. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science* (Denver, Colorado) *(WORKS '17)*. Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3150994.3150996

[26] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (nov 2012), 26 pages. https://doi.org/10.1145/2382553.2382556

[27] Saurabh Kumar Garg, Chee Shin Yeo, Arun Anandasivam, and Rajkumar Buyya. 2011. Environment-conscious scheduling of HPC applications on distributed cloud-oriented data centers. *J. Parallel and Distrib. Comput.* 71, 6 (2011), 732–749.

[28] Google. 2022. Google's Green PPAs: What, How, and Why. https://static.googleusercontent.com/media/www.google.com/en//green/pdfs/renewable-energy.pdf.

[29] Walid A Hanafy, Roozbeh Bostandoost, Noman Bashir, David Irwin, Mohammad Hajiesmaili, and Prashant Shenoy. 2023. The War of the Efficiencies: Understanding the Tension between Carbon and Energy Optimization. In *Proc. 2nd ACM Workshop on Hot Topics in Sustainable Computing Systems (HotCarbon'23)*.

[30] Fiona Harvey. 2021. The Guardian, Major Climate Changes Inevitable and Irreversible – IPCC's Starkest Warning Yet. https://www.theguardian.com/science/2021/aug/09/humans-have-caused-unprecedented-and-/irreversible-change-to-climate-scientists-warn.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[32] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Boston, MA, 14. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center
[33] VMware Inc. 2023. Journey to Net Zero. https://www.vmware.com/company/net-zero.html.
[34] World Resource Institute. 2022. *GreenHouseGas Protocol.* https://ghgprotocol.org/
[35] Sam Adé Jacobs, Nikoli Dryden, Roger Pearce, and Brian Van Essen. 2017. Towards Scalable Parallel Training of Deep Neural Networks. In *Proceedings of the Machine Learning on HPC Environments (MLHPC'17).*
[36] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 2015. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 407–420. https://doi.org/10.1145/2785956.2787488
[37] Nicola Jones. 2018. How to Stop Data Centres from Gobbling Up the World's Electricity. *Nature* (2018).
[38] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the Computational Cost of Deep Learning Models. In *2018 IEEE International Conference on Big Data (Big Data).*
[39] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. 2010. Virtual Machine Power Metering and Provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/1807128.1807136
[40] Kubeflow. 2022. Kubeflow: The Machine Learning Toolkit for Kubernetes. https://www.kubeflow.org/. Accessed: 2022-10-03.
[41] Kubernetes. 2022. Kubernetes: Production-grade Container Orchestration. https://kubernetes.io/. Accessed: 2022-10-03.
[42] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2023. Sustainable HPC: Modeling, Characterization, and Implications of Carbon Footprint in Modern HPC Systems. arXiv:2306.13177 [cs.DC]
[43] Diptyaroop Maji, Prashant Shenoy, and Ramesh K. Sitaraman. 2022. CarbonCast: Multi-Day Forecasting of Grid Carbon Intensity. In *Proceedings of the 9th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (Boston, Massachusetts) *(BuildSys '22)*. Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/3563357.3564079
[44] Diptyaroop Maji, Ramesh K. Sitaraman, and Prashant Shenoy. 2022. DACF: Day-Ahead Carbon Intensity Forecasting of Power Grids Using Machine Learning. In *Proceedings of the Thirteenth ACM International Conference on Future Energy Systems (e-Energy'22).*
[45] Electricity Maps. 2022. Electricity Map. https://www.electricitymap.org/map.
[46] Eric R. Masanet, Arman Shehabi, Nuoa Lei, Sarah J. Smith, and Jonathan G. Koomey. 2020. Recalibrating Global Data Center Energy-use Estimates. *Science* (2020).
[47] Valérie Masson-Delmotte, Panmao Zhai, Anna Pirani, Sarah L Connors, Clotilde Péan, Sophie Berger, Nada Caud, Yang Chen, Leah Goldfarb, Melissa I Gomis, et al. 2021. *Summary for Policymakers. In: Climate Change 2021: The Physical Science Basis. Contribution of Working Group I to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change.* Technical Report. United Nation Intergovernmental Panel on Climate Change (IPCC).
[48] META. 2022. How We're Helping Fight Climate Change. https://about.fb.com/news/2021/06/2020-sustainability-report-how-were-helping-fight-climate-change/.
[49] Microsoft. 2022. AWS Customer Carbon Footprint Tool. https://aws.amazon.com/blogs/aws/new-customer-carbon-footprint-tool/.
[50] Microsoft. 2022. Microsoft Carbon accouting tool. https://www.microsoft.com/en-us/sustainability/emissions-impact-dashboard.
[51] Microsoft. 2022. Microsoft is Changing the Way It Buys Renewable Energy. https://www.theverge.com/2021/7/14/22574431/microsoft-renewable-energy-purchases.
[52] Fereydoun Farrahi Moghaddam, Reza Farrahi Moghaddam, and Mohamed Cheriet. 2014. Carbon-aware Distributed Cloud: Multi-level Grouping Genetic Algorithm. *Cluster Computing* (2014).
[53] NVIDIA. 2022. Manage and Monitor GPUs in Cluster Environments. https://developer.nvidia.com/dcgm. Accessed: 2022-10-08.
[54] Yosuke Oyama, Akihiro Nomura, Ikuro Sato, Hiroki Nishimura, Yukimasa Tamatsu, and Satoshi Matsuoka. 2016. Predicting Statistics of Asynchronous SGD Parameters for a Large-scale Distributed Deep Learning System on GPU Supercomputers. In *2016 IEEE International Conference on Big Data (Big Data).*
[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NIPS'19).*

[56] Ziqian Pei, Chensheng Li, Xiaowei Qin, Xiaohui Chen, and Guo Wei. 2019. Iteration Time Prediction for CNN in Multi-GPU Platform: Modeling and Analysis. *IEEE Access* (2019).

[57] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/3190508.3190517

[58] Qi, Evan R. Sparks, and Ameet S. Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *The International Conference on Learning Representations (ICLR'17)*.

[59] Ana Radovanovic, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, Maya Haridasan, Patrick Hung, Nick Care, Saurav Talukdar, Eric Mullen, Kendal Smith, Mariellen Cottman, and Walfredo Cirne. 2022. Carbon-Aware Computing for Datacenters. *IEEE Transactions on Power Systems* (2022), 1–1. https://doi.org/10.1109/TPWRS.2022.3173250

[60] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. 2018. Scalable system scheduling for HPC and big data. *J. Parallel and Distrib. Comput.* 111 (2018), 76–92. https://doi.org/10.1016/j.jpdc.2017.06.009

[61] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[62] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. United States Data Center Energy Usage Report. (6 2016). https://doi.org/10.2172/1372902

[63] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. 2018. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. 949–957. https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.000-4

[64] Kubernetes SIGs. 2022. *Kubernetes Metrics Server*. Kubernetes SIGs. https://github.com/kubernetes-sigs/metrics-server

[65] Abel Souza, Noman Bashir, Jorge Murillo, Walid Hanafy, Qianlin Liang, David Irwin, and Prashant Shenoy. 2023. Ecovisor: A Virtual Energy System for Carbon-Efficient Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3575693.3575709

[66] Garrick Staples. 2006. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, New York, NY, USA, 8.

[67] Emma Stewart. 2023. Net Zero + Nature: Our Commitment to the Environment. https://about.netflix.com/en/news/net-zero-nature-our-climate-commitment.

[68] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David Irwin, and Prashant Shenoy. 2023. Quantifying the Benefits of Carbon-Aware Temporal and Spatial Workload Shifting in the Cloud. arXiv:2306.06502 [cs.DC]

[69] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 6105–6114. https://proceedings.mlr.press/v97/tan19a.html

[70] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[71] WattTime. 2022. WattTime. https://www.watttime.org/.

[72] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 945–960.

[73] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. 2021. Let's Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 260–272. https://doi.org/10.1145/3464298.3493399

[74] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, New York, NY, USA, 44–60.

[75] Chaojie Zhang and Andrew A. Chien. 2021. Scheduling Challenges for Variable Capacity Resources. In *Job Scheduling Strategies for Parallel Processing*, Dalibor Klusáček, Walfredo Cirne, and Gonzalo P. Rodrigo (Eds.). Springer International Publishing, Cham, 190–209.

[76] Jiajia Zheng, Andrew A. Chien, and Sangwon Suh. 2020. Mitigating Curtailment and Carbon Emissions through Load Migration between Data Centers. *Joule* 4, 10 (2020), 2208–2222. https://doi.org/10.1016/j.joule.2020.08.001

[77] Zhi Zhou, Fangming Liu, Yong Xu, Ruolan Zou, Hong Xu, John C.S. Lui, and Hai Jin. 2013. Carbon-Aware Load Balancing for Geo-distributed Cloud Services. In *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems.* IEEE, New York, NY, USA, 232–241. https://doi.org/10.1109/MASCOTS.2013.31

A CARBONSCALER OPTIMALITY

The carbon scaling problem addressed by `CarbonScaler` is a marginal resource allocation problem, where greedily selecting the local optimum (maximum marginal capacity per unit carbon), as in Algorithm 1, yields the global optimum solution [22]. Consequently, the optimality of the `Carbon Scaling Algorithm` follows from the theoretical results of [22] and is shown below.

Theorem 1. *Consider a distributed batch job with a known monotonically decreasing marginal capacity curve, s.t. $MC_m > MC_{m+1} > .. > MC_M$. The job needs to finish work W, within n time slots with known carbon costs $c_1, c_2, ..., c_n$, respectively. Greedily selecting the slot i and scaling the job to j servers with the highest marginal capacity per unit carbon MC_j/c_i[5], in each step, results in the lowest (optimal) amount of carbon consumption.*

Proof. We prove Theorem 1 by contradiction. Let S be an optimal solution schedule that finishes work W and has a carbon cost C_S. The schedule S is constructed by allocating time slots and number of servers, until W is completed. The tuple (i, j) denotes the i-th time slot and the j-th server allocated to the job. MC_j is the marginal work done when allocating the j-th server, and c_i is the carbon cost used per server at time slot i, where we assume perfect knowledge of both. The total carbon cost is $C_S = \sum_{i \in n} c_i \times S[i]$, where $S[i]$ is the used number of servers at time slot i.

The tuple (k, l) denotes the k-th time slot and the l-th server, with marginal capacity per unit carbon of MC_l/c_k. Assume that there exists a time slot i and a number of servers j, where $MC_l/c_k > MC_j/c_i$, s.t., $(i, j) \in S$ and $(k, l) \notin S$. We denote S' as a new schedule, where we only switch the i-th time slot and j-th server with k-th time slot and the l-th server, which has the higher marginal capacity per unit carbon. To ensure that the schedule S' finishes work W, the amount of work MC_j must be incorporated into the new schedule. We denote c_i and γ, as the old carbon and new carbon costs to perform work MC_j, respectively. γ is computed based on the relationship between l and j, where:

$$\gamma = \begin{cases} c_k \cdot \frac{MC_j}{MC_l}, & \text{if } l \leq j. \\ c_k + (\frac{MC_j - MC_l}{MC_j}) \cdot c_i, & \text{otherwise } (l > j). \end{cases} \tag{1}$$

In the first case $(l \leq j)$, $MC_l \geq MC_j$ and job will use part or all of the time slot k. In the second case $(l > j)$, $MC_l < MC_j$. We perform MC_l work in time slot k, and run the overflow work, $MC_j - MC_l$, in time slot i, utilizing $\frac{MC_j - MC_l}{MC_j}$ of time slot i and all of time slot k.

Next, to show that $c_i > \gamma$ and $(C_S > C_{S'})$, we consider both cases. In the first case, since $MC_l/c_k > MC_j/c_i$, then:

$$c_i > c_k \cdot \frac{MC_j}{MC_l} \tag{2}$$

$$c_i > \gamma \tag{3}$$

In the second case, since $\frac{MC_l}{c_k} > \frac{MC_j}{c_i}$, then:

$$c_k < \frac{MC_l}{MC_j} \cdot c_i \tag{4}$$

[5]We assume that switching cost (scaling up or down) between time slots is negligible.

By substituting c_k in case 2:

$$\frac{MC_l}{MC_j} \cdot c_i + (\frac{MC_j - MC_l}{MC_j}) \cdot c_i > \gamma \tag{5}$$

$$\frac{MC_l \cdot c_i + MC_j \cdot c_i - MC_l \cdot c_i}{MC_j} > \gamma \tag{6}$$

$$c_i > \gamma \tag{7}$$

Therefore, carbon consumption of $S^{'}$, denoted as $C_{S'} = C_S - c_i + \gamma$ is less than carbon cost of S (C_S), since $c_i > \gamma$. Hence, S is not optimal, a contradiction. □

Received February 2023; revised October 2023; accepted October 2023