Going Green for Less Green: Optimizing the Cost of Reducing Cloud Carbon Emissions

Walid A. Hanafy
University of Massachusetts Amherst
USA

Qianlin Liang
University of Massachusetts Amherst
USA

Noman Bashir
Massachusetts Institute of Technology
USA

Abel Souza
University of Massachusetts Amherst
USA

David Irwin
University of Massachusetts Amherst
USA

Prashant Shenoy
University of Massachusetts Amherst
USA

Abstract

The continued exponential growth of cloud datacenter capacity has increased awareness of the carbon emissions when executing large compute-intensive workloads. To reduce carbon emissions, cloud users often temporally shift their batch workloads to periods with low carbon intensity. While such time shifting can increase job completion times due to their delayed execution, the cost savings from cloud purchase options, such as reserved instances, also decrease when users operate in a carbon-aware manner. This happens because carbon-aware adjustments change the demand pattern by periodically leaving resources idle, which creates a trade-off between carbon emissions and cost. In this paper, we present GAIA, a carbon-aware scheduler that enables users to address the three-way trade-off between carbon, performance, and cost in cloud-based batch schedulers. Our results quantify the carbon-performance-cost trade-off in cloud platforms and show that compared to existing carbon-aware scheduling policies, our proposed policies can double the amount of carbon savings per percentage increase in cost, while decreasing the performance overhead by 26%.

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

Keywords: Sustainable Computing, Cloud Computing

ACM Reference Format:
Walid A. Hanafy, Qianlin Liang, Noman Bashir, Abel Souza, David Irwin, and Prashant Shenoy. 2024. Going Green for Less Green: Optimizing the Cost of Reducing Cloud Carbon Emissions. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651374

1 Introduction

The increasing demand for computing and accelerating growth in cloud datacenter capacity have long raised concerns about their sustainability, environmental impact, and resulting carbon footprint [23]. Although cloud operators previously addressed such concerns by increasing datacenters' energy efficiency, or work done per unit of energy consumed, via software and hardware optimizations, recent work highlights that continuing improvements to energy efficiency are likely to see diminishing returns [12]. For example, large datacenters already operate near the optimal PUE value of 1.0, so there is little room to further improve PUE. Thus, to continue reducing their carbon emissions, cloud providers are increasingly employing *supply-side* optimizations, such as purchasing renewable energy from solar and wind farms to offset datacenter demand [22, 41]. However, eliminating all carbon emissions using supply-side techniques alone can be very expensive [6, 15].

Researchers have also proposed *demand-side* techniques to decrease computing's operational carbon emissions. These techniques utilize 1) visibility into grid energy's carbon intensity (in g·CO_2eq/kWh) and 2) application-level flexibility to *modulate* demand based on variations in energy's carbon intensity [6, 21, 31, 36, 44]. For example, prior work on Wait Awhile [44] and Ecovisor [35] utilize batch workloads' *temporal flexibility* to optimize carbon by executing jobs when energy's carbon intensity is low and pausing execution when carbon intensity is high. Importantly, while state-of-the-art techniques consider carbon-performance trade-offs, they ignore the cost implications of carbon-aware optimizations. Specifically, carbon-aware scheduling exploits batch jobs' temporal flexibility to delay their execution until energy's carbon intensity is low. However, this delay also increases the completion times of batch jobs. Thus, carbon-aware scheduling exhibits carbon-performance trade-off: decreasing carbon emissions generally results in longer completion times.

In addition to the performance trade-off above, there is also a cost trade-off when using temporal shifting to optimize carbon emissions. This cost trade-off manifests in clusters from leaving resources idle when carbon intensity is high. Cloud users encounter this trade-off when purchasing cheaper, long-term reserved instances to reduce their cloud costs [14, 27]. Idling reserved instances during periods of high carbon intensity wastes their computational cycles and increases the effective per-hour price per unit of computation due to the reduced utilization.

While prior work has explored the trade-off between cost and performance from using reserved versus on-demand instances in the cloud [8, 9], carbon-aware scheduling adds a new dimension to the problem. Specifically, a cost-aware scheduler may choose to delay jobs when the cluster is saturated rather than acquire additional instances, which *spreads* demand across time and decreases cost. In contrast, a carbon-aware scheduler typically *concentrates* jobs to low-carbon periods, which can reduce the utilization of fixed reserve capacity during high-carbon periods. The resulting decrease in utilization increases the effective price of reserved instances, potentially reducing the cost savings relative to using on-demand instances. Thus, reducing cloud carbon emissions using a mix of reserved and on-demand instances requires addressing a complex three-way carbon-performance-cost trade-off.

To address the problem, in this paper, we quantify the trade-off between carbon, cost, and performance for a cloud-based batch scheduler. In doing so, we develop GAIA, which enables carbon-aware scheduling of batch jobs on a mix of on-demand, reserved, and spot instances. Our hypothesis is that carbon-aware scheduling policies that also consider performance and cost can reduce carbon emissions for lower cost with higher performance than existing policies. In evaluating our hypothesis, we make the following contributions.

1. **Trade-off Analysis.** We present a quantitative and qualitative analysis of the three-way trade-off between carbon, cost, and performance for cloud-based batch schedulers, and highlight "good" points in the trade-off where significantly improving one metric has little impact on the others.
2. **GAIA Design.** We present the design of GAIA, a cloud-based batch scheduler that employs configurable carbon-aware scheduling policies and supports a range of cloud purchase options, e.g., reserved, on-demand, and spot instances.
3. **Implementation and Evaluation.** We implement a GAIA prototype on AWS ParallelCluster [1], and conduct a large-scale evaluation of its scheduling policies using publicly available production workload traces and carbon intensity profiles from different geographical regions. Our results quantify the carbon-performance-cost trade-off and show that compared to prior carbon-aware policies, GAIA can double the amount of carbon savings per percentage increase in cost, while decreasing the performance overhead by 26%.

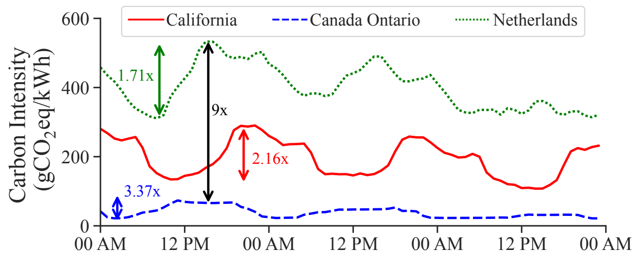


Figure 1. *Grid carbon intensity for three different regions showing 9× spatial and 3.37× temporal variations.*

2 Background

Below, we provide background on grid carbon intensity, carbon-aware scheduling, and cloud-based schedulers.

2.1 Grid Carbon Intensity

The electric grid uses a mix of generation sources to satisfy the energy demand of its customers. Common generation sources include both carbon-intensive generators that burn coal, oil, or natural gas and low-carbon sources, such as hydro, nuclear, wind, and solar. Since the electricity demand varies over time and the availability of renewable energy is intermittent, the mix of generation sources and the relative proportion of electrical energy they generate also varies. Grid energy's carbon intensity (CI) – in grams of CO_2 equivalent per watt (g·CO_2eq/kWh) – reflects the average weighted carbon intensity of the mix of sources used to generate that energy at any given moment. Figure 1 shows the carbon intensity variations of grid energy for three days in three parts of the world. These variations motivate using temporal shifting, as a job can have up to 1.7×-3.37× higher carbon footprint depending on whether it executes during a high or low carbon-intensity period. Moreover, the figure shows that grid energy's carbon intensity varies by up to 9× across regions. While geographically distributed clusters can take advantage of such differences, our current focus is exploiting temporal carbon intensity variations *within* a single region. Spatial batch scheduling across geo-distributed clusters is left for future research.

2.2 Cluster-wide temporal shifting

As noted earlier, temporal flexibility in batch workloads enables workload demand to matched to low carbon periods. Recent efforts have exploited suspend-resume to match the workload with low carbon periods of batch jobs [20, 35, 44]. In such methods, a job specifies a deadline, and the system executes the job in a carbon-efficient manner by running it in time slots where the carbon intensity is low and suspending execution in high carbon periods. Wait Awhile[44] and Ecovisor[35] are recent examples of suspend-resume-based approaches for optimizing carbon usage of individual jobs.

Several temporal shifting methods, such as Wait Awhile [44] and CarbonScaler [21], assume each job specifies a deadline and that individual job lengths are known. While schedulers have access to historical job lengths, knowledge of individual job lengths may not always be available [9, 24]. As a result, batch schedulers such as Slurm [47] are configured such that users submit their jobs to a particular job queue, such as a short- or long-job queue, which provides an upper bound on job length even when the actual job length is not known. Further, users may also not specify a deadline when submitting jobs, and even if they do, a deadline is only meaningful to a system when the job length is known. Finally, state-of-the-art carbon-aware temporal shifting methods focus on reducing carbon for individual jobs and thus ignore other typical cluster-wide objectives. Focusing solely on carbon emissions conflicts with other system-wide objectives, such as cost and energy consumption, and performance objectives, such as completion time, which we will detail in Section 3. Thus, a carbon-aware scheduler must consider these costs and their trade-offs when optimizing the entire cluster's carbon footprint.

2.3 Cloud-based Batch Schedulers

Traditional batch schedulers, such as Slurm [47], GridEngine [19], and Torque [37], focus on fixed-sized clusters and try to optimize for system-wide objectives, such as utilization, cost, or energy consumption. Such schedulers require queued jobs to wait when all servers in the cluster are busy. At the same time, cluster operators carefully provision cluster sizes in a way that tries to *maximize* cluster utilization to ensure cost-efficient operation and *minimize* waiting time to ensure users' productivity and convenience.

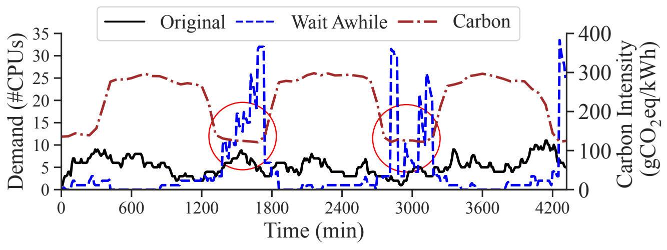
To overcome the limitations of traditional batch systems, cloud-based batch systems, such as AWS ParallelCluster [1] and Kubernetes [13], have been proposed to utilize the elasticity of cloud resources. These schedulers are designed to handle dynamic workloads by varying resource needs over time — by dynamically acquiring and releasing cloud servers to reduce job waiting times while utilizing cloud elasticity to improve cluster utilization and only pay for the necessary resources. To retain the benefits of both fixed-size and elastic clusters, batch systems may also employ a hybrid setting by using a private cloud or long-term *reserved instances.* In this setting, reserved instances present a cheap and fixed resource allocation base that covers average daily demand. However, they require a larger time commitment with 1- or 3-year contracts, and the allocation cost must be paid for the whole contract period, even if the instances are turned off when not in use. This hybrid setting provides a cost-effective alternative to using on-demand instances while retaining the ability to scale when demand exceeds the reserved capacity [14, 27]. Another approach to decrease cost is to utilize spot instances, which are often priced much lower than on-demand instances but may be revoked at any time based on the infrastructure supply and demand [7, 38].

3 The Carbon-Performance-Cost Tension

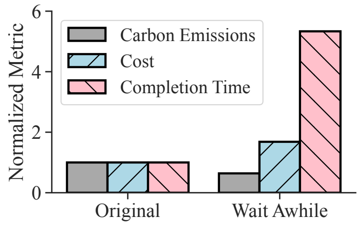
We illustrate the tension between carbon-aware execution and typical operational objectives using a simple example. Our example consists of a synthetic three-day workload trace with an exponentially distributed mean inter-arrival rate of 48 minutes, an exponentially distributed mean job length of 4 hours, and 1 CPU core per job. This yields an average cluster-wide demand of 5 CPUs. We simulate the execution of this workload using a mix of reserved and on-demand instances. We assume that the scheduler has five reserved instances, each with one core. We selected reserved instances equal to the mean demand to achieve low operational cost, as explained in [8]. In this case, the scheduler meets the current demand utilizing the available reserved instances and scales the cluster using on-demand instances when no idle reserved instances are available.

Using our example workload, we calculate the carbon footprint, cost, and completion time of a *carbon-agnostic* FCFS scheduler and compare it to the *carbon-aware* schedule computed using Wait Awhile [44] that executes jobs in a suspend-resume manner. We configure Wait Awhile with a 24-hour maximum waiting time and compute the carbon footprint of each job based on the carbon intensity of the time slots when it executes. We assume that reserved instances are turned off when idle, i.e., they do not consume energy or carbon during idle periods. To determine the cost, we assume a normalized pricing scheme [8] where reserved instances cost 40% of the on-demand price. It is important to note that reserved instances are paid upfront for the whole period, whether utilized or not, while on-demand instances are billed in a pay-as-you-go fashion. Finally, completion time is computed as the sum of the waiting time and the job execution time.

Figure 2a shows the original and the carbon-aware allocation when scheduling the workload based on the February 2022 carbon intensity of the California, US AWS cloud region. As shown, the original demand is stable and able to utilize the reserved instances effectively. In contrast, the carbon-aware demand shows high spikes in low-carbon time slots, where most of the demand is met by on-demand instances, which increases the effective price of reserved instances and reduces their cost savings. The implications of these scheduling decisions are shown in Figure 2b, which compares the performance of Wait Awhile to the original carbon-agnostic schedule. The carbon-aware schedule can achieve a 36% reduction in carbon emissions. However, changes in demand patterns and effective price increase the cost by 68% and completion time by 5.3×. Although some users may be willing to tolerate longer completion times in return for carbon



(a) Resource Demand



(b) Objectives Tension

Figure 2. *The tension between carbon-aware scheduling and typical cost metrics for a three-day workload in US California.*

reductions, the resulting cost increases may be unacceptable for many customers.

While the increase in operational costs and completion times are key trade-offs, as explained earlier, carbon-aware schedulers that do not consider other metrics can be aggressive in their scheduling decisions, irrespective of the actual gains. Figure 2a shows an example of time slots (marked by red circles) with almost identical carbon intensity. As shown, the carbon-aware scheduler focuses on time slots with the lowest carbon intensity, regardless of the actual gains and the performance and cost overheads of such a decision. Although not shown in this figure, we also repeated this experiment in Sweden's AWS region, where carbon intensity is low and stable. In this case, the carbon-aware scheduler resulted in a 76% increase in cost and 4.9 times longer completion time while only providing a 4% reduction in carbon emissions.

4 GAIA Design and Policies

In this section, we outline the design of GAIA, our cloud scheduler for batch processing, and highlight our carbon-, performance-, and cost-aware scheduling policies.

4.1 System Architecture

We design GAIA as a set of modules and services that can be integrated into any existing cloud-enabled batch scheduler. Figure 3 presents the architecture of GAIA, where components we augment for carbon awareness are highlighted in blue. Below, we describe the key elements of our carbon-aware batch scheduler in the cloud.

Job submission. Users submit jobs using an interface that allows them to specify the various aspects of the job, such as the desired queue, resource requirements, and time limits. GAIA maintains the same interface provided by the underlying batch scheduler and does not require any modifications to users' job submission workflow or how monitoring or accounting are done.

Queues. GAIA follows typical scheduling configurations, where queues represent different resource types, job lengths, and user classes. For simplicity, we focus on job length queues, where the queue determines the maximum length the job can run for.

Carbon Information Service (CIS). To enable carbon-aware scheduling, GAIA uses third-party CIS services such as ElectricityMaps [25, 26, 42] that provide real-time per-region carbon intensity information and forecasts.

Resource Manager. The resource manager is responsible for resource allocation and monitoring. It operates on standard cloud offerings such as on-demand, reserved, and spot instances. The resource manager follows the schedule and uses reserved instances when available. If the demand exceeds the available reserved resources (or in on-demand-only clusters), the resource manager allocates and releases on-demand instances based on the number of jobs in the queue. The resource manager can also alternate between spot and on-demand instances based on their availability and expected eviction rates.

Accounting. Today's batch schedulers offer comprehensive accounting capabilities that allow monitoring of a job's resource consumption, execution time, waiting time, and exit codes. For example, Slurm uses a DB Daemon (SlurmDBD) and MySQL to maintain current and historical resource consumption traces. However, in addition to standard metrics, GAIA requires accounting for carbon footprint, cost, and elasticity overheads, which vary based on execution time and the cloud purchase options used for the job. To account for a job's carbon footprint, GAIA combines the job's energy consumption with the carbon intensity of energy from the CIS. GAIA also considers the dynamics of various cloud offerings, such as on-demand, reserved, and spot instances, to account for the cost, carbon, and overhead of spawning cloud instances. Finally, we highlight that the cost and energy consumption of on-demand and spot instances are accounted for based on usage. However, reserved instances' costs are fully paid in advance for the whole allocation time, aside from the actual usage. Nonetheless, reserved instances' energy and carbon consumption can be accounted for based on actual usage.

GAIA Scheduler. At the core of our system lies the GAIA scheduler. It is responsible for scheduling jobs submitted by users based on policies determining when (waiting time) and where (cloud offering) the job should run. The scheduler supports a wide range of policies considering the carbon

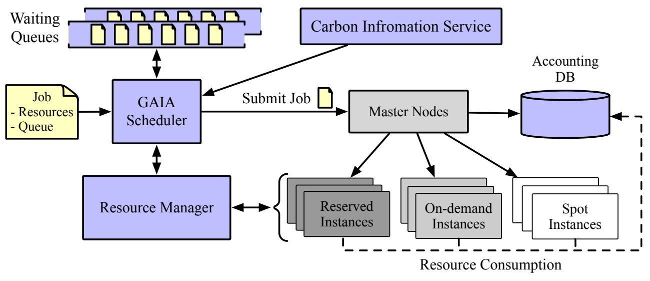


Figure 3. GAIA *architecture and its components.*

footprint, waiting time, and cost, as discussed below. Currently, GAIA is restricted to uninterruptible scheduling of batch workloads, where the scheduler is responsible for picking a start time, and the job is executed until completion. Adding suspend-resume capability to the scheduler is part of future work. Such a capability can further increase carbon savings by suspending jobs during high carbon intensity periods, albeit at the expense of increasing completion times.

4.2 GAIA Scheduling Policies

Our policies assume that users submit their jobs to a job queue. For ease of exposition, we describe the policies assuming two queues — a short and a long queue. However, our policies can be extended to an arbitrary number of queues. These queues provide an upper bound on the job length, where the cluster administrators specify a maximum time J_{short}^{max} and J_{long}^{max} respectively, for which a job in each queue can run before it is terminated. We also assume that the cluster administrator specifies a maximum waiting time, W_{short} and W_{long}, that a job in each queue is willing to wait — the scheduler guarantees the job will begin executing no later than W. Note that W_{short} and W_{long} are system-wide parameters, and *jobs do not need to specify any slack or deadlines.* In the rest of this section, we describe carbon-aware and carbon- and waiting-aware policies while assuming a constant pricing scheme, i.e., only on-demand instances. Then, we extend these policies to include cost awareness using cheaper reserved and/or spot instances.

4.2.1 Carbon-aware Batch Scheduling. To compute a carbon-aware start time t_{start} for a job arriving at time t, the scheduler computes a start time within the window $[t, t+W)$ that optimizes the carbon footprint of the job and runs the job until it finishes. The carbon-aware t_{start} can be computed in two ways. One way is to examine the carbon intensity forecasts in the window $[t, t+W)$ and choose the slot with the lowest carbon intensity to begin executing the job, denoted as the Lowest Carbon Slot (`Lowest-Slot`) policy. An alternative is to schedule the jobs around such low carbon periods, and choose the time segment of length J with the lowest carbon intensity. We refer to this approach as the Lowest Carbon Window (`Lowest-Window`) policy. In this case, the scheduler chooses $t_{start} \in [t, t+W)$ such that the total carbon footprint of the job in the interval $[t_{start}, t_{start} + J]$ is minimized. The total carbon footprint $\mathbb{C}(t_{start})$, at t_{start} is given by:

$$\mathbb{C}(t_{start}) = \sum_{t=t_{start}}^{t_{start}+J} c(t) \cdot e.$$

where $c(t)$ is carbon intensity at time slot t and e is the energy cost of the cloud server for a time unit. However, as mentioned in prior research [9, 18, 24], a batch scheduler may not know the job length J prior to execution and may only know a coarse upper bound based on the queue. In this case, the `Lowest-Window` policy uses the historical queue-wide average, J_{short}^{avg} and J_{long}^{avg} for selecting the start time.

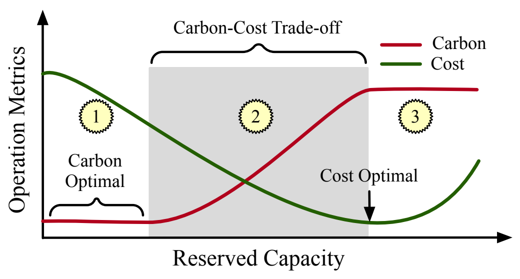
4.2.2 Waiting- and Carbon-aware Batch Scheduling. The above carbon-aware policies are oblivious to increases in completion time compared to the actual carbon savings. To consider the performance overhead of carbon savings, a scheduler can choose $t_{start} \in [t, t+W)$ such that the Carbon Saving per Completion Time, denoted as CST, is maximized. We call this policy `Carbon-Time`. Under this policy, when a job starts at t_{start}, its $CST(t_{start})$ is given by:

$$CST(t_{start}) = \frac{\mathbb{C}(t) - \mathbb{C}(t_{start})}{t_{start} + J - t}.$$

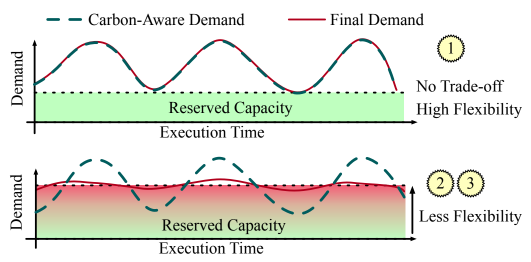
where $\mathbb{C}(t)$ is the carbon footprint of immediately starting the job, i.e., carbon agnostic footprint, and $\mathbb{C}(t_{start})$ is the carbon footprint at t_{start}. Similar to the earlier policies, GAIA utilizes the mean job length J_{short}^{avg} and J_{long}^{avg} as a coarse approximation of the job length.

4.2.3 Cost- and Carbon-aware Batch Scheduling. Our policies optimize carbon and consider the performance when running jobs exclusively on on-demand servers at t_{start}. However, as explained in Section 2.3, a typical cloud-based cluster utilizes a hybrid setting by using a "fixed" set of reserved instances to fulfill a portion of their demand at a cheaper cost. In this hybrid setting, when a job is dequeued, the resource manager schedules the job on a reserved instance if one is available, benefiting from the lower prices. Otherwise, it launches an additional on-demand instance to execute the job. Unfortunately, as highlighted in Section 3, this approach reduces the cluster utilization, decreasing the utility of the pre-paid reserved instances.

To maximize the utilization of reserved instances, and thus lower on-demand costs, a scheduler should operate in a work-conserving manner [9]. In this case, upon the arrival of a new job, if an idle reserved server is available, it is scheduled immediately for execution with no wait. Otherwise, the scheduler computes the carbon-aware t_{start} that minimizes the job's carbon footprint. If a reserved server becomes idle before the earliest t_{start}, the job with this t_{start} is started on this reserved server. Otherwise, the scheduler acquires an on-demand server at t_{start} and executes the job on that



(a) *The effect of reserved capacity on carbon and cost.*



(b) *Illustrating scheduling flexibility across operating regimes.*

Figure 4. *The carbon-cost trade-off.*

server. We refer to this policy as the Reserved First (`RES-First`) policy. The intuition behind this policy is that since these instances have been paid for, it is better to use a work-conserving approach that always schedules a newly arriving or a queued job whenever a reserved server becomes idle. This ensures cost-efficient execution using reserved servers.

Since cost-efficiency is crucial in this hybrid setting, a key issue is that the amount of reserved capacity, which dictates temporal flexibility and carbon emissions. For example, when using only on-demand, the `RES-First` boils down to a carbon-aware policy, resulting in the lowest carbon footprint. However, increasing the reserved capacity and ensuring its high utilization limits the temporal flexibility, and jobs may be executed as they arrive or at a suboptimal t_{start}. We illustrate this relationship in Figure 4. Figure 4a illustrates possible operating regimes. In regime ①, the carbon footprint is minimal, and acquiring reserved instances to cover the lower demand bound, see Figure 4b (top), will allow the scheduler to maintain its carbon savings while achieving cost savings. Regime ② shows the trade-off between carbon and cost, where the operator must choose the most critical objective. This scenario is illustrated in Figure 4b (bottom), where surpassing the demand bound increases the cost savings but limits the scheduler's flexibility to follow the carbon-aware schedule. Finally, in the regime ③, the reserved capacity is in excess, cannot satisfy the cost-breakeven utilization, and offers no temporal flexibility — a regime operators should always avoid. We note that increasing the reserved instances for a work-conserving policy always reduces waiting time.

4.2.4 Spot-aware Batch Scheduling.

On-demand instances tend to be the best option for minimizing carbon emissions. However, this comes at an increased operational cost for cloud users due to the higher cost of on-demand instances. While reserved instances helps with this issue, they significantly limit carbon savings. One alternative is to use spot instances, which are excess cloud resources rented at a discounted rate, e.g., 20% of the on-demand price until needed by a higher-paying customer so the current customer is evicted. The likelihood of being evicted (eviction rate), defined as the percent of evicted customers in a time slot, e.g., an hour, changes as the daily and weekly demand changes [46]. While there is a chance of eviction and loss of a job's progress, the high discount rate makes spot instances attractive.

To benefit from spot instances' discounted rate, `GAIA` uses spot instances to satisfy a portion of the demand. In this case, we compute a carbon-aware t_{start} and execute the job on a spot instance rather than an on-demand one. Since spot instances can be evicted, we currently assume that all of the job's progress is lost. This assumption is common in HPC settings [29] where application-agnostic system-level checkpoint/restart is challenging to implement [32]. However, in scenarios where checkpoint/restart functionality is available, then an additional tradeoff exists between the checkpointing overhead, eviction rate, and the amount of recomputation required on each eviction, as discussed in prior work [33, 34]. Exploring this additional tradeoff in the context of carbon, cost, and performance is future work.

The risk of being evicted and progress loss increases with execution time; thus, we use spot instances only for short job queues and restart the job on an on-demand instance if it is evicted. We refer to this policy as Spot First (`Spot-First`). The carbon and cost benefits of spot instances depend on the eviction and discount rates, where a higher eviction rate means more lost progress, which increases cost and carbon overheads. Although the added cost is fixed, it is a function of the difference between spot cost, on-demand cost, and amount of lost progress. The added carbon depends on the carbon intensity of the new execution window.

As a cloud user, combining different purchasing options for maximum benefits is common. `GAIA`'s batch scheduler follows this same approach by integrating various resources and types to get the most out of each purchasing option. Specifically, we integrate the Reserved First (`RES-First`) policy and Spot First (`Spot-First`) and use them for long and short jobs, respectively. The short jobs follow the `Spot-First` policy by running on spot instances, and in case they fail, they can run on either on-demand or reserved instances based on availability. On the other hand, long jobs strictly follow the `RES-First` policy and use on-demand only if the start time

t_{start} arrived and no reserved instance was available, a policy we denote as Short Spot Reservation (Spot-RES). We note that the combined policy is subject to the same trade-offs as the RES-First policy, where potential gains and drawbacks are experimentally detailed in Section 6.

5 Implementation

GAIA is designed as an extensible interface that can integrate with standard batch schedulers such as SLURM, Spark, etc. GAIA's current implementation relies on the AWS Slurm Cluster service, AWS ParallelCluster [1], an open-source management tool that enables creating and managing High-Performance Computing (HPC) clusters in AWS. It provides users with automatic resource scaling that can dynamically grow and shrink the cluster according to fluctuations in demand. The cluster utilizes AWS elasticity to scale the cluster based on resource demand and can be configured to use multiple instance purchasing options and sizes. We deploy GAIA along with the Slurm master node, which intercepts all incoming job submissions and queues them based on the underlying policy. The implementation uses PySlurm [3] to submit and monitor jobs, as well as monitor cluster status. We consider the entire instance time, including initiation and termination times, for carbon and cost accounting. Other accounting details are explained in Section 4.1.

To enable large-scale and year-long evaluation, we also implement a cloud simulator, called GAIA-Simulator, that incorporates GAIA's components by emulating their cost model, e.g., on-demand versus reserved pricing, and behavior, e.g., instance revocations in spot instances. The simulator follows the same interface and accounting methodology. Although the current version does not account for initialization and termination overheads, the results in Section 6 focus on normalized metrics, enabling us to neglect such overheads. The GAIA implementation for AWS and cloud emulators was done in Python using ~2k SLOC. The code is available at https://github.com/umassos/GAIA.

6 Evaluation

This section evaluates our GAIA prototype and its scheduling policies with regard to their carbon emissions, performance, and cost. We start by exploring carbon emissions and its effect on completion time using AWS ParallelCluster [1]. Next, we illustrate and quantify trade-offs in carbon-aware scheduling when using GAIA in hybrid settings. Finally, we leverage GAIA-Simulator to generalize our findings across large-scale workload traces, regions, and settings.

6.1 Experimental Setup

Workload Traces Generation. Our experiments use three real-world cluster traces: a two-month long *Alibaba-PAI* trace [43], a month-long *Azure-VM* trace [11, 16], and a five-year long Los Alamos National Lab (LANL) *Mustang-HPC*

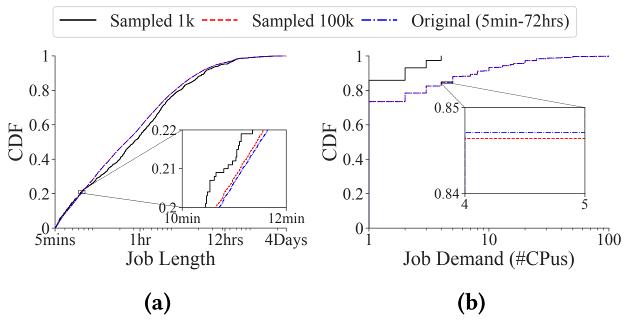


Figure 5. *Job length (a) and CPU demand (b) distributions between original and sampled* Alibaba-PAI *traces.*

trace [10]. Note that the traces include jobs with multiple tasks running in parallel. For example, the *Mustang-HPC* trace includes many parallel MPI jobs. We used these traces to construct synthetic traces as follows:

1) Sampling We used each original trace's job length information to construct year-long and week-long multi-node job traces. First, we uniformly sampled each original trace to construct a 100k job trace spanning a full year. Each such trace is used to simulate a year-long cluster run. Second, to evaluate the GAIA prototype, we uniformly sampled 1k jobs from the first week of the *Alibaba-PAI* trace forming a week-long trace. Since such trace is used to evaluate the GAIA prototype running on a real AWS ParalellCluster testbed, we limited the sampling to jobs that run on four CPUs or less for budgetary reasons. Further, when constructing these traces, we ignored very short jobs (i.e., jobs less than five minutes) and very long jobs (i.e., jobs longer than three days). We assume that very short jobs may not tolerate long delays of several hours and may not contribute to carbon consumption. For example, in the *Alibaba-PAI* trace, although jobs that are less than five minutes are 38% of the total number of jobs, they only contribute 0.36% of the total compute cycles. In addition, very long jobs spanning multiple days will see little benefit from being delayed due to the diurnal nature of carbon intensity variations [39].

2) Length Extension. To construct our year-long workload traces, we used the last year of the *Mustang-HPC* trace and replicated the *Azure-VM* and *Alibaba-PAI* traces multiple times and then sample from them. Although such replication does not capture seasonal demand changes, using a year-long trace captures the effects of seasonal variations in carbon intensity on the workload's carbon footprint.

3) Demand Normalization. Traces use different compute units, e.g., *Azure-VM* trace provides resource buckets, *Alibaba-PAI* trace uses GPUs, while *Mustang-HPC* trace uses a 24-core machine as a unit. We use these numbers as a proxy of CPU demand and assume that resources are homogeneous.

Figure 5 shows the distributions of job lengths and job demand for the year-long (100k) and week-long (1k) traces

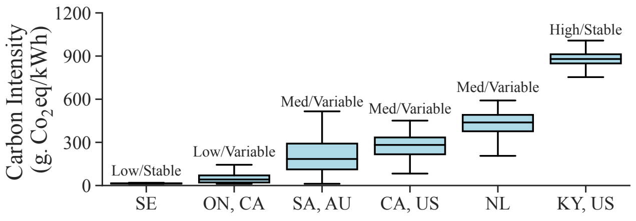


Figure 6. *Carbon intensity across diverse cloud regions.*

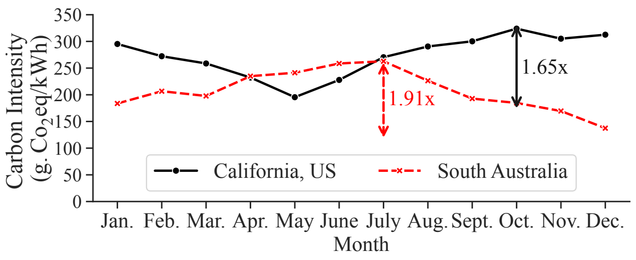


Figure 7. *Mean carbon intensity across different months in California, US and South Australia.*

constructed from the original *Alibaba-PAI* trace using the above process. The figure shows that both traces have job length distributions that resemble the original trace. The CPU demand distribution of the week-long trace is somewhat different from the original (due to the trace being limited to jobs with 4 CPUs or less for experimental tractability), while the year-long trace is similar to the original trace.

Carbon Intensity Traces. We use hourly carbon intensity traces from Electricity Maps [26] for 2022 from 5 geographical regions, shown in Figure 6. We group them into three categories based on average carbon intensity (Low, Medium, and High) and two categories based on variability (Stable and Variable). The selected regions are representative examples of various combinations of carbon intensity and variability in today's cloud regions. Figure 7 shows the variations in average carbon intensity in California, US and South Australia. In addition to daily variations, see Figure 1, and yearly variations, see Figure 6, the figure highlights year-long variations that significantly impact the total carbon consumption. For example, the carbon intensity in South Australia almost doubles between July and December. Finally, we assume perfect knowledge of the future carbon intensity, as prior work demonstrates that carbon intensity forecasts are highly accurate [25].

GAIA Deployment. We deploy GAIA in AWS where the head node uses a c7gn.xlarge instance, while workers use c7gn.medium. The c7gn.medium costs \$0.0624 per hour. We utilize actual pricing discounts for 3-year reserved and spot instances. The 3-year reserved and spot instance costs 40% and 20% of the on-demand instance price, respectively. Finally, we note that GAIA experiments on AWS Parallel cluster

Table 1. Summary of scheduling policies

Policy	Job Length	Carbon-Aware	Performance-Aware
NoWait [9]	-	-	-
AllWait-Threshold [9]	-	-	-
Wait Awhile [44]	Yes	Yes	-
Ecovisor [35]	-	Yes	-
Lowest-Slot	-	Yes	-
Lowest-Window	J_{avg}	Yes	-
Carbon-Time	J_{avg}	Yes	Yes

use an expedited time frame where we accelerate experiments by a factor of 5×.

Baselines. Table 1 summarizes our baseline policies and our proposed policies from Section 4.2 and their assumptions. We later explain how we use these policies in a work-considering manner or with spot instances.

1. **No Jobs Wait** (NoWait) [8]: This policy runs jobs as they arrive and represents a carbon- and cost-agnostic baseline.
2. **All Jobs Wait Threshold** (AllWait-Threshold) [8]: A cost-aware baseline that delays the job until a reserved instance is available or until the maximum waiting time is reached. Upon reaching the maximum waiting time, the scheduler runs the job on an on-demand instance.
3. **Wait Awhile** [44]: A carbon-aware policy that knows the job length J and the deadline. The policy schedules the workload by selecting time slots summing to J with the lowest carbon intensity within this deadline, which we set as $J + W$. To emulate such an assumption, we profile the workload and then configure the job to run for a predetermined number of cycles corresponding to a specific amount of time.
4. **Ecovisor** [35]: This policy uses a greedy threshold approach and executes the job if the current carbon intensity is below the threshold; otherwise, it pauses the job. We set the threshold to the 30th percentile of the carbon intensity over the next 24 hours. To ensure compliance with our waiting limits, the job is executed to completion after waiting for the allowed time.
5. **Proposed Policies**: The Lowest-Slot policy does not know the job length, the Lowest-Window, and Carbon-Time policies know a queue-wide average. Lastly, the Carbon-Time acknowledges the performance overheads.

For all scheduling policies, unless otherwise mentioned, the maximum time for the short queue (J_{short}) is 2 hrs, and all other jobs are submitted to the long queue. We assume that users accurately assign their short and long jobs to the appropriate job queue. Finally, we configure maximum waiting times for the short and long jobs (W_{short} and W_{long}) to be 6 and 24 hrs, respectively.

6.2 Carbon- and Performance-aware Scheduling

In this section, we examine scheduling policies for different job characteristics and assumptions using GAIA-prototype and the week-long (1k jobs) from the *Alibaba-PAI* trace.

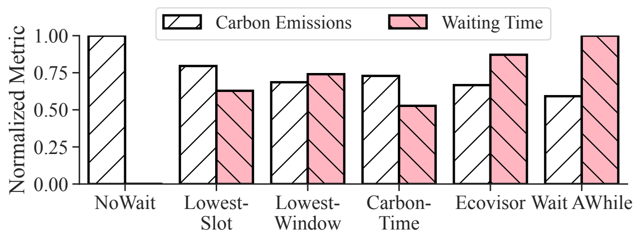


Figure 8. *Normalized carbon emissions and waiting times for different policies in South Australia.*

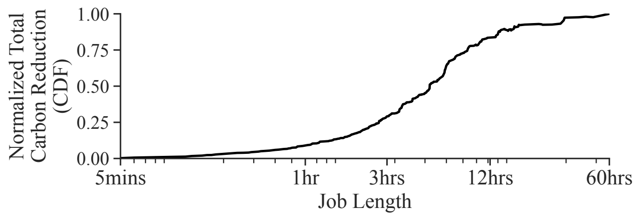


Figure 9. *CDF of normalized total carbon reduction for the* Alibaba-PAI *trace in South Australia across job length using the* `Carbon-Time` *policy.*

6.2.1 Scheduling Policies. We start by evaluating the reductions in carbon emissions and performance overhead of carbon-aware and carbon-performance-aware scheduling policies. Figure 8 depicts the carbon emissions and waiting times of six policies, see Table 1, normalized to the highest value in each metric. The figure shows that the suspend-resume policies (e.g., Wait Awhile and Ecovisor) achieve both the lowest carbon emissions and the highest performance penalty (i.e., highest waiting times), as such policies are often aggressive in their scheduling decisions and may delay a job until the latest possible moment if it yields any carbon savings. The figure also shows the ability of policies that only know a coarse-grained estimate of job length and do not require suspensions to achieve comparable carbon emissions. For instance, the `Lowest-Window` incurs 3% and 16% more carbon emissions compared to Ecovisor and Wait Awhile policies, respectively, without knowing the exact job length or interrupting executions. Finally, the figure shows that the `Carbon-Time` policy that considers both carbon savings and performance overhead can reduce the performance overhead by 50% compared to the WaitAwhile while adding 6% and 23% more carbon emissions compared to the `Lowest-Window` and Wait Awhile policies.

6.2.2 Effect of Job Characteristics. Job length and resource demand greatly influence carbon emissions [39]. Short jobs can be shifted to the time slots with the lowest carbon intensity, while long jobs will be subjected to daily carbon intensity patterns. In contrast, medium-length jobs have more potential for carbon savings, as they use more compute cycles than short jobs and can still be shifted to low-carbon periods. Figure 9 shows shows the CDF of total carbon reduction by job length using the `Carbon-Time` policy. Although jobs ≤1hr are almost 50% of the total jobs in the traces, see Figure 5a, they only contribute 10% to the total savings. Moreover, the figure shows that only 7.5% carbon savings come from long jobs(≥24 hrs), as they benefit less from temporal shifting since the carbon intensity has roughly a 24 hour period. Lastly, the graph shows that 50% of the carbon savings come from jobs between 3 and 12hrs since they consume the most CPU cycles in the cluster and have some flexibility to be shifted to the lowest carbon windows. ***Key Takeaways:*** *Coarse grain estimates of job length and understanding the carbon-performance trade-off enable carbon-aware scheduling policies to reduce waiting time by 50% while incurring 23% more carbon emissions compared to the Wait Awhile policy that knows job length and allows interruptible executions. Total carbon emissions savings from very short and very long jobs are negligible.*

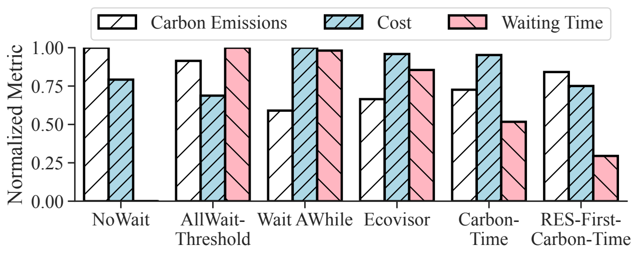


Figure 10. *Normalized carbon, cost, and waiting time across policies using 9 reserved instances, in South Australia.*

6.3 Cost- and Carbon-aware Scheduling

In this section, we consider a realistic hybrid setting where users are cost-conscious and financial costs are at least as important as carbon savings. We reveal the trade-off by employing the work-conserving (`RES-First`) and the spot-aware (`Spot-First`) variants of the proposed policies using our `GAIA`-prototype and the week-long (1k jobs) from the *Alibaba-PAI* trace.

6.3.1 Reserved-Aware Scheduling. As explained in Section 4.2.3, the amount of reserved capacity and its utilization impacts the cost and carbon consumption of the cluster. Figure 10 shows the performance of six scheduling policies (see Table 1) normalized to the highest value in each metric when running on 9 reserved instances. As shown, the `NoWait` policy gives the highest carbon footprint, while the `AllWait-Threshold` policy provides the lowest cost for this amount of reserved instances, but results in a high carbon consumption and the highest waiting time. The figure demonstrates the drawback of carbon-aware policies, which results in the highest costs while yielding the same carbon savings shown in Figure 8. The high execution costs of Wait Awhile and

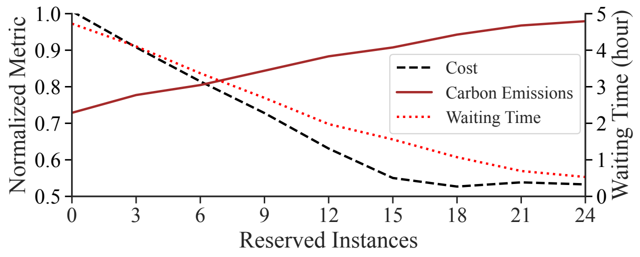


Figure 11. *Normalized carbon and cost w.r.t.* `NoWait` *(on-demand) execution (left-axis) and waiting time (right-axis) across reserved instances using the* `RES-First-Carbon-Time` *policy in South Australia.*

Ecovisor are due to the suspend-resume execution, which highly fragments the demand, leading to lower utilization. Finally, we show how the proposed `RES-First-Carbon-Time` policies balance the metrics, achieving the best performance and saving 21% of the operational cost while retaining 50% of the achieved carbon savings of the `Carbon-Time` policy.

Reserved instances and attentiveness toward their high utilization affect the scheduling decisions and the performance outcomes. This hybrid setting introduces a three-way trade-off between cost, carbon, and waiting time (performance). Figure 11 highlights this relationship using the `RES-First-Carbon-Time` policy. The figure shows normalized cost, carbon (left y-axis), and waiting time (right y-axis) with respect to the `NoWait` with no reserved instances. As shown, adding reserved instances, up to a certain point, decreases the cost as jobs benefit more from the cheaper reserved instances while limiting carbon savings as jobs increasingly run on reserved instances. The figure shows that selecting 18 instances yields the lowest cost while yielding 6% less carbon than the `NoWait` baseline. The figure also shows how users can select their trade-off point based on their objectives. For example, compared to selecting 18 reserved instances, users can use 15 reserved instances, increasing the carbon savings to 10% for an extra 4% cost, i.e., 55% cost-savings compared to a pure on-demand cluster. Finally, the results show that the average waiting time strictly decreases with the number of reserved instances, as it reduces the probability of staying in the waiting queue until the carbon-optimal start time.

6.3.2 Spot-Aware Scheduling.

Spot instances offer carbon and cost benefits by allowing the scheduler to follow a carbon-aware schedule while running at a discounted price. However, the benefits of spot instances are limited by the ability to acquire and keep the instance during the entire execution time. To address this, we only schedule short jobs (≤ 2 hrs) on spot instances, an assumption we revisit later.[1] Figure 12 depicts the carbon, waiting time, and cost of different policies and add the `Carbon-Time` policy with 0 reserved instances for reference. As shown, using the spot instances with carbon-aware policies, i.e., `Spot-First-Carbon-Time` and `Spot-First-Ecovisor`, maintains the carbon savings depicted in Figure 8, while providing 17% cost savings. The figure also shows that combining spot and reserved instances (e.g., `Spot-RES-Carbon-Time`) introduces the same trade-off between carbon and cost. However, the trade-off points will depend on the demand not covered by spot instances. For example, when using 9 reserved instances, the remaining load will decrease the cost, saving 42% compared to `Carbon-Time` with 0 reserved instances, but will force more long jobs to run at a sub-optimal start time, reducing the carbon savings to 15%. In contrast, when using 6 reserved instances, the system achieves 20% carbon savings for 11% extra cost compared to `Spot-RES` (9). Finally, we highlight that `GAIA` does not dictate the size of reserved capacity but provides the best carbon savings under any configuration.

[1]Spot instances were never evicted in our experiments.

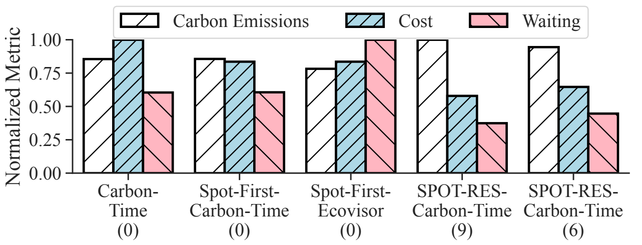


Figure 12. *Normalized carbon and cost when using spot and reserved instances. The The value (R) below each label represent the number reserved instances.*

Key Takeaways: *Using reserved and spot instances allows users to configure the carbon-cost trade-off point. Our results demonstrate that we can double the carbon savings per percentage increase in cost while decreasing performance overheads.*

6.4 Large Scale Experiments

In this section, we utilize `GAIA-Simulator` to evaluate our proposed scheduling policies in a simulated large cluster setting using the year-long (100k) real-world cluster traces from *Mustang-HPC* trace, *Alibaba-PAI* trace, and *Azure-VM* trace.

6.4.1 Trade-offs Across Workload Traces.

Figure 13 presents the normalized carbon emissions and waiting time for different workloads executed in US California. Figure 13a shows that the Wait Awhile policy achieves lowest carbon emissions and the highest performance overheads. The *Mustang-HPC* and *Azure-VM* traces show maximum carbon savings of 26% and 19%, respectively. The reason for this variation is attributed to the job distribution among traces. For example, the *Mustang-HPC* trace maximum job length is 16 hrs, allowing it to gain high savings, while the *Azure-VM* trace has long jobs that span across cycles of carbon intensity. The figure also depicts the sensitivity to the knowledge of

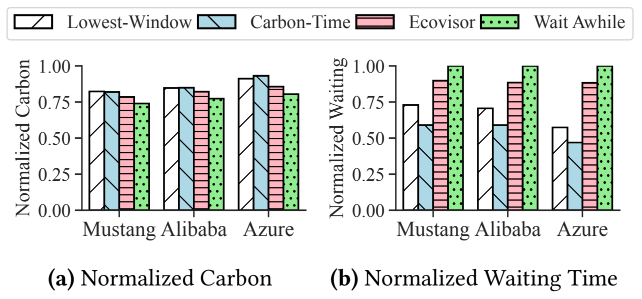


Figure 13. *Normalized carbon (a) and waiting time (b) across policies and cluster traces in US California.*

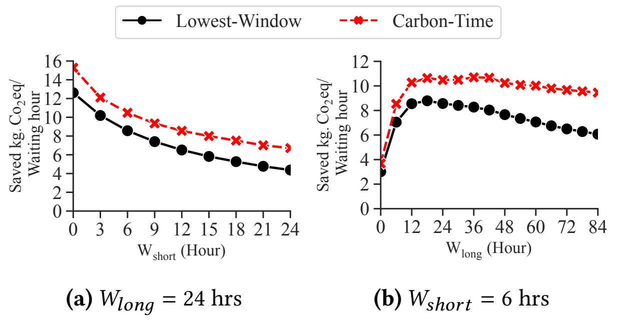


Figure 14. *Saved carbon per waiting times for different waiting time thresholds for the* Alibaba-PAI *trace in South Australia.*

the job and the benefits of suspend-resume execution. For example, compared to the Wait Awhile policy that knows the job length and can suspend the job, the `Lowest-Window` policy retains 68% of the carbon savings for the *Mustang-HPC* trace while only preserving 44% of the savings for the *Azure-VM* trace. This variation directly results from the job length distribution between traces where the job length average of the *Mustang-HPC* trace is representative of the whole trace, while jobs in *Azure-VM* are more variable. Finally, we demonstrate the benefits of the `Carbon-Time` policy, which can reduce the waiting time by 20% compared to the `Lowest-Window` policy while yielding comparable carbon savings.

Key Takeaways: *Under the same scheduling assumptions, augmenting carbon-aware policies with performance-awareness decreases waiting time by 20% with similar carbon savings.*

6.4.2 Effect of Waiting Time. Waiting time represents the performance penalty the user is willing to pay to obtain carbon savings. Although willingness to wait longer (i.e., lower performance) should yield higher savings, extending the waiting time yields diminishing gains [21, 39]. Moreover, as noted earlier, some carbon-aware schedulers delay jobs if it results in any carbon savings, which may not be worthwhile compared to the incurred performance overhead. Figure 14 shows the carbon saving per each waiting hour incurred by the user when applying the `Lowest-Window` and `Carbon-Time` policies for different maximum waiting time thresholds. Figure 14a shows the effect of W_{short} while fixing W_{long} = 24 hrs, while Figure 14b shows the effect of W_{long} while fixing W_{short} = 6 hrs. As depicted, extending the waiting time for short jobs results in lower savings per waiting hour. Short jobs constitute most of the workload trace and can significantly impact the mean waiting time, but have a limited effect on the overall carbon savings as stated earlier. Conversely, extending the waiting time for long jobs leads to higher savings, but there is a point beyond which waiting longer shows diminishing carbon savings but high waiting times. Lastly, although the `Lowest-Window` policy can reduce carbon more than the `Carbon-Time` policy, the `Carbon-Time` policy consistently outperforms the `Lowest-Window` policy by providing higher savings per waiting time. The results show that the `Carbon-Time` policy achieves 80-90% of the carbon savings of the `Lowest-Window` policy while decreasing waiting time by 20-30%.

Key Takeaways: *Increasing waiting time results in diminishing increases in carbon savings and considering the benefits of waiting will limit the carbon-aware schedulers' aggression towards limited carbon savings.*

6.4.3 Effect of Geographic Regions. Carbon intensity characteristics determine the possible carbon savings. For example, carbon intensity variations within the scheduling horizon dictate the potential savings a job can yield, while the carbon intensity, in g·CO_2eq, governs the total carbon savings. Figure 15 illustrates the normalized carbon emissions and total savings, compared to the `NoWait` policy, for various regions and workload traces using the `Carbon-Time` policy.

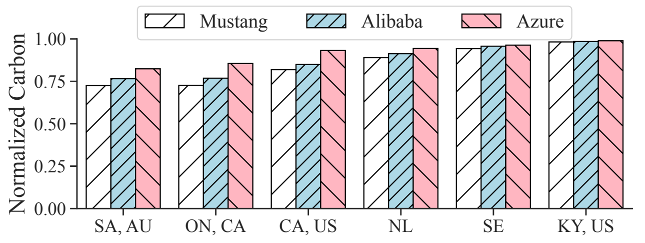


Figure 15. *Normalized carbon emissions across workloads and regions, using* `Carbon-Time` *policy.*

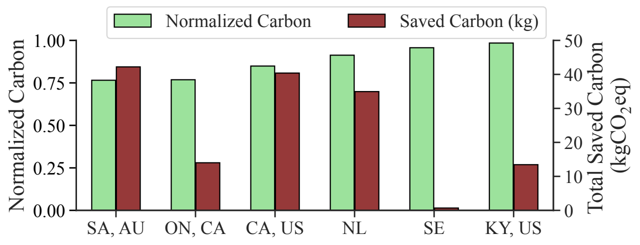
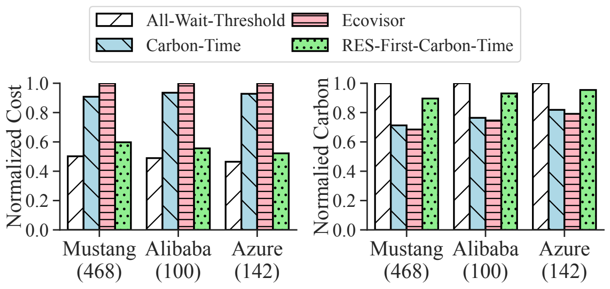
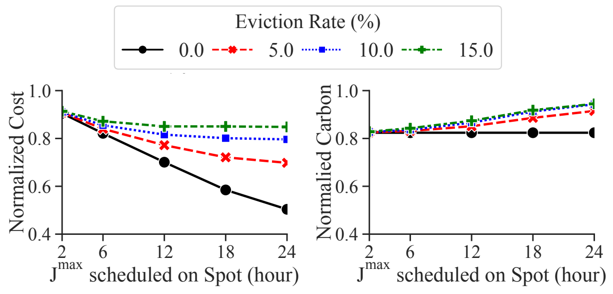


Figure 16. *Normalized and saved carbon emissions for the* Alibaba-PAI *trace across regions, using* `Carbon-Time` *policy.*



(a) *Normalized Cost* **(b)** *Normalized Carbon*

Figure 17. *Normalized cost and carbon emissions across workload traces and policies, in South Australia. The value (R) refers to the number of reserved instances for each trace.*



(a) Normalized Cost **(b)** Normalized Carbon

Figure 18. `Spot-First` *carbon and cost w.r.t.* `NoWait` *(on-demand) execution for* Azure-VM *trace in South Australia.*

As shown, regions with high variations result in low carbon emissions and vice versa. For example, South Australia, which has the highest variation, see Figure 6, yields the lowest emissions, achieving 27.5% less carbon emissions, while the Kentucky trace only yields 1% less carbon emissions. Furthermore, it is worth noting that waiting time, which we omit for space constraints, remains the same across regions for the same workload trace.

Effective carbon-aware scheduling policies are judged not only by normalized carbon savings but also by their total carbon reductions. Figure 16 illustrates the normalized and total carbon savings of the *Alibaba-PAI* trace using the `Carbon-Time` policy. The graph shows the impact of regional carbon intensity characteristics depicted in Figure 6, where regions with higher variations results in lower normalized carbon emissions. The results also show that, although the total savings in Ontario, Canada, and Kentucky, US, are the same at 10 kg·CO_2eq, their normalized savings differ 20%. Thus, users should consider their decisions' total carbon reduction, rather than normalized savings, to judiciously select their carbon-performance-cost trade-off configurations across operating regions.

Key Takeaways: *Carbon consumption and savings varies significantly across regions, while performance overhead (waiting time) is static aside from these variations.*

6.4.4 Reserved-First Scheduling.

Reserved instances introduce a trade-off between cost and carbon savings that varies across workload traces and policies. Figure 17 shows the normalized cost and carbon per workload traces compared to the highest value across policies in South Australia. Since traces' compute demand differs, we allocate a different number of reserved instances R per trace, where R is selected as the trace's mean demand, which results in high cost-savings as mentioned in [8].

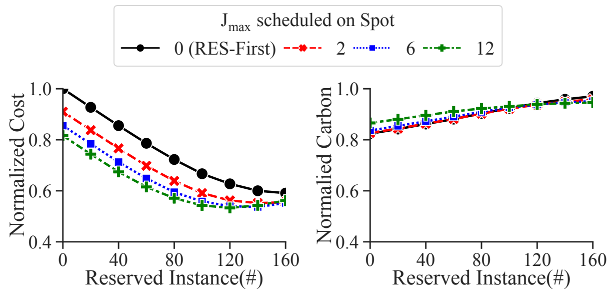
As expected, the `AllWait-Threshold` policy yields the highest carbon consumption and cost savings up to 46%, while the Ecovisor policy yields the highest cost, adding 117 and 12% more cost compared to the `AllWait-Threshold` and the `Carbon-Time` policies. In contrast, the `RES-First-Carbon-Time` can bridge the gap by adding up to 9% more cost than the `AllWait-Threshold`, while using up to 11% more carbon than the Ecovisor policy. The results also depict variations among traces where the *Azure-VM* trace shows the highest cost savings and lowest carbon reductions while the *Mustang-HPC* trace shows the lowest cost savings and highest carbon reduction. This is related to the demand variability where the mean demand — the used number of reserved instances — matches the demand where most jobs are executed in the reserved instances. The demand coefficient of variant (standard deviation/mean) for the *Mustang-HPC* and *Azure-VM* traces are 0.8 and 0.3, respectively.

Key Takeaways: *The variations in demand limit the cost savings but increase the scheduling flexibility, which in turn allows for higher carbon savings.*

6.4.5 Hybrid Cloud Clusters.

Spot instances relax the carbon-cost tension by allowing cost savings while following the carbon-aware schedule. However, spot instances may be evicted at any time, and the chance of being evicted increases with execution time. Figure 18 details the potential gains and overheads of the `Spot-First` policy across different settings when replaying the *Azure-VM* trace in South Australia. The x-axis depicts the maximum allowed time to be executed on the spot instance, i.e., J^{max}. We test with various eviction rates that typically appear in a real datacenter and assume that evicted instances progress is lost. The case of zero eviction presents an upper bound on cost savings and retains the same carbon savings as the carbon aware policy, which is computed using the (`Carbon-Time`) policy.

As expected, extending J^{max} without evictions is always beneficial in cost and maintains all the carbon savings. However, with the introduction of eviction, extending J^{max} yields no or diminishing cost savings while strictly increasing carbon consumption. For example, when the eviction rate is 15%, extending J^{max} beyond 6 hrs yields no cost savings but increases the carbon by up to 12%. The reason for such



(a) Normalized Cost **(b)** Normalized Carbon

Figure 19. *Normalized carbon and cost across J_{max} values for* Azure-VM *trace in South Australia, with a 10% eviction rate.*

increases is that longer jobs are more likely to be evicted, where the amount of lost progress, which incurs cost overhead, outweighs the cost and gains from trying to follow the carbon-aware schedule.

Mixing cloud instances yields a different trade-off as it allows high carbon savings from following the carbon-aware schedule by using spot instances and gaining cost savings from the discounted rate of spot and reserved instances, realized by the `Spot-RES-Carbon-Time` policy. We note that the `Spot-RES-Carbon-Time` policy boils down to `Spot-First-Carbon-Time` when reserved is 0 and `RES-First-Carbon-Time` when spot instances are not utilized. Figure 19 illustrates the effect of extending the reserved capacity across different J^{max} for the *Azure-VM* trace in South Australia, with a 10% eviction rate. As shown, extending the reserved capacity shows similar trends across all cases. However, the lowest cost point achieves higher carbon savings since the demand is partially split between regular and spot instances. For example, when considering 12hr jobs (J^{max} = 12) for spot instances, the lowest cost is achieved by having 120 reserved instances while achieving 7% carbon savings. Similarly, when (J^{max} = 6), the lowest cost point (140 instances) offers 5.5% carbon savings.

Key Takeaways: *Using of spot instances for long jobs has a negative effect on cost and carbon.* Hybrid clusters that use on-demand, spot, and reserved instances can partially overcome the carbon-cost tension.

7 Discussion

We have shown that a trade-off exists between carbon emissions, performance, and cost in hybrid batch schedulers that include both reserved and on-demand resources. We list our findings to help users select their appropriate trade-off point.

1. **Consider both carbon and performance.** Scheduling to minimize only carbon can introduce a high performance penalty for comparatively little carbon savings. Thus, considering both carbon savings and performance, e.g., in `Carbon-Time` policy, is important when making scheduling decisions.
2. **Waiting for 12hrs balances carbon and performance.** Configuring the waiting time at the knee of the carbon-performance frontier by setting the waiting time to 12hrs for long jobs, see Figure 14, strikes a balance between carbon and performance.
3. **Delaying medium-length jobs is most beneficial.** Medium (3-12hrs) jobs have the most potential to reduce carbon emissions as they can be flexibly moved to low-carbon slots. In contrast, delaying several-day jobs has less carbon reduction potential as they are subjected to diurnal carbon intensity variations, as demonstrated in Figure 9.
4. **Reserve between the base and the mean demand.** Users can reduce their cost without affecting the scheduling flexibility by reserving enough capacity to satisfy their base demand (See regime 1 from Figure 4). Further increases in reserved instances till the mean demand allows users to configure the trade-off point as shown in Figure 11 and Figure 17.
5. **Use spot instances for short jobs.** Spot instances can alleviate the carbon-cost tension when used with short jobs to avoid eviction overheads.

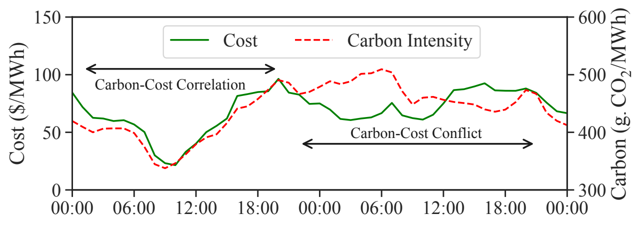


Figure 20. Carbon intensity and energy cost for June 7-8 2022 in US, Texas.

Selecting an appropriate trade-off point depends on user requirements and workload demand patterns. However, using a mixture of reserved, somewhere between the baseload and average demand, and spot instances increases cost savings. In addition, using the `Carbon-Time` policy, focusing on medium-length jobs, and configuring the scheduler to allow 12hrs of waiting time can increase the carbon reductions.

This paper focuses on carbon-cost trade-offs introduced by different cloud pricing models and their utilization from a cloud customer perspective. However, this trade-off also presents itself in private clouds due to dynamic energy pricing. Thus, as the compute cost varies throughout the day, a carbon-aware schedule might not comply with a cost-aware one. For example, when the carbon intensity and cost valleys are aligned, the provider can find a schedule that optimizes both carbon and cost. Otherwise, the user is left with a trade-off between carbon and cost. Figure 20 shows the ERCOT electricity grid (US, Texas) carbon intensity [26] and pricing per MWh [2] for two consecutive days. The figure demonstrates both cases where the first day of a carbon-aware schedule is also cost-effective. In contrast, the second day depicts a mismatch between carbon and cost, where the users are forced to prioritize one over the other. Finally, we note that carbon intensity and cost data from ERCOT showed a correlation coefficient of 0.16; thus, operators are left with a similar carbon-cost trade-off.

Finally, while our work considers an explicit carbon-performance-cost trade-off, an alternative approach is to assign an explicit cost to carbon and thus reduce the problem to a simpler cost-performance trade-off. For example, a cost may be assigned to carbon emissions by applying a carbon tax or mandating the purchase of carbon offsets. Assigning such a direct cost to carbon would then enable policymakers to adjust the incentive to reduce carbon emissions by adjusting the cost. Thus, a high carbon tax would translate to high carbon periods also being high cost periods. A few countries have such a carbon tax, although most countries, including the United States, do not [5], likely because a carbon tax would raise energy costs [28]. Mandating the purchase of carbon offsets can serve a similar role as a carbon tax in assigning a cost to carbon. Some countries even permit the use of carbon offsets to satisfy carbon taxes [4]. Importantly, even if governments impose a carbon tax or mandate the purchase of carbon offsets, to simplify the trade-off, cloud platforms would need to expose this carbon cost to cloud users by incorporating it into their resource cost. Currently, even in countries with a carbon tax or mandatory carbon offset scheme, cloud platforms do not vary resource prices based on carbon emissions.

8 Related Work

Carbon-aware Scheduling. Prior work has employed temporal shifting to schedule jobs at low carbon slots [17, 35, 39, 40, 44], spatial shifting by considering the carbon intensity across multiple possible cloud regions [17, 36, 39], and demand regulation by adjusting the job demand according to the carbon intensity by scaling jobs or curtailing their demand without pausing them [20, 21, 35, 40]. However, prior work generally focuses on minimizing the carbon emissions of a single job, while GAIA focuses on quantifying the carbon-cost-performance trade-off.

Perhaps the most relevant work is presented in [30, 45, 48] where the authors of [48] discuss the role of capacity caps in carbon-aware scheduling along with key opportunities and challenges. The authors of [30] explore such a mechanism in carbon-aware management of private clouds containing interactive and batch workloads by applying a global virtual capacity limit to minimize carbon consumption while adhering to SLO constraints. Another relevant approach is presented in [45], where the authors enforce a datacenter-level power cap at high carbon periods by modulating the demand through temporal shifting and demand throttling of both interactive and batch jobs using the power-performance trade-offs across heterogeneous workloads while fairly attributing such resource throttling across multiple workloads. Similarly, GAIA employs temporal shifting but considers only batch workloads. In addition, GAIA focuses on the cloud customer's perspective while these works are from a cloud provider's perspective and thus do not consider the cost trade-off for cloud users introduced by on-demand, spot, and reserved instances. Nonetheless, using resource caps across different purchase options instead of carbon-aware scheduling policies, as in GAIA, can yield similar carbon-performance-cost trade-offs. Finally, we note that, although not tackled in these works, providers may face a carbon-cost trade-off due to variations in energy prices.

Overheads of Carbon Reduction. The trade-offs introduced by carbon-aware scheduling have been mentioned in earlier work [6, 20, 21, 40] without providing methods to address them. The authors of [40] highlighted that increasing completion time is a byproduct of carbon-aware scheduling. Additionally, in [20, 21], the authors demonstrated that exploiting computing flexibility for carbon reduction increases energy consumption and cost. Finally, the authors of [6] noted the expected demand changes in carbon-aware scheduling. In contrast, we quantify this conflict's breadth and propose practical mechanisms to overcome these overheads.

9 Conclusion

In this paper, we analyzed the costs of carbon reduction and showed the fundamental tension between carbon emissions, performance, and cost. We showed the breadth of these trade-offs in multiple settings and key reasons behind such tension. To limit this tension, we presented GAIA and scheduling policies that navigate the trade-offs between carbon, performance, and cost. We have shown different ways to explore the three-way trade-off under different provisioning mixtures and offer methods to increase carbon savings with minimal cost and performance increases. In future work, we will focus on other carbon-saving modalities, such as scaling, and evaluate them in geographically federated clusters.

Acknowledgements

We thank the ASPLOS reviewers and our shepherd, John Wilkes, for their valuable comments, which improved the quality of this paper, and electricityMap for access to their carbon-intensity data. This research is supported by NSF grants 2211302, 2211888, 2213636, 2105494, US Army contract W911NF-17-2-0196, DOE award DE-EE0010143, VMware, and Amazon Web Services.

A Artifact Appendix

A.1 Abstract

This artifact contains the GAIA scheduler's source code, the proposed scheduler in the paper: Going Green for Less Green: Optimizing the Cost of Reducing Cloud Carbon Emissions, and execution instructions. GAIA is written in Python and requires 1) `pcluster`: AWS ParallelCluster management command line interface and 2) `PySlurm`: Python-based SLURM interface. GAIA contains two interfaces: an AWS ParallelCluster interface and a simulation interface. The AWS ParallelCluster is used to demonstrate our policies in a real testbed, while the simulation is used for large-scale experiments. The artifact also includes a configurable MPI program that illustrates different job lengths and resource requirements in the AWS ParallelCluster (Slurm) testbed. The artifact details instructions for executing GAIA in both environments and lists instructions for repeating Figures 8-12 in the simulation environment. We provided instructions for running the simulations and a Jupyter notebook to plot the results.

A.2 Artifact check-list (meta-information)

- **Algorithm: Scheduling polices explained in Section 4.2**
- **Program: GAIA scheduler with AWS ParallelCluster (Slurm) and simulation interfaces.**
- **Compilation: `cmake v3.10+`, `libopenmpi-dev/openmpi-bin v4.1`, and `PySlurm v23.2.2`.** [2]
- **Data set: Utilized job traces and carbon-traces are provided.**
- **Run-time environment: Code do not require specific environment but AWS tests used `Ubuntu 20.04`, AWS ParallelCluster v3.6.1, and `Slurm v23.02.2`.**
- **Metrics: Carbon Emissions, Dollar Cost, and Waiting Time.**
- **Output: CSV files**
- **Experiments: We demonstrate the tension between carbon, performance (waiting time) and cost. We also show the effect of policies and knowledge assumptions**
- **How much time is needed to prepare workflow (approximately)?: Required simulations take 10 minute per experiment. Real-experiments uses 2-3 days per experiment**.
- **How much time is needed to complete experiments (approximately)?: Simulation: 1 hours, Real: 50 Days (Can be executed in parallel)**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT**
- **Archived (provide DOI)? 10.5281/zenodo.10888009**

A.3 Description

GAIA implements two software interfaces for simulation and for AWS ParallelCluster testbed. We list instructions for both separately.

[2]Only required for AWS tests and instructions provided.

A.3.1 How to access. The code can be pulled from the public Git repository https://github.com/umassos/GAIA, where the branch 'artifact' is prepared for artifact evaluation. The dependencies are PySlurm and AWS ParallelCluster CLI. Detailed installation steps are available in the `README.md`.

A.3.2 Hardware dependencies. The code does not have any hardware requirements. The AWS ParallelCluster tests were executed on instance type `c7gn.medium`.

A.3.3 Software dependencies. The simulations and plotting were implemented using `pandas v1.4.3`, `matplotlib v3.5.3`, and seaborn v0.12.0.

The AWS ParallelCluster tests were executed using `Ubuntu 20.04` and `Slurm`. The sample MPI job requires `cmake`, `libopenmpi-dev`, and `openmpi-bin` for compilation. The scheduler required `PySlurm v23.2.2` to communicate with the `Slurm v23.02.2` scheduler.

A.3.4 Data sets. All the required data is provided in the repository.

A.4 Installation

To download the code:

```
git clone -b artifiact \
github.com:umassos/GAIA
```

A.4.1 Simulation Environment. The simulation relies on pandas and numpy to install requirements use:

```
pip3 install -r requirements.txt
```

A.4.2 AWS ParallelCluster Environment. In addition to the simulation dependencies, running GAIA in AWS ParallelCluster (Slurm) requires ParallelCluster CLI, `PySlurm`, and an executable job. The `README.md` contains 1) Installing AWS ParallelCluster CLI tool, 2) sample configuration files used to create the cluster, 3) The code and complication details for an `N`-body simulation MPI job to be used in our experiments, and 4) `PySlurm` installation details.

A.5 Experiment workflow

The experiments demonstrate the conflict between performance, carbon, and cost. To execute the experiments, the GAIA scheduler loads a carbon trace, workload trace, and cluster assumptions (reserved instance, spot, etc.). GAIA uses the selected scheduling algorithms, job length knowledge assumptions, and user configuration (waiting time) to select each job start time. We list a few examples here. Details are available in the `README.md`

Example 1: To run in a carbon and cost-agnostic manner.

```
python3 src/run.py --scheduling-policy \
cost --carbon-policy waiting -w 0x0
```

Example 2: To run using the lowest carbon window (using cluster-wide aggregate) and maximum waiting 6hrs for short jobs (<2hrs) and 24hrs for long jobs.

```
python3 src/run.py --scheduling-policy \
carbon --carbon-policy waiting -w 6x24
```

The default is to execute simulation. To use the AWS ParallelCluster run the code inside the cluster master and provide the `--cluster-type slurm` flag.

```
ssh user@masternodeIP
python3 src/run.py --cluster-type slurm
```

A.6 Evaluation and expected results

To reproduce Figures 8-12, we provide four bash scripts that customize and run the experiments with the needed configuration. We provided a jupyter notebook to plot the Figures in `notebooks/evaluation_ploti□pynb`.
- Figure 8: Normalized carbon emissions and waiting times across policies.
- Figure 9: CDF of the normalized total carbon reductions.

```
./src/figure8-9.sh
```

- Figure 10: Normalized Carbon, Cost, and Waiting Time across policies when using reserved instances.

```
./src/figure10.sh
```

- Figure 11: Effect of reserved instances on the carbon savings and cost using a work-conserving and carbon-aware scheduling policy.

```
./src/figure11.sh
```

- Figure 12: Effect of both spot and reserved instances on the carbon savings and cost using multiple policies and configurations.

```
./src/figure12.sh
```

The result for each experiments are an aggregate file that contains the total consumption, a details file that contains the consumption of each job, and a run time file that contains the allocation and carbon consumption during the execution time.

A.7 Experiment customization

Users can customize their configurations and use examples from the evaluation section.

1. Carbon Trace: Carbon Intensity (per hour) for expected experiment duration.
2. Carbon Index: Start index within the trace to test seasons and times of day.
3. Workload Trace: Workload trace (jobs with length and CPU requirements).
4. Scheduling Policies (`README.md` provides the mapping between policies and configurations).
5. Cluster Configuration: Types of used resources, e.g., Reserved, Spot, and On-Demand.

A.8 Notes

The code to execute the jobs in AWS ParallelCluster is available, and we are happy to work with reviewers to show how it runs. However, we are asking the reviewers to reproduce results in simulation, as reproducing the results in the AWS ParallelCluster testbed is time-consuming and costly.

Also, we omitted the instructions to run Figures 13-19 as they are a super-set of the findings in Figures 8-12 and require many hours in simulation as traces are year-long and have 100k jobs. However, we provided the used traces and will provide the details if the reviewers need them.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] AWS ParallelCluster. https://docs.aws.amazon.com/parallelcluster/, 2023.

[2] ERCOT electricity grid market prices. https://www.ercot.com/mktinfo/prices, September 2023.

[3] PySlurm: Python client library for the Slurm Workload Manager. https://github.com/PySlurm/pyslurm, 2023.

[4] Carbon Offset Guide. https://www.offsetguide.org/understanding-carbon-offsets/carbon-offset-programs/mandatory-voluntary-offset-markets/, Acessed March 2024.

[5] Which countries have a carbon tax? https://ourworldindata.org/carbon-pricing, Accessed March 2024.

[6] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 118–132, 2023.

[7] Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma, and Prashant Shenoy. SpotWeb: Running Latency-sensitive Distributed Web Services on Transient Cloud Servers. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 1–12, Phoenix, Arizona, June 2019.

[8] Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy. Waiting Game: Optimally Provisioning Fixed Resources for Cloud-enabled Schedulers. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–14, November 2020.

[9] Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy. Good Things Come to Those Who Wait: Optimizing Job Waiting in the Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 229–242, 2021.

[10] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, July 2018.

[11] Microsoft Azure. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset, Accessed October 2020.

[12] Noman Bashir, David Irwin, Prashant Shenoy, and Abel Souza. Sustainable Computing – Without the Hot Air. In *Proceedings of the First Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon)*, 2022.

[13] Eric A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 167, 2015.

[14] Amanda Calatrava, Eloy Romero, Germán Moltó, Miguel Caballer, and Jose Miguel Alonso. Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures. *Future Generation Computer Systems*, 61:13–25, 2016.

[15] Wesley J Cole, Danny Greer, Paul Denholm, A Will Frazier, Scott Machen, Trieu Mai, Nina Vincent, and Samuel F Baldwin. Quantifying the challenge of reaching a 100% renewable energy power system for the United States. *Joule*, 5(7):1732–1748, 2021.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, 2017.

[17] 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. Measuring the carbon intensity of ai in cloud instances. In *2022 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '22, pages 1877–1894, 2022.

[18] Ana Gainaru, Guillaume Pallez Aupy, Hongyang Sun, and Padma Raghavan. Speculative scheduling for stochastic hpc applications. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 32:1–32:10, 2019.

[19] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[20] Walid A. Hanafy, Roozbeh Bostandoost, Noman Bashir, David Irwin, Mohammad Hajiesmaili, and Prashant Shenoy. The War of the Efficiencies: Understanding the Tension between Carbon and Energy Optimization. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, 2023.

[21] Walid A. Hanafy, Qianlin Liang, Noman Bashir, David Irwin, and Prashant Shenoy. CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(3), December 2023.

[22] Urs Hölzle. Meeting Our Match: Buying 100 Percent Renewable Energy. https://blog.google/outreach-initiatives/environment/meeting-our-match-buying-100-percent-renewable-energy/, 2018.

[23] IEA. Global trends in internet traffic, data centre workloads and data centre energy use, 2010-2019. https://www.iea.org/data-and-statistics/charts/global-trends-in-internet-traffic-data-centre-workloads-and-data-centre-energy-use-2010-2019, 2022.

[24] Michael Kuchnik, Jun Woo Park, Charles D. Cranor, Elisabeth Moore, Nathan Debardeleben, and George Amvrosiadis. This is why ml-driven cluster scheduling remains widely impractical. Technical report, CMU-PDL-19-103, 2019.

[25] Diptyaroop Maji, Prashant Shenoy, and Ramesh K. Sitaraman. 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*, BuildSys '22, page 198–207, 2022.

[26] Electricity Maps. Electricity Maps. https://www.electricitymap.org/map, Accessed September 2022.

[27] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52, 2010.

[28] Congressional Budget Office. Effects of a Carbon Tax on the Economy and the Environment. https://www.cbo.gov/publication/44223. May 2013.

[29] Lucas Perotin, Chaojie Zhang, Rajini Wijayawardana, Anne Benoit, Yves Robert, and Andrew Chien. Risk-Aware Scheduling Algorithms for Variable Capacity Resources. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1306–1315, 2023.

[30] 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. Carbon-Aware Computing for Datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2022.

[31] Ana Radovanović, 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. Carbon-aware Computing for Datacenters. *IEEE Transactions on Power Systems*, 2023.

[32] Manuel Rodriguez-Pascual, J.A. Morinigo, and Rafael Mayo-Garcia. Checkpoint/Restart in Slurm: current status and new developments. https://slurm.schedmd.com/SLUG16/ciemat-cr.pdf, March 2024.

[33] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-Interactive Data-Intensive Processing for Transient Servers. In *ACM European Conference on Computer Systems (EuroSys)*, EuroSys '16, London, United Kingdom, 2016.

[34] Supreeth Shastri, Amr Rizk, and David Irwin. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, Salt Lake City, Utah, November 2016.

[35] Abel Souza, Noman Bashir, Jorge Murillo, Walid Hanafy, Qianlin Liang, David Irwin, and Prashant Shenoy. Ecovisor: A Virtual Energy System for Carbon-Efficient Applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 252–265, March 2023.

[36] Abel Souza, Shruti Jasoria, Basundhara Chakrabarty, Alexander Bridgwater, Axel Lundberg, Filip Skogh, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. CASPER: Carbon-Aware Scheduling and Provisioning for Distributed Web Services. In *Proceedings of the 14th International Green and Sustainable Computing Conference (IGSC), Toronto, ON, Canada*, 10 2023.

[37] Garrick Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8, 2006.

[38] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. SpotOn: A Batch Computing Service for the Spot Market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*, pages 329–341, Kohala Coast, Hawai'i, August 2015.

[39] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David Irwin, and Prashant Shenoy. On the Limitations of Carbon-Aware Temporal and Spatial Workload Shifting in the Cloud. In *Nineteenth European Conference on Computer Systems (EuroSys)*, Athens, Greece, 2024.

[40] John Thiede, Noman Bashir, David Irwin, and Prashant Shenoy. Carbon Containers: A System-level Facility for Managing Application-level Carbon Emissions. In *Proceedings of 14th Symposium on Cloud Computing (SoCC)*, 11 2023.

[41] Noelle Walsh. Achieving 100 percent Renewable Energy with 24/7 Monitoring in Microsoft Sweden. https://azure.microsoft.com/en-us/blog/achieving-100-percent-renewable-energy-with-247-monitoring-in-microsoft-sweden/, 2020.

[42] WattTime. WattTime. https://www.watttime.org/.

[43] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 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), pages 945–960, 2022.

[44] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. Let's Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud. In *Proceedings of the 22nd International Middleware Conference (Middleware)*, page 260–272, December 2021.

[45] Jiali Xing, Bilge Acun, Aditya Sundarrajan, David Brooks, Manoj Chakkaravarthy, Nikky Avila, Carole-Jean Wu, and Benjamin C. Lee. Carbon Responder: Coordinating Demand Response for the Datacenter Fleet. *E-Print arXiv 2311.08589*, 2023.

[46] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Terry Yang, Victor Rúhle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Snape: Reliable and Low-Cost Computing with Mixture of Spot and On-Demand VMs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 631–643, 2023.

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

[48] Chaojie Zhang and Andrew A. Chien. Scheduling Challenges for Variable Capacity Resources. In Dalibor Klusáček, Walfredo Cirne, and Gonzalo P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 190–209, 2021.