Caspian: A Carbon-aware Workload Scheduler in Multi-Cluster Kubernetes Environments

Tayebeh Bahreini
Wayne State University
Detroit, MI
tayebeh.bahreini@wayne.edu

Asser N. Tantawi
IBM T.J. Watson Research Center
Yorktown Heights, NY
tantawi@us.ibm.com

Olivier Tardieu
IBM T.J. Watson Research Center
Yorktown Heights, NY
tardieu@us.ibm.com

***Abstract*—The surge in demand for computing resources in data centers coupled with the rise of environmental concerns has motivated cloud providers to reduce carbon emission due to computational energy consumption. An opportunity lies in the fluctuating availability of renewable energy over time and the variability of power sources over grid regions, leading to variations in space and time in carbon intensity. Exploiting such variations, this paper introduces Caspian, a carbon-aware workload scheduler in multi-cluster Kubernetes environments, which aims at reducing the carbon footprint (CFP) due to executing workloads, while satisfying Quality of Service (QoS) requirements. Caspian cooperates with a multi-cluster management platform to apply scheduling and placement decisions over distributed clusters. We present efficient optimization algorithms to achieve these goals. Further, we describe an implementation of Caspian, integrated with Multi Cluster App Dispatcher (MCAD), a multi-cluster management platform which handles queuing and dispatching of workloads over multiple clusters. Our experimental results show that Caspian effectively reduces CFP with reasonable QoS, compared to a baseline scheduler which only satisfies the QoS of workloads. Specifically, Caspian reduces CFP by about 33%, with about 98% of workloads completing at an average fraction of 0.6 of their deadline.**

***Index Terms*—cloud sustainability, green computing, Kubernetes, multi-cluster management, placement, scheduling.**

I. Introduction

Over the past few years, due to the surge in using cloud computing for executing compute intensive jobs, such as artificial intelligence and machine learning (AI/ML) workloads, data centers have contributed to the production of a significant CFP compared to some other industries such as the airline industry. Data centers consume approximately 1–2% of global electricity production [13], resulting in an estimated 2.5–3.7% of global carbon emission [7]. This trend will most likely continue in the next decade, leading to a concerning contribution to global warming.

Many technology companies have invested in renewable energy to lower the energy cost and the environmental impact of their services [11], [12]. These companies either generate their own renewable energy onsite or directly supply it from renewable power plants. However, achieving carbon-free computing still remains challenging due to the uncertain and variable availability of renewable energy sources such as solar and wind. One potential solution to tackle this variability is to store energy. However, there are concerns about using batteries for renewable energy storage such as cost and their impact on the environment. Another approach is to dynamically manage the execution of workloads to match the low-carbon energy supply. This approach requires revisiting the design of resource management platforms in data centers as they mainly focus on optimizing the trade off between performance, utilization, and energy consumption [14].

In recent years, an extensive research has been conducted on carbon-aware resource management in cloud computing systems. These studies address dynamic workload management by exploiting temporal and/or spatial variability of carbon intensity of electricity of data centers, as well as taking advantage of the fact that many types of batch workloads, such as AI/ML training jobs, have a substantial temporal and geographical flexibility. For example, Google traces indicate that a high percentage of jobs submitted to the Borg scheduler have a low priority and fall in the free and best-effort-batch tiers with weak Service Level Agreements (SLAs) [18]. Similar characteristics are observed for jobs in Meta. For example, in Meta, about 87% of offline data processing workloads have completion time Service Level Objectives (SLOs) more than four hours, with the majority having 24-hour SLOs. This offers considerable flexibility for adjusting the schedule of workloads based on the availability of low carbon energy [1].

Enabling temporal and spatial shifting of workloads requires the deployment of dynamic scheduling algorithms, which take real time data about power consumption, carbon intensity of electricity, and workload characteristics and make efficient scheduling and placement decisions. A body of research has considered temporal shifting of workloads employing optimization techniques such as linear programming, matching, and greedy techniques [2], [4], [8], [10], [15]. These studies considered various performance metrics in their objective functions such as electricity price, carbon price, carbon-intensity of grid energy, and QoS. In addition to temporal shifting, many studies considered the spatial workload placement among geographically distributed data centers [3], [16], [19], as well as distributed web services [17]. These efforts optimize various objective functions including minimizing carbon footprint, and/or maximizing the profit of cloud provider while guaranteeing QoS of workloads. They employed optimization techniques such as linear programming, distributed algorithms, reinforcement learning, and federated learning.

Combining temporal and spatial workload shifting, we

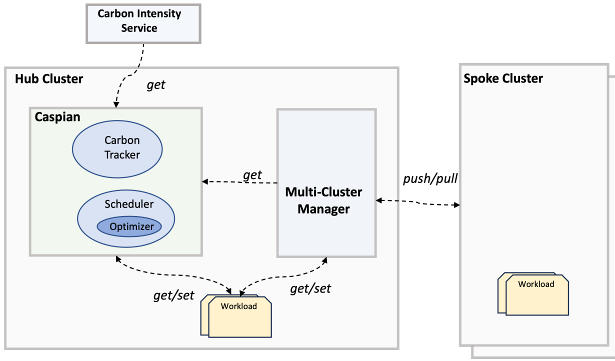


Fig. 1: A high level architecture of Caspian in a multi-cluster environment.

introduce Caspian, a carbon-aware workload scheduler in multi-cluster Kubernetes[1] environments. In summary, our main contributions are as follows.

- We introduce a multi-objective optimization problem with the aim of: (1) minimizing carbon emission by running workload when and where low-carbon energy is available; (2) minimizing the average lateness of workloads; and (3) minimizing the average completion time of workloads.
- We design efficient algorithms to address the complexity of the optimization problem and deploy them in Caspian.
- We design a framework in which Caspian collaborates with multi-cluster management platforms to apply its decision in the target clusters.
- We implement Caspian and deploy it in a running Kubernetes multi-cluster environment.
- We demonstrate that, using real-world carbon intensity data and workloads with various characteristics, Caspian achieves high carbon reduction compared to a baseline scheduler which only considers the QoS of workloads.

II. System model and problem statement

Caspian is a batch scheduler in a multi-cluster Kubernetes environment which decides when and where workloads are executed, based on the sustainability aspects of the clusters such as carbon intensity of energy sources and server power efficiency. Figure 1 shows a high level architecture of Caspian and its interactions in a multi-cluster environment. We consider a *hub-spoke* configuration consisting of a master (hub) cluster, as a multi-cluster control plane, and several worker (spoke) clusters, as target clusters managed by the hub to run workloads. Caspian, as a scheduling and placement controller, lives in the hub, interacting with a multi-cluster manager which handles queuing, dispatching, and execution of workloads in the spoke clusters.

Caspian consists of two main components: a carbon tracker and a scheduler. The carbon tracker (1) gets the geographical location of the spoke clusters from the multi-cluster manager and (2) fetches periodically the predicted values of carbon intensity for those locations from external sources, such as a carbon intensity service. The scheduler runs periodically. At the beginning of each period, it takes the information of spoke clusters (such as carbon intensity, power efficiency, and resource availability), as well as characteristics of workloads (such as run time, resource requirement, and deadline), and determines the best scheduling and placement for workloads using an optimizer. Once these decisions are made, Caspian updates the scheduling specifications of workloads' Kubernetes manifest files, notifying the multi-cluster manager about the scheduling decisions for the next time slot. Next, the multi-cluster manager applies these decisions in the destination clusters and updates the status of workloads accordingly. More details about the multi-cluster manager and its interactions with Caspian is provided in Section IV.

A. Problem statement

This section describes the optimization problem that is addressed by the optimizer in Caspian. The multi-cluster environment considered here consists of a hub cluster and M spoke clusters. We consider a discrete time slotted system in which the decisions are made periodically, based on the status of the system, over a time horizon of T *time slots*.

We denote by $\mathcal{N}$ the set of workloads. Each workload i is represented by the tuple (a_{ij}, r_i, l_i, d_i). Attribute $a_{ij} \in \{0, 1\}$ is the policy of workload i for cluster j; it is zero if workload i is not allowed to be executed on cluster j; and one otherwise. Attribute r_i is the required amount of compute resources, l_i is the run time of execution of workload i, and d_i is the deadline. In this formulation, we consider a single type of compute resource, but the formulation could be easily extended to multiple types. Here, meeting the deadline of a workload is a soft constraint, i.e., the optimizer is not forced to schedule the workload before its deadline; but violating the deadline has an impact on the value of the objective function.

We denote the set of spoke clusters by $\mathcal{M}$. Each cluster j is represented by the tuple (C_j^t, I_j^t, ρ_j), where C_j^t is the available computing resources and I_j^t is the carbon intensity during time slot t in cluster j. Attribute ρ_j is an estimate of power efficiency in cluster j. We assume that power consumption is linear[2] in resource utilization for all clusters [6]. The dynamic power consumption is estimated as the product of resource utilization and the constant factor ρ_j. Thus, the carbon emission associated with the dynamic power consumption in cluster j in time slot t is the product of the resource utilization, power consumption factor ρ_j, and carbon intensity I_j^t. Further, we assume that job runtime is invariant to the cluster, as it may request specific hardware.

We address the problem of Multi-Objective Scheduling and Placement (MOSP) of workloads over multiple geographically distributed clusters. The decision variable is matrix $\mathbf{x}$ of binary variables x_{ij}^t, where x_{ij}^t is one if workload i starts at time slot t in cluster j; and zero otherwise. The primary goal of MOSP

[1] https://kubernetes.io

[2] This is asymptotically true for large data centers and concave power profiles, as local energy-aware schedulers would tend to pack servers.

is to minimize the CFP due to the execution of workloads. To avoid infeasibility due to the limited resources, we do not force the optimizer to schedule all workloads; rather we encourage the optimizer to maximize resource utilization when and where low carbon energy is available. Thus, our first objective is to maximize $Z_1(\mathbf{x})$ given by,

$$Z_1(\mathbf{x}) = \sum_{j \in \mathcal{M}} \sum_{t \in \mathcal{T}} \frac{\sum\limits_{i \in \mathcal{N}} \sum\limits_{t' \in \Delta_i^t} x_{ij}^{t'} \cdot r_i}{C_j^t \cdot I_j^t \cdot \rho_j}. \tag{1}$$

In this formulation, quantity $(\sum\limits_{i \in \mathcal{N}} \sum\limits_{t' \in \Delta_i^t} x_{ij}^{t'} \cdot r_i)/C_j^t$ represents the resource utilization at time slot t in cluster j. The resource utilization of a cluster in a particular time slot is calculated as the ratio of resource requirement of all workloads running during the time slot over cluster capacity. Workload i is running on cluster j during time slot t if it is started in any time slots in the interval $\Delta_i^t = \{t - l_i + 1, ..., t\}$.

A second objective is concerned with the QoS of workloads. Here, the goal is to schedule workloads before their deadlines. However, due to the limited compute resources, some workloads may not meet their deadlines. To handle such cases, and also to avoid infeasible solutions, we assume that meeting deadlines is a soft constraint. Thus, instead of guaranteeing that all workloads are scheduled before their deadlines, we attempt to schedule as many workloads as possible while minimizing the sum of the lateness of workloads, or equivalently maximizing $Z_2(\mathbf{x})$ given by,

$$Z_2(\mathbf{x}) = \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{M}} \sum_{t \in \mathcal{T}} \frac{x_{ij}^t}{\gamma(i,t)}, \tag{2}$$

where $\gamma(i,t)$ is a penalty function for lateness. This penalty function discourages the decision maker to schedule a workload after its deadline. A simple definition of the penalty function could be $\gamma(i,t) = \max(1, l_i + t - d_i)$, which monotonically decreases the objective value with the increase of deviation of completion time of workload from its deadline.

A third objective controls the completion time of workloads. Scheduling the current workloads while not considering upcoming workloads in the future may result in inefficient utilization of compute resources. Let us consider a scenario in which the current workloads have extended deadlines. Under objective functions Z_1 and Z_2, the optimizer may decide to postpone the execution of workloads and schedule them later when low carbon energy is available, while a heavy demand may also arrive to the system in the future. This may cause some future workloads to miss their deadlines. One approach to handle such cases is to schedule the current workloads while reserving resources for the predicted future load [3]. This approach requires the prediction of future load. In this paper, we assume that either the prediction of future load is not available or the future load is hard to predict. Consequently, we define a third objective function, $Z_3(\mathbf{x})$, which controls the completion time of workloads and does not let workloads be executed very late, even if they still have time ahead of their deadlines,

$$Z_3(\mathbf{x}) = \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{M}} \sum_{t \in \mathcal{T}} \frac{x_{ij}^t}{\lambda(i,t)}, \tag{3}$$

where $\lambda(i,t)$ is the penalty function of completion time. Given this penalty function, the decision maker is encouraged to schedule workloads as soon as possible. Here, we simply consider the completion time as the penalty function, i.e., $\lambda(i,t) = t + l_i$. This function may also be defined by considering an estimation of arrival rate of future workloads.

Given objective functions $Z_1(\mathbf{x})$, $Z_2(\mathbf{x})$, and $Z_3(\mathbf{x})$, we formulate the MOSP problem as an integer programming problem.

$$\max_{\mathbf{x} \in X} (Z_1(\mathbf{x}), Z_2(\mathbf{x}), Z_3(\mathbf{x})) \tag{4}$$

subject to:

$$\sum_{i \in \mathcal{N}} \sum_{t' \in \Delta_i^t} x_{ij}^{t'} \cdot r_i \leq C_j^t \quad \forall j \in \mathcal{M}, t \in \mathcal{T}, \tag{5}$$

$$\sum_{j \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{ij}^t \leq 1 \quad \forall i \in \mathcal{N}, \tag{6}$$

$$x_{ij}^t \leq a_{ij} \quad \forall i \in \mathcal{N}, \forall j \in \mathcal{M}, t \in \mathcal{T}, \tag{7}$$

$$x_{ij}^t \in \{0,1\} \quad \forall i \in \mathcal{N}, \forall j \in \mathcal{M}, t \in \mathcal{T}, \tag{8}$$

where X is the feasible set of decision matrices. Constraint (5) guarantees that for each cluster j, the total amount of resources allocation cannot exceed its available capacity. Constraints (6)-(7) consider the policy of the workload for clusters and also ensure that each workload is scheduled on a maximum of one cluster. Constraint (8) represents the integrality requirements for the decision variables.

B. Problem Complexity

The special case of MOSP with a single-objective function $\max Z_1(\mathbf{x}), \mathbf{x} \in X$, i.e., maximizing resource utilization when and where low-carbon energy is available, is NP-hard [3]. This implies that MOSP, with multiple objective functions is NP-hard. Furthermore, solving problems with multiple objective functions is not as straight forward as it is with single-objective problems. In multi-objective optimization problems with conflicting objectives, the goal is to find Pareto optimal solutions; the solution that could not be dominated by another feasible solution. It might be possible to obtain the Pareto optimal solutions of the problem when there is a few number of workloads in the system. But with a large number of workloads it is practically impossible to solve the problem in a reasonable amount of time. In the next section, we introduce an efficient algorithm to solve MOSP in polynomial time.

III. Caspian Algorithms

The scheduler determines the schedule time as well as the target cluster for execution of workloads. The scheduler works in a timely manner where the length of the interval between its invocations is determined based on the characteristics of workloads such as QoS requirement and the average execution time of workloads. A high level description of the scheduler algorithm is given in Algorithm 1. The input to the scheduler

Algorithm 1 Scheduler (λ)

```
1: while true do
2:     (M, N) ← getClusterAndWorkloadInfo()
3:     y ← optimizer(M, N)
4:     for each i ∈ N do
5:         if isValid(y[i]) then
6:             setTarget(i, y[i])
7:         else
8:             suspend(i)
9:     wait(λ)
```

is δ, the length of the interval between periods. At the beginning of each period, the scheduler gets all available spoke clusters as well as the workloads (Line 2) and saves their characteristics in sets $\mathcal{M}$ and $\mathcal{N}$, respectively. Then, it calls the optimizer to find the best scheduling and placement for the workloads (Line 3). The scheduler has two choices. It could ask the optimizer to decide on the scheduling and placement of (1) the queued workloads or (2) the running workloads, by revisiting its previously made decisions, in addition to the queued workloads. In this implementation, the scheduler considers the second option and asks optimizer to decide about all workloads in the system, those currently running in the spoke clusters as well as workloads queued in the hub cluster. The output of the optimizer is a vector $\mathbf{y}$ which indicates the destination cluster for each workload. Here, $y_i = j$ means that workload i is scheduled on cluster with index j and $y_i = -1$ means that workload i is scheduled for a time period later than the current time period. For workloads with a valid target cluster index, the scheduler sets the target cluster field in the scheduling specification of the workload Kubernetes manifest, notifying the multi-cluster manager where to dispatch them. For workloads that are not scheduled for the upcoming time slot, the scheduler sets the sustainability gate in the scheduling specification of the workload Kubernetes manifest, notifying the multi-cluster manager to suspend them (Lines 4-8).

The optimizer solves the MOSP optimization problem with three objectives: $Z_1(\mathbf{x})$, $Z_2(\mathbf{x})$, and $Z_3(\mathbf{x})$. Many techniques for solving multi-objective optimization problems are based on converting the original problem into a single objective problem [5], known as scalarization. Consequently, we construct a single objective function by summing the individual objective functions, each multiplied by a coefficient, as

$$Z(\mathbf{x}) = \bar{\omega}_1 \cdot Z_1(\mathbf{x}) + \bar{\omega}_2 \cdot Z_2(\mathbf{x}) + \bar{\omega}_3 \cdot Z_3(\mathbf{x}),$$

where $\bar{\omega}_k$ is the normalized value of ω_k, the preference value for objective function $Z_k(\mathbf{x})$. Due to the inconsistent units of the objective functions, normalization of the preferences of objectives is necessary. Normalization plays a significant role in achieving the optimal solutions with the given preferences for objectives. Several studies have focused on various normalization methods [9]. However, the most efficient approaches consider normalizing by the magnitude of the optimal values, where each optimal value is obtained by solving an optimization problem with a single objective function. Since it is not possible to find the optimal solution for each single objective in a reasonable amount of time, we consider the LP relaxation of each problem and obtain coefficient $\theta_k = 1/\bar{Z}_k$,

Algorithm 2 Optimizer($\mathcal{M}, \mathcal{N}$)

1: $_, \theta_k \leftarrow$ LPSolver$(Z_k, \mathcal{M}, \mathcal{N}) \quad \forall k \in \{1,2,3\}$
2: $\bar{\omega}_k \leftarrow \frac{\omega_k}{\theta_k} \quad \forall k \in \{1,2,3\}$
3: $Z(\mathbf{x}) \leftarrow \sum_{k \in \{1,2,3\}} \bar{\omega}_k \cdot Z_k(\mathbf{x})$
4: $\bar{\mathbf{x}}, _ \leftarrow$ LPSolver$(Z, \mathcal{M}, \mathcal{N}\})$
5: $x_{ij}^t \leftarrow 0 \quad \forall i \in \mathcal{N}, j \in \mathcal{M}, t \in \mathcal{T}$
6: $y_i \leftarrow -1 \quad \forall i \in \mathcal{N}$
7: $b_i \leftarrow \sum_{j \in \mathcal{M}} \sum_{t \in \mathcal{T}} \bar{x}_{ij}^t \quad \forall i \in \mathcal{N}$
8: Sort workloads in non-increasing order of b_i.
9: **for each** $i \in \mathcal{N}$ **do**
10: $\quad (j^*, t^*) \leftarrow$ Allocation$(i, \mathbf{x}, Z, \mathcal{M})$
11: $\quad$ **if** isValid(j^*) **then**
12: $\quad\quad x_{ij^*}^{t^*} \leftarrow 1$
13: $\quad$ **if** $t^* == 1$ **then**
14: $\quad\quad y_i \leftarrow j^*$
15: **Output:** $\mathbf{y}$

where $\bar{Z}_k$ is the optimal objective value obtained by solving the LP relaxation of the problem with the single objective Z_k. Even with this simplification, the scheduling and placement problem is NP-hard and we need to find an efficient algorithm that solves the problem in reasonable amount of time.

Algorithm 2 describes the optimizer. First, the optimizer calls the LP-Solver to find the optimal LP solution with each of the objective functions. The output of LPSolver is the scheduling matrix and the objective value of the solution. Here, since we are only interested in the objective values, i.e., $\theta_1, \theta_2, \theta_3$, we do not save the scheduling matrix (Line 1). Then, the optimizer, based on the LP solution formulates the objective function $Z(\mathbf{x})$ and calls the LPSolver; this time to obtain LP solution for the MOSP problem with $Z(\mathbf{x})$ as the objective function (Lines 2-4). The solution is saved in matrix $\bar{\mathbf{x}}$. Next, it builds a feasible solution $\mathbf{x}$ from the fractional solution $\bar{\mathbf{x}}$. For this purpose, we initialize an allocation matrix $\mathbf{x}$ to zero (Line 5). We also initialize target vector $\mathbf{y}$ to save the target cluster for those workloads that are scheduled for the upcoming time slot (Line 6). The algorithm sorts the workloads in non-increasing order of their total allocation value, b_i, obtained by LPSolver (Lines 7-8). Then it chooses an unallocated workload with the maximum value of b_i and calls allocation algorithm to find the best time slot and cluster to fit the workload. If a valid spot (j^*, t^*) is found, the algorithm updates the allocation matrix and target vector and proceeds to the next candidate workload (Lines 9-14).

Algorithm 3 describes the allocation algorithm which finds the best target cluster and best schedule time for each workload. The input to this algorithm are the candidate workload i, the current allocation matrix $\mathbf{x}$, the objective function Z, and the set of spoke clusters $\mathcal{M}$. First, the algorithm sorts all pairs of (target cluster j, scheduled time slot t) for workload i in descending order of z_{ij}^t values (Line 1). Next, the algorithm tries to find the first pair of (j, t) in which the workload could be fitted (Lines 3-7). Here, fit$(i, \mathbf{x}, j, t)$ checks if workload i with current allocation $\mathbf{x}$ is schedulable in time slot t and cluster j or not. For this purpose, the fit procedure checks if both constrains 5 and 7 are satisfied.

Algorithm 3 Allocation$(i, x, Z, \mathcal{M})$

1: Sort all $(j,t) \in \mathcal{M} \times \mathcal{T}$ in non-increasing order of z_{ij}^t.
2: $(j^*, t^*) \leftarrow (-1, -1)$
3: **for each** $(j,t) \in \mathcal{M} \times \mathcal{T}$ **do**
4: **if** $\mathsf{fit}(i, \mathbf{x}, j, t') \quad \forall t' \in \{t, \dots t + l_i - 1\}$ **then**
5: $(j^*, t^*) \leftarrow (j, t)$
6: **break**
7: **Output:** (j^*, t^*)

IV. Implementation

Caspian[3] is an open source Kubernetes controller written in Golang. Caspian uses Multi-Cluster App Dispatcher (MCAD)[4] as a workload queuing and multi-cluster management platform to dispatch and handle workloads in the destination clusters. In this section, we describe MCAD in more detail and elaborate on its interaction with Caspian.

A. Multi Cluster App Dispatcher (MCAD)

MCAD is a Kubernetes controller that enables the deployment of batch jobs in multi-cluster environments. It provides mechanisms to guarantee that sufficient resources are available in the destination clusters before creating pods for executing workloads. MCAD deploys an AppWrapper (AW) custom resource to wrap any Kubernetes object including service, job, deployment, and other custom resources defined by the user. AW custom resource helps MCAD to support batch jobs and gang scheduling in Kubernetes clusters. Kubernetes objects within an AW stay in the queue until the total requested resources are available in the destination cluster.

At the AW level, a user can specify resource requirement for each object, expected run time of the workload, the deadline for the completion time of the workload, and the policies for the clusters. In addition to workload characteristics, there are scheduling specification fields in AW to support scheduling and placement in multi-cluster environment. One of these fields is *dispatchingGates* which indicates the list of gates that are placed by different placement engines on the workload. Each placement engine (such as Caspian) may have some policies on the workloads. For instance, Caspian may want to suspend an AW from execution due to sustainability concerns. To accomplish this, Caspian adds a sustainability gate to this field requesting MCAD to suspend the execution of a workload. Another scheduling related field in AW is *targetCluster* which is set by the placement engines. For example, if the optimizer decides to place a workload on a specific cluster Spoke1, the scheduler in Caspian sets the *targetCluster* field in AW to Spoke1, notifying MCAD that this workload is scheduled on Spoke1. Then, MCAD will do its part in managing the life cycle of the workload in Spoke1.

MCAD works in two different modes: dispatcher mode and runner mode. In the hub cluster, MCAD runs in dispatcher mode (MCAD Dispatcher), while in each spoke cluster, it runs in runner mode (MCAD Runner). MCAD Dispatcher and MCAD Runner coordinate with each other to manage queuing,

[3]https://github.com/sustainablecomputing/caspian
[4]https://github.com/project-codeflare/mcad

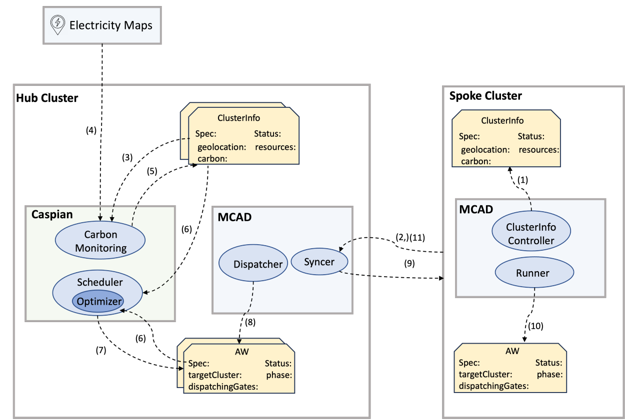


Fig. 2: Integration of Caspian with MCAD.

error handling, and execution of AWs through the Syncer component in MCAD. In addition to the AW, MCAD introduces the ClusterInfo custom resource, which represents the characteristics of clusters, such as resource availability, power consumption characteristics, geolocation, and carbon intensity. Some of these fields such as resource availability, power consumption characteristics, and geolocation are updated by the ClusterInfo Controller (another component of MCAD) in spoke clusters, while some other characteristics such as carbon intensity could be set by other external controllers such as the carbon tracker component in Caspian.

B. Integration of Caspian with MCAD

A summary of interactions between MCAD and Caspian is depicted in Figure 2. (1) In each spoke cluster, ClusterInfo controller periodically gets the geo-location of the cluster as well as the available resources in the cluster and updates ClusterInfo custom resource accordingly. (2) The Syncer in the hub cluster, periodically upsyncs the objects from the spoke clusters to the hub cluster. This way, Caspian has access to the updated status of ClusterInfo as well as the status of AWs executing in the spoke clusters. (3) In the hub cluster, Caspian runs periodically. At the beginning of each period, it gets the geo-location of the spoke clusters. (4) Then, it fetches carbon intensity data of those specific zones from Electricity Maps[5], a tool that provides real-time information about the carbon intensity of electricity generation in various regions. (5) Carbon Tracker component in Caspian updates the carbon intensity field in ClusterInfo. (6) Next, Scheduler in Caspian gets the list of spoke clusters as well as AWs. Then, it calls the optimizer to find the best scheduling and placement for AWs. (7) Finally, the scheduler updates *targetCluster* for those AWs that are scheduled during the upcoming time slot and removes the sustainability gates from the list of *dispatchingGates*. For those AWs that are not scheduled during the upcoming time slot, the scheduler adds a sustainability gate to the list of *dispatchingGates*. (8) MCAD Dispatcher is notified about these changes. Those AWs that are in Running or Dispatching phases, and now are requested

[5]https://app.electricitymaps.com/map

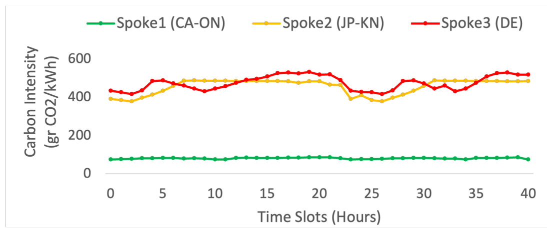


Fig. 3: Hourly carbon intensity of clusters.

by Caspian for suspension, will transit to Requeuing phase (to be queued again). Those AWs that are scheduled during the upcoming time slot, if they are ready to dispatch from MCAD perspective, the MCAD dispatcher changes their status to Dispatching phase. (9) MCAD Syncer pushes the scheduled AWs to the corresponding spoke clusters. (10) MCAD Runner executes workloads in the spoke cluster and updates their status accordingly. (11) The status of AWs are pulled by the MCAD Syncer in the hub cluster, letting users aware of the status of their workloads.

V. Experimental Analysis

In this section, we compare the performance of Caspian for different types of problem instances. First, we investigate the impact of sustainability of clusters on the scheduling and placement decisions made by Caspian and compare the results with a baseline which does not consider sustainability of clusters in its decision making. We show that Caspian schedules a high percentage of workloads within their deadline while it also significantly reduces the carbon footprint due to the execution of workloads. Then, we vary the sustainability weight (ω_1) and investigate its impact on lateness, carbon emission and completion time of workloads. We show that by increasing the sustainability weight, a higher amount of carbon is reduced; but a higher percentage of workloads may miss their deadline. We conclude that great care has to be taken when setting the values of the weights ω_k.

A. Experimental Setup

In conducting our experiments, we consider a configuration consisting of four Kubernetes clusters: a hub cluster and three spoke clusters that are denoted by Spoke1, Spoke2, and Spoke3. All clusters are created using k3d[6]. Each spoke cluster consists of two homogeneous nodes, each equipped with 16 GPUs. Thus, each spoke cluster has a total of 32 GPU cores. We assume that spoke clusters are located in three different electricity grid regions. Spoke1 is located in Ontario, Canada (CA-ON), Spoke2 in Kansai, Japan (JP-KN), and Spoke3 in Germany (DE). Figure 3 shows the carbon intensity of electricity source of each cluster during the 40-hour period starting from November 14th, 2023. We employ Electricity Map to get carbon intensity data for the grid regions. The data indicates that electricity source of Spoke1 has the lowest carbon intensity, whereas electricity sources of Spoke2 and Spoke3 both illustrate high carbon intensities.

We assume that Spoke2 is twice more power efficient than other spoke clusters and set $\rho_1 = \rho_3 = 200,\ \rho_2 = 100$. This implies that although Spoke2 and Spoke3 both are powered by a high carbon intensity electricity source; Spoke2 is more power efficient and is expected to be more preferable by Caspian. In our experimental setup, we focus on long-running workloads with execution time ranging from one hour to five hours, e.g. AI training jobs. We exclude short-running workloads from our experiments as they typically come with tight deadlines and cannot tolerate delays. We use Poisson distribution for the arrival of 200 workloads over 24 hours. Each workload is represented by an AW consisting of a BusyBox[7] image to execute the workload's task. The task assigned to each AW is a simple UNIX *sleep* command[8].

To perform an extensive experiments, we consider various characteristics of the workloads. The GPU requirement of each workload is randomly drawn from the uniform distribution $U(1, 5)$. Similarly, the run time for each workload is randomly chosen from another independent uniform distribution, $U(1, 5)$. We assume that workloads have deadlines proportional to their run times. Thus, the deadline is calculated based on a slowdown factor β. For example, for workload i, if the workload is submitted in time slot t, the deadline of workload is defined as $d_i = t + \beta \times l_i$, where the value of β is randomly chosen from uniform distribution $U(2, 4)$.

B. Analysis of Results

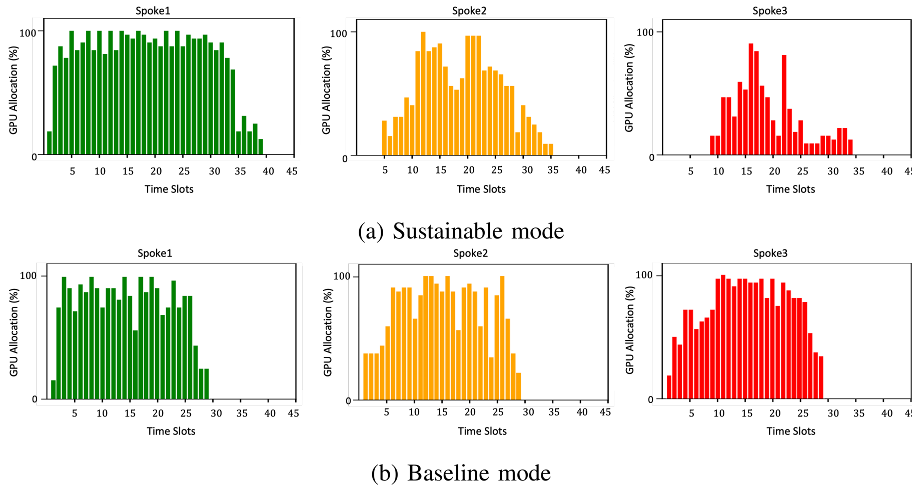
First, we investigate the impact of sustainability of clusters on the scheduling and placement decisions. We run Caspian in two different modes but under the same environmental conditions: (1) sustainable mode in which Caspian considers a non-zero weight for sustainability objective, i.e., $\omega_1 = 1$, and (2) baseline mode in which the weight for sustainability objective is zero, $\omega_1 = 0$. For both modes, we consider the same preference for the other objective functions, namely $\omega_2 = \omega_3 = 1$.

Figure 4a illustrates the GPU allocation of clusters over time slots when Caspian is in the sustainable mode. We observe that Caspian first starts allocating workloads to Spoke1 which has the lowest carbon emission rate. With the increase of the load over time, Caspian starts using Spoke2 (at time slot $t = 5$) to manage QoS of workloads. Later, it also uses Spoke3 (time slot $t = 9$) to schedule workloads. Then, when the peak time is passed, Caspian reduces the allocations on Spoke2 and Spoke3 and relies more on Spoke1. Thus, in all, Spoke1 has the highest GPU allocation compared to the other spoke clusters. We also observe that although the carbon intensity of electricity source for Spoke2 and Spoke3 is in the same range, Spoke2 receives higher workloads requests compared to Spoke3. The reason is that Spoke3 has a lower power efficiency, hence not preferred by Caspian. Another observation from the figure is the makespan of execution of workloads. We observe that all workloads are completed by time slot $t = 39$. Figure 4b illustrates the GPU allocation of clusters when Caspian is in

[6]https://k3d.io

[7]https://busybox.net

[8]Note that, in our experiments, we do not directly measure utilization, rather we assume it to be proportional to resource demand.



(a) Sustainable mode

(b) Baseline mode

Fig. 4: GPU Allocation of clusters over time slots running Caspian in (a) sustainable mode; (b) baseline mode.

the baseline mode. In this mode, a lower makespan is achieved (29 hours). We also observe that Caspian distributes workloads equally among clusters to guarantee low average completion time and low average lateness of workloads.

Figure 5 compares the overall performance of Caspian in the two modes. Figure 5a illustrates the carbon measure of execution of workloads in each mode. Carbon measure in time slot t reflects the total carbon footprint generated by the execution of workloads from the first time slot to the current time slot t. We observe that when Caspian is in sustainable mode, a significantly lower carbon emission is achieved compared to the baseline mode. We define a performance metric *carbon saving* (CS) as follows,

$$CS = 100 \times \frac{C - C^*}{C}, \tag{9}$$

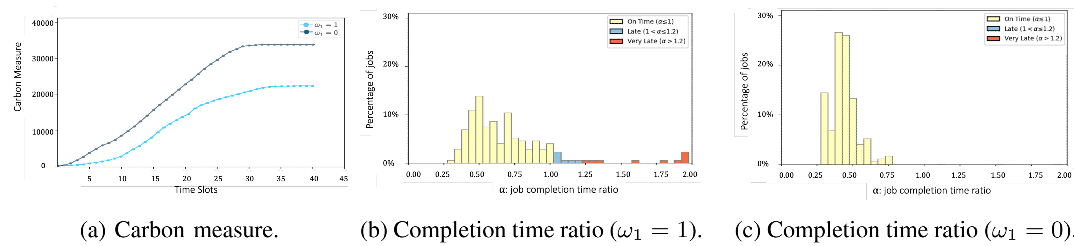
where C and C^* are the carbon measures in baseline and sustainable modes, respectively. In this experiment, we observe that the value of CS is 38.2% indicating that by executing workloads in sustainable mode, Caspian reduces carbon emission by 38.2%.

To evaluate the efficiency of Caspian regarding the QoS of workloads, we define α as the completion time ratio of workloads. The higher value of α indicates the longer response time of workloads compared to their deadline. Let us denote the submission time of workload i by s_i and the completion time by f_i, the completion time ratio of workload i is given by $\alpha = (f_i - s_i)/(d_i - s_i)$. Here, $\alpha \leq 1$ indicates that the workload is completed by its deadline. The greater values of α indicate the lateness of workloads. Figure 5b illustrates a histogram of the value of α for workloads scheduled by Caspian in sustainable mode. We observe that a relatively high percentage of workloads are executed by their deadline and just a fraction of workloads (around 7.2%) miss their deadline. The reason is that, in sustainable mode, Caspian tries to balance a trade off between all objective functions. Thus, to reduce the carbon emission, some workloads may miss their deadline. However, since Caspian also considers QoS, this percentage is pretty low. Figure 5c shows the histogram of the value of α when Caspian is in baseline mode. Since in this mode, the scheduler does not postpone any workload due to sustainability, all workloads complete by their deadline.

In the next set of experiments, we investigate the impact of weight ω_1 on overall performance. We set $\omega_2 = \omega_3 = 1$, and vary the value of ω_1 from zero to two, assigning various weights to the sustainability objective. Here, $\omega_1 = 0$ is the baseline case where Caspian does not incorporate sustainability aspects of clusters in its decision making. With the increase in the value of ω_1, Caspian considers a higher weight for sustainability. Figure 6a shows the percentage of carbon saving obtained due to various values of ω_1. The value of carbon saving is obtained based on Equation (9), in which C is the carbon measure using Caspian in the baseline mode and C^* is the carbon measure when we run Caspian with the given value of ω_1. We observe that by increasing ω_1, a higher value of carbon saving is achieved. For example, for $\omega_1 = 0.25$, carbon saving is 19.4% while for $\omega_1 = 2$, the value is 42.4%. In Figure 6b, the average completion time ratio of all workloads for different values of ω_1 is depicted. We observe that as the value of ω_1 increases, more emphasis is placed on the first objective Z_1, leading to an increase in the completion time of workloads. For instance, for $\omega_1 = 0$, the average completion ratio is 0.49, while for $\omega_1 = 1$, this ratio is 0.72. We also observe a big jump in completion time ratio over the case of $\omega_1 = 1$ and the case of $\omega_1 = 2$. This jump is also observed in Figure 6c where the percentage of workloads that meet their deadline is plotted. We observe that for $\omega_1 = 2$, about 15% of workloads miss their deadline, which is not a negligible value. For $\omega_1 = 0.5$, we get 33.21% carbon reduction, 98.28% meeting deadline, and 0.6 completion ratio. This experiment demonstrates the importance of setting the right values for the preference factors.

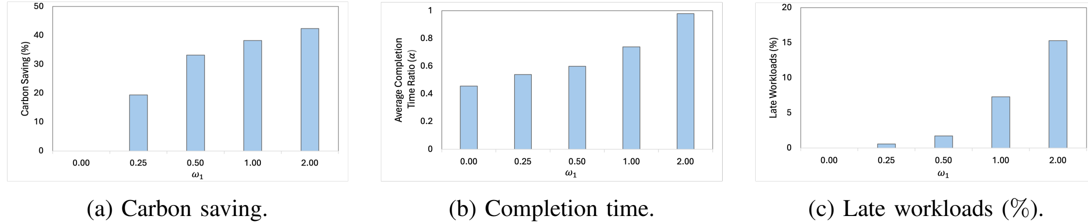
VI. Conclusion and Future Work

In this paper, we proposed Caspian, a carbon-aware batch scheduler in multi-cluster Kubernetes environments, with the aim of minimizing carbon emission due to the execution



(a) Carbon measure. (b) Completion time ratio ($\omega_1 = 1$). (c) Completion time ratio ($\omega_1 = 0$).

Fig. 5: Overall performance of Caspian: sustainable mode vs baseline mode.



(a) Carbon saving. (b) Completion time. (c) Late workloads (%).

Fig. 6: Performance of Caspian for various values of ω_1 .

of workloads, while satisfying Quality of Service (QoS) requirements. We designed a framework for multi-cluster environments in which Caspian interacts with the multi-cluster management platform. Caspian periodically collects information about workloads and clusters, then decides on when and where workloads are executed. Such scheduling and placement decisions are communicated to the multi-cluster manager, which in turn applies those decisions and handles the execution of workloads over the multiple clusters. Employing a running implementation in multiple Kubernetes clusters, we evaluated Caspian using real-world carbon intensity data and workloads with varying characteristics. The experimental results showed that Caspian achieves high carbon reduction compared to a baseline scheduler which only considers the QoS of workloads. In the future, we plan to enhance Caspian by: (i) investigating the potential of considering clusters with heterogeneous processing power, (ii) integrating machine learning algorithms to predict the power consumption of workloads, and (iii) extending the experiments by considering real-word workloads.

References

[1] B. Acun, B. Lee, F. Kazhamiaka, K. Maeng, U. Gupta, M. Chakkaravarthy, D. Brooks, and C.-J. Wu. 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*, pages 118–132, 2023.

[2] T. Bahreini, A. Tantawi, and A. Youssef. An approximation algorithm for minimizing the cloud carbon footprint through workload scheduling. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 522–531, 2022.

[3] T. Bahreini, A. Tantawi, and A. Youssef. A carbon-aware workload dispatcher in cloud computing systems. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 212–218, 2023.

[4] N. Bashir, T. Guo, M. Hajiesmaili, D. Irwin, P. Shenoy, R. Sitaraman, A. Souza, and A. Wierman. Enabling sustainable clouds: The case for virtualizing the energy system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 350–358, 2021.

[5] M. Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005.

[6] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH Computer Architecture News*, 35(2):13–23, 2007.

[7] H. Ferreboeuf, F. Berthoud, P. Bihouix, P. Fabre, D. Kaplan, L. Lefèvre, et al. Lean ict: Towards digital sobriety. *Report for the Think Tank The Shift Project*, 6:16–28, 2019.

[8] Í. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. Greenhadoop: leveraging green energy in data-processing frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 57–70, 2012.

[9] O. Grodzevich and O. Romanko. Normalization and other topics in multi-objective optimization. 2006.

[10] W. A. Hanafy, Q. Liang, N. Bashir, D. Irwin, and P. Shenoy. Carbonscaler: Leveraging cloud workload elasticity for optimizing carbon-efficiency. *SIGMETRICS Perform. Eval. Rev.*, 52(1):49–50, June 2024.

[11] L. Joppa. Made to measure: Sustainability commitment progress and updates. *Microsoft (Blog), July*, 2021.

[12] R. Koningstein. We now do more computing where there's cleaner energy (2021). *URL https://www. blog. google/outreach-initiatives/sustainability/carbon-aware-computing-location*.

[13] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[14] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. V. Vasilakos. Cloud computing: Survey on energy efficiency. *ACM Computing Surveys*, 47(2):1–36, 2014.

[15] A. Radovanovic, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, et al. Carbon-aware computing for datacenters. *arXiv:2106.11750*, 2021.

[16] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. Heats: Heterogeneity-and energy-aware task-based scheduling. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 400–405. IEEE, 2019.

[17] A. Souza, S. Jasoria, B. Chakrabarty, A. Bridgwater, A. Lundberg, F. Skogh, A. Ali-Eldin, D. Irwin, and P. Shenoy. Casper: Carbon-aware scheduling and provisioning for distributed web services. In *Proceedings of the 14th International Green and Sustainable Computing Conference*, IGSC '23, pages 67–73, New York, NY, USA, 2024.

[18] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

[19] Z. Wang, S. Chen, L. Bai, J. Gao, J. Tao, R. R. Bond, and M. D. Mulvenna. Reinforcement learning based task scheduling for environmentally sustainable federated cloud computing. *Journal of Cloud Computing*, 12(1):1–17, 2023.