Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

CO_2 Aware Job Scheduling for Data Centers

Tobias Piontek

Course of Study:	Informatik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Dr. Kawsar Haghshenas
Commenced:	November 1, 2021
Completed:	May 2, 2022

Abstract

Data centers consume large amounts of power around the world. It is estimated that over 1% of global energy consumption is used for powering data centers. Therefore data centers have some potential to reduce global CO_2 output. This thesis therefore introduces a novel practical scheduler implementation for saving CO_2 emissions on a cluster by shifting load in time. Therefore a custom CO_2 power grid efficiency scheduler is developed. The implementation is written for kubernetes, as it is widely used open source cloud orchestration tool. Different architectural solutions to implement a scheduler inside kubernetes are discussed, to find a good approach for realization for this specific cause. The scheduler predicts future CO_2 emissions by using historical data and shifts job in time to CO_2 efficient power grid times. For comparison the implemented scheduler is tested against the default kubernetes scheduling implementation with multiple different scenarios that were built by using real world workload log data. The implementation presented achieved an average CO_2 emission reduction between 0.5% and 2.0%. The scheduler CO_2 reduction is similar to Googles Borg scheduler implementation.

Contents

1 Introduction **17**

2 Kubernetes **19**
2.1 Container . . . 19
2.2 Container Orchestration . . . 20
2.3 Architecture . . . 21
2.4 Kubectl CLI . . . 24
2.5 Minikube . . . 26

3 Scheduler **27**
3.1 Kubernetes Scheduler Reference Implementation . . . 27
3.2 Options to Modify the Kubernetes Scheduling Behavior . . . 28
3.3 Suitable Approach for CO_2 Scheduler Implementation . . . 29
3.4 Scheduler Extender . . . 32

4 Test Methodology **35**
4.1 Kubernetes Benchmark Tools . . . 35
4.2 CO_2 Cluster Benchmark . . . 38
4.3 Suitable Scheduler Testing Approach . . . 39
4.4 Cluster Utilization Logging . . . 39
4.5 Workload Modeling . . . 40

5 State of the Art **41**
5.1 Carbon-Aware Computing for Datacenters . . . 41
5.2 A Low Carbon Kubernetes Scheduler . . . 42
5.3 Let's wait awhile: How temporal workload shifting can reduce carbon emissions in the cloud . . . 44
5.4 Benefits of the CO_2 Scheduler . . . 45

6 CO_2 Emission and Workload Prediction **47**
6.1 CO_2 Emission Prediction . . . 47
6.2 Prediction Accuracy . . . 48
6.3 Workload Forecasting . . . 48

7 Test Implementation **49**
7.1 Generating a Workload Scenario . . . 49
7.2 Executing a Workload Scenario on Cluster . . . 49
7.3 Logging Performance Data . . . 52

8 Scheduler Implementation **53**
8.1 Service Level Agreement . . . 53
8.2 CO_2 Scheduler SLA . . . 55

9 Test Environment **57**
9.1 Hardware Requirements . . . 57
9.2 Software Requirements . . . 57
9.3 Minikube Setup . . . 58

10 Evaluation of Test Run Results **59**
10.1 Benchmark System Specifications . . . 59
10.2 Power Model . . . 60
10.3 Workload Scenarios . . . 60
10.4 Benchmark Runs . . . 62
10.5 Summary of Results . . . 68

11 Conclusion and Outlook **71**

Bibliography **73**

List of Figures

2.1 Comparison VM vs Container 20
2.2 Architecture from Kubernetes Cluster 21
2.3 Pod Infrastructure with two Containers 23

3.1 Scheduler Extender Architecture 32

4.1 Flow Chart of the K-Bench Benchmark 36
4.2 Kubernetes Dashboard Performance Metrics 40

8.1 ER Diagram of Service Level Agreement (SLA) Classification 55

10.1 Utilization against Power Consumption 59
10.2 First Workload Log Analysis 61
10.3 Second Workload Log Analysis 62
10.4 First Scenario 64
10.5 Second Scenario 65
10.6 Third Scenario 66
10.7 Fourth Scenario 67

List of Tables

3.1 Benefits and Drawbacks of Scheduler Implementation Approaches 31

10.1 Workload Scenario 60
10.2 Benchmark Results 69

List of Listings

2.1 Example of Minimal Pod Configuration 23
4.1 Example Configuration of a Sysbench CPU Test YAML File 37
7.1 Benchmark Pod YAML File 52

List of Algorithms

3.1 Kubernetes Scheduler Heap Sort Function 27
5.1 The Low Carbon Kubernetes Scheduler 43
7.1 Workload Analysis 50
7.2 Workload Generation 51
8.1 CO_2 Scheduler 54

Acronyms

API Application Programming Interface. 17

CI/CD Continuous Integration / Continuous Delivery. 24

CICS Carbon-Intelligent Computing System. 41

CIS Center for Internet Security. 35

CLI Command Line Interface. 21

DCP Dynamic Capacity Planning. 42

DSM Demand Side Management. 42

FAAS Function-as-a-Service. 44

FIFO Operating System. 27

GKE Google Kubernetes Engine. 36

GUI Graphical User Interface. 58

IAAS Infrastructure-as-a-Service. 20

k8s kubernetes. 17

MAPE mean absolute percentage error. 48

ML Machine Learning. 44

OCI Open Container Initiative. 19

OS Operating System. 19

PAAS Platform-as-a-Service. 20

RNG Random Number Generator. 38

SLA Service Level Agreement. 7

SLO Service Level Objective. 41

VCC Virtual Capacity Curves. 41

VM Virtual Machine. 19

WMA weighted moving average. 47

WSL Windows Subsystem for Linux. 37

1 Introduction

Kubernetes is one of the largest open source projects in the world. It defines a standard Application Programming Interface (API) for cloud deployments and is available at almost any public cloud provider in the market (p.1 [BBH18]). About 1% of the global energy usage is consumed by data centers(p.1 [RKS+21]). Therefore finding solutions for reducing CO_2 emissions without sacrificing too much service quality could contribute to saving global emissions. This means that implementing an intelligent scheduler that queues jobs on the kubernetes (k8s) cluster at times where power grid efficiency is good can introduce a reduction of CO_2 emission.

Currently there exists some implementation for data centers that try to optimize CO_2 output by shifting jobs in time or in execution environment. Google implemented a proprietary Borg scheduler, that is able to delay jobs in execution time[RKS+21]. Their approach was tested with real business workload, a more detailed description is summarized in Section 5.1. At Bristol students implemented a carbon aware kubernetes scheduler[JS19]. CO_2 is saved by moving existing pods to other regions around the world, where currently CO_2 power grid efficiency is better. The implementation however does not seem to be open source. How their approach worked is explained in detail in Section 5.2. Employees of University of Potsdam and Berlin theoretically evaluated potential reduction of CO_2 emissions by shifting load (p.1 [WBS+21]). They build a model with certain error margins to calculate relative CO_2 optimization potential. Their findings are summarized in Section 5.3.

Googles sophisticated carbon aware scheduler is implemented for their proprietary Borg scheduler and relies on an external API (p.6 [RKS+21]) that provides their CO_2 prediction. The Low Carbon Kubernetes Scheduler shifts jobs in location but not in time. The theoretical evaluation of potential CO_2 reduction lacks a real world data center implementation to prove their claims in a more practical way.

This thesis will describe the process of developing a job scheduler for k8s that reacts to the power grid CO_2 efficiency. The implemented k8s CO_2 scheduler will be tested in its ability to reduce CO_2 emissions.

For implementing the scheduler, multiple scripts have been written to allow the benchmarking of different scheduler implementations. Then an approach was found to predict future CO_2 power grid efficiency. This prediction is then utilized by a k8s scheduler implementation to shift different pods in time. The benchmark scenarios are used to compare the custom k8s scheduler implementation to the default k8s scheduler.

This thesis therefore introduces a new practical implementation approach of a K8s scheduler that predicts CO_2 power grid efficiency and shifts pods to time slots where efficiency is better. The resulting performance numbers in the evaluation gives a resilient efficiency estimation of this approach.

This thesis explains the core components of k8s in Chapter 2 and its scheduler in Chapter 3. Also different approaches for testing are discussed in Chapter 4. The necessary CO_2 and workload prediction is explained in Chapter 6. Then, practical implementation of tests and scheduler code is presented in Chapter 7 and Chapter 8. Practical advice of software requirements the originated code, as well as instructions to setup the environment is shown in Chapter 9.

The evaluation of the k8s CO_2 scheduler showed a potential to save CO_2 emissions by about 0.5% and 2.0% as seen in Chapter 10. These numbers where achieved by four different scenarios that were passed by the k8s CO_2 scheduler and the default k8s default scheduler. The CO_2 scheduler therefore is a viable approach for reducing cluster CO_2 output.

2 Kubernetes

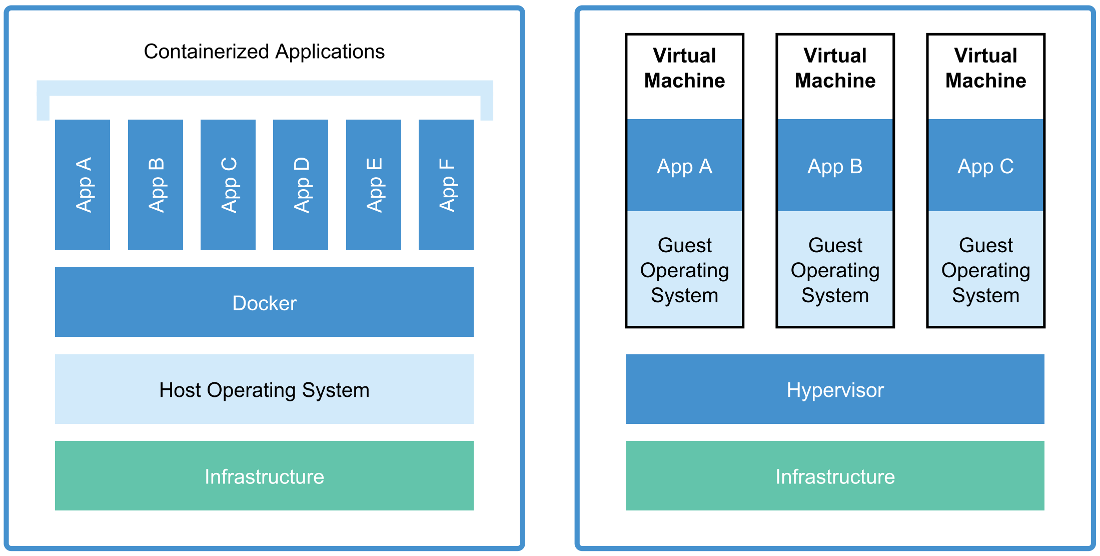
"k8s is an open source orchestrator for deploying containerized applications. It was originally developed by Google, inspired by a decade of experience deploying scalable, reliable systems in containers via application-oriented API (p.1 [BBH18])."

This definition relies on a lot of previous knowledge and is far from self-explanatory. The following sub sections will more deeply explain, what exactly k8s is and why it is used so much. k8s itself is a very complex software, so facts mentioned here are not complete by any means. The different components explained however are needed to understand the ensemble of the k8s scheduler implementation.

2.1 Container

The traditional way of running applications on a cluster has been one single virtual or physical machine that runs all the programs (p.3 [BBH18]). This introduces a major disadvantage. All applications deployed on this specific environment are tightly coupled to the same version of libraries they might share (p. 13[BBH18]). Also runtime failures of certain components can potentially cause other components to fail, that would have performed flawlessly on their own.

Containers are the core technology that enables users to accomplish encapsulation between components and solves dependency management (p.14 [BBH18]). There exist two different kinds of standards for containerized applications Docker and Open Container Initiative (OCI). But since this work relies on docker, explanations will focus on the leader of container formats. A container image is a binary package that encapsulates all of the files necessary to run a program inside of an OS container (p. 14[BBH18])."It is important to know that containerized applications are fundamentally different in structure than a Virtual Machine (VM) which can be observed in the architecture diagrams as in Figure 2.1. Whereas the Container applications in Figure 2.1a share one host operating system and share the Operating System (OS) Kernel[web21]. This results in a smaller size of the container image and fewer VM and operating systems. On the other hand a VM as in Figure 2.1b is hosted by a Hypervisor which coordinates full operating system instances. This introduces a larger boot time and in many cases more resource consumption as a individual host instance is usually larger than its container equivalent. A general advantage for a VM and a container is that these constructs can run in many different kind of environments and are agnostic to it at to least a certain degree (p.43 [Kha17]). This enables users to easily change the environment of their deployments with little to none changes necessary. Container technology itself is a very far-reaching research topic, so this definition should suffice.



(a) Container **(b)** VM

Figure 2.1: Comparison VM vs Container[web21]

2.2 Container Orchestration

"Container orchestration platforms can be broadly defined as a system that provides an enterprise-level framework for integrating and managing containers at scale (p.44 [Kha17])."Some important key criteria for a container orchestration in scope of this thesis are:

cluster state management is the service level that the service provider and customer have agreed (p. 44-45[Kha17])

scheduling is the cloud orchestration tool can schedule different deployments to the appropriate infrastructure(p. 45[Kha17]).

simplifying networking is the network management in most parts is done automatically by the cloud orchestration tool and does not need to be controlled manually(p. 46[Kha17]).

providing monitoring means the different deployments are monitored by the cloud orchestration environment. This includes if provided, health status, resource consumption and up time (p. 47[Kha17]).

Many different Cloud orchestration tools exist on the market varying from googles k8s, Amazon's Elastic Container Service, and many different others (p. 44[Kha17]). While Amazon's Elastic Container Service is a proprietary from amazon, k8s as an open source can be used in many different ways. Amazon, Google, Microsoft, IBM, Docker, Cisco, VMware, Tencent and many more providers offer either managed instances which is essentially Platform-as-a-Service (PAAS) or self managed productive k8s environments Infrastructure-as-a-Service (IAAS)[Kub21c] (p. 28-29[BBH18]). Anybody can host a k8s instance due to the Apache 2.0 license without paying any fee[Kub21d]. For testing purposes, it is also possible to install a less resource intensive version of k8s on a PC using Minikube (p. 29-30[BBH18]). Its main purpose is "[...]local development,

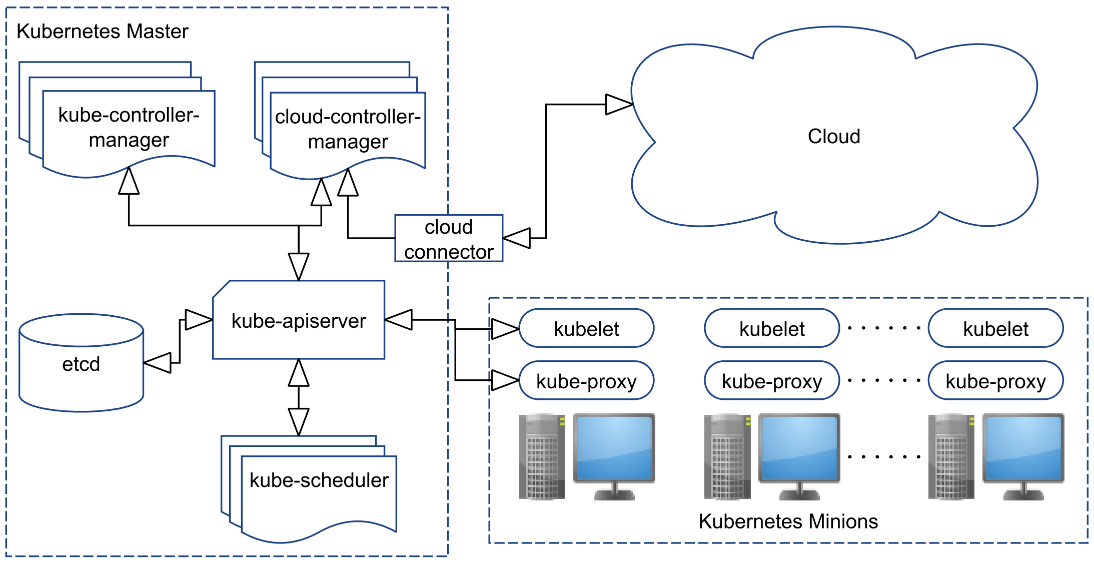


Figure 2.2: Architecture from k8s Cluster[Kub21a]

learning, and experimentation (p. 29[BBH18])". Although disregarding the obvious drawbacks of reduced computational power and no distribution in the network which potentially decreases reliability a Minikube cluster indeed in its feature level and behavior is a full-fledged k8s Cluster operating with bare minimum resource usage.

2.3 Architecture

k8s itself is very versatile in its deployment capabilities. Multiple nodes can be composed to one large k8s Cluster making it very easy managing cross cloud and hybrid cloud deployments. The following describes an architectural overview as in Figure 2.2 of k8s and its main components. Every k8s at least consists out of one **k8s Master** node (p. 32[BBH18]), which can be seen in the dotted box in the left side of Figure 2.2. This master node is responsible for managing all deployments in this distributed platform.

The **kube-apiserver** is in center of the master node Figure 2.2. The kube-apiserver connects components internally and to k8s nodes that are directly attached. The API server also provides the backend for the Command Line Interface (CLI). The kubectl tool accesses the k8s master node to retrieve information of pods, nodes overall cluster state and all metrics that are collected by the cluster (p. 37[BBH18]). As the API and CLI of k8s will be important to this work, they are described in detail in Section 2.4.

In the following a description of the components of Figure 2.2 starting from the top left corner in clockwise rotation of the k8s master will be provided. The **kube-controller-manager** is responsible for the

Node Controller which initializes the other nodes and determines their network addresses[Kub21a].

Route Controller which is an optional component that can be used on Google compute engine clusters to enable communication between Containers on different nodes[Kub21a].

Service Controller which is responsible for generating, deleting and updating services. It also listens to events on the cluster and logs them[Kub21a].

The **cloud-controller-manager** in Figure 2.2 is an optional component to the k8s cluster which enables cloud vendors to easily introduce new code to k8s on a plugin basis without modifying the core code of k8s[Kub21a]. The cloud controller manager does not need to be used, but shows the flexibility of k8s. The cloud controller is connected to the cloud connector which establishes and maintains the connection to a vendor specific k8s cloud implementation[Kub21a].

The **kube-scheduler** will be described in detail in Chapter 3. In general it is consulted by the master node to perform its allocation decisions on the whole cluster the k8s master is responsible for (p.48 [BBH18]). More specific it is responsible for when and where different deployments get deployed.

The **etcd** server is a persistent storage where all API objects are stored. Persistence in general is an important topic in k8s. Entities always need to store information externally, as containers in k8s are completely wiped if they are deleted and rebuild again (p.51 [BBH18]). This will be described in more detail in Section 2.3.1.

The **k8s Minions**, seen in the lower right corner in Figure 2.2 also known as the k8s worker nodes, are responsible for hosting the deployments that get hosted on the cluster (p.32 [BBH18]). Usually, they get work assigned by the kube-apiserver and are accessible through it. Commonly only worker nodes receive deployments, whereas the master node only focuses on managing the cluster (p.32 [BBH18]). The **kubelet** is the node agent of the worker node, responsible for managing and registering the worker node at the kube-api server[Kub21a]. The **kube proxy** is responsible for managing network traffic of the node as well as load balancing it (p.34 [BBH18]). It is also important to note that k8s enables users to utilize multiple different namespaces on the same cluster. For example pods in the same namespace can interact with each other easily by their various service names (p.77 [BBH18]). Namespaces should be seen as another layer of encapsulation enabling different parts of a company to separate their projects from one another.

2.3.1 Pods

"k8s groups multiple containers into a single atomic unit called a Pod (p.46 [BBH18])."A Pod itself at the bare minimum contains one container, but can also contain multiple different containers.

Figure 2.3 shows one Pod that contains the **Web servicing Container** and the **Git Synch Container**. They both share a database that is also contained in their deployment. Because Pods are the atomic unit of a deployment in k8s, all decisions of k8s regarding scheduling, allocation or scaling can not be more granular than the Pod level. This means that entities that should be scaled independently, such as a web service and a database shall never be combined into one pod, but be separated into multiple ones (p.47 [BBH18]). Therefore Figure 2.3 should be considered as an Anti-Pattern, as for instance the **Web servicing Container** can not be scaled independently from the database and vice versa(p.47 [BBH18]). Since Docker has a whale as its company logo the naming "Pod"resembles

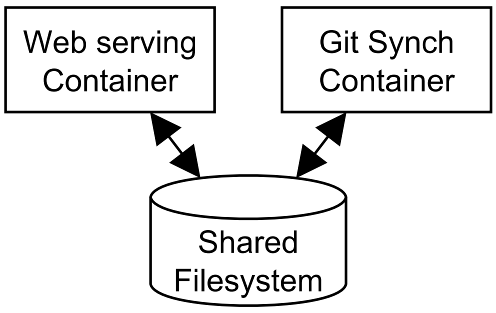


Figure 2.3: Pod Infrastructure with two Containers(p.45[BBH18])

Listing 2.1 Example of Minimal Pod Configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: podname
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
    - containerPort: 80
```

this relation. A "Pod of whales"means a group of multiple whales, so multiple docker containers form a Pod (p.46 [BBH18]). When multiple containers are grouped into one pod, this ensures that:

same time All containers contained in the pod are created together when the pod is scheduled (p.46 [BBH18]). This ensures that all containers start in conjunction not one by one.

same place All containers are queued on the same node (p.46 [BBH18]). As one node is always one physical machine, communication between the different deployments has less lag, high bandwidth and a good reliability.

Listing 2.1 shows an almost bare minimum working pod configuration. **apiVersion** is an internal parameter necessary for k8s.

Pod is the type of the deployment. There exist various other types of entities such as services or replicas.

Metadata name is the only essential field of various other metadata types. This is the name of the created pod which can be important for monitoring or maintenance.

In the **spec** field the different containers are listed. In the example in Listing 2.1 there is only one container named nginx-container, with the docker image nginx. The **container port** field opens port 80 on the container for external access.

It has to be stated, that this example is not optimal for production use, as it is the bare minimum in its configuration. As no namespace is mentioned in the configuration, it will fall back to the default namespace (p.38 [BBH18]).

Containers on k8s also don't get saved in their state once deleted. "It's important to note that when you delete a Pod, any data stored in the containers associated with that Pod will be deleted as well (p.51 [BBH18])."This means that any files that should survive the reboot of a pod need to be saved to a external storage service called **PersistentVolume** on the k8s cluster(p.51 [BBH18]).

2.4 Kubectl CLI

"The kubectl command-line utility is a powerful tool[...](p.37 [BBH18])". Kubectl is locally installed on the machine where it is used for interaction with the k8s cluster. Different namespaces can be directly attached to the kubectl command or be included into the yaml description of the entity deployed. It has to be noted that defining no namespace results in using the default namespace. This thesis uses two different abilities of the API. The k8s CLI is able to create objects, monitor them and also view the deployment on the cluster. The Cluster that wants to be accessed via kubectl can be passed inline with each command. This is not recommended, since the access token is a potential security threat. The recommended way is by using a config yaml file usually located in the .kube folder in the user directory. It contains the server ip address, the client certificate and key storage location or is directly included in the config. These commands can be used manually via console or be embedded into other source code such as bash to. With kubectl CLI complex scenarios can be automated on a k8s cluster or even Continuous Integration / Continuous Delivery (CI/CD) pipelines can be realized.

2.4.1 Viewing and Monitoring Objects

All objects created in k8s can be managed by the kubectl CLI. The most commands follow the schema:

"**kubectl get <resource-name>**(p.38 [BBH18])"

In the following some example commands that are necessary for implementing and debugging basic deployments in k8s are listed:

kubectl get pods lists all pods created in the default namespace(p.38 [BBH18]).

kubectl get pods –all-namespaces lists all pods created on the whole k8s cluster (p.38, 41 [BBH18]).

kubectl describe pod storage-provisioner –namespace kube-system describes the pod named "storage-provisioner"in the "kube-system"namespace (p.117-118[BBH18]). This includes basic information about the deployment, information about its state and a backlog of all events that occurred with it during deployment(p.117-118[BBH18]).

2.4.2 Viewing and Monitoring Resources

The kubectl API offers some basic mechanisms to protocol resource usage:

kubectl top nodes lists cluster nodes and their CPU and Memory usage (p.41 [BBH18]).

kubectl describe nodes is a detailed report of the deployed Pods on the Cluster with their resource usage, events that occurred and basic information about the cluster(p.32 [BBH18]).

The Cluster has 3 different kind of performance metrics that need to be understood and differentiated. All of them can be applied to memory or CPU usage.

The actual resource usage of the k8s node that can be obtained by the **kubectl top nodes**(p.41 [BBH18]).

The requested resource usage acquired by **kubectl describe nodes**. Each container can reserve some CPU time or memory on its k8s node. The scheduler ensures that never more than 100% of the CPU and memory is reserved (p.57 [BBH18]). If the resource consumption of Pods is too high and other services could potentially suffer from it, the scheduler forcefully shuts down services that exceed their reserved usage too much.

The limited CPU usage is also acquired by **kubectl describe nodes**. This describes a hard cap of the resource usage. The k8s cluster will never assign more resources to a container than this (p.58-59 [BBH18]). It also means, that a k8s cluster node can have an accumulated resource limit of over 100%.

2.4.3 Creating Objects

k8s deployments can be created by either using a yaml or json description of them (p.48 [BBH18]). The following commands are most commonly used to deploy for example a pod on the cluster via the CLI:

kubectl apply -f obj.yaml Creates a k8s object based on the specified file (p.49 [BBH18]).

kubectl delete -f obj.yaml Finds the deployment specified in the file on k8s and deletes the deployment on the k8s cluster (p.51 [BBH18]).

kubectl edit <resource-name> <obj-name> Changes a k8s deployment that has been previously deployed on the cluster. This usually triggers a termination of the old deployment and redeploys a new instance according to updated definition (p.39 [BBH18]).

kubectl delete <resource-name> <obj-name> Finds a k8s deployment with the matching name and deletes it(p.51 [BBH18]).

2.5 Minikube

Minikube is "[...]a simple single-node cluster[...] (p.29 [BBH18])". Its main goal is for "[...]development, learning and experimentation[...](p.29 [BBH18]). It should not be used for real world productive environments, as a single node cluster does not take advantage of the upside of a distributed cluster. There exists an offical Minikube tool on github at: https://github.com/kubernetes/minikube that automates installation, setup, starting and stopping or deleting of the minimal cluster setup(p.30 [BBH18]). An advantage is, that Minikube is easy to setup. Multiple different k8s versions can be tested easily, as the version of k8s desired to be run can be attached as a console parameter and the tool will take care of it. However, Minikube is limited as it only runs on a VM(p.29-30 [BBH18]). But running k8s on a regular computer is not meant for real high performing deployments anyway. An alternative to that is running k8s directly in a Docker container which allows using multiple simulated nodes if necessary(p.30 [BBH18]). Minikube is also limited to 110 Pods, as this is the maximum amount of pods that are allowed for one node in k8s[Kub21b]. This is due to ip - address limitations, as every pod gets its own physical address space on a node to be reachable externally[Kub21b].

3 Scheduler

This section describes how the k8s scheduler works and what options exist to implement an alternative k8s scheduler. These different approaches are explained in detail and finally compared against each other. The best suiting approach is than explained in a more detailed way in Section 3.4.

3.1 Kubernetes Scheduler Reference Implementation

"The k8s scheduler is a control plane process which assigns Pods to Nodes"[Aut21b].

The basic implementation of the k8s scheduler first picks a pod from its queue of pods waiting for deployment (p.32 [BBH18]). This priority queue is first sorted by the priority classes of the pods as seen in Algorithm 3.1[Kub21f]. In the default case, no priority is attached to a pod, which results in a default priority value of 0, the higher the value, the more privileged the pod becomes[Doc21]. Pod values roughly can range from zero to one billion, but it has to be noted, that high priority values are reserved for system critical k8s jobs. Using priority values that are too large can potentially decrease cluster reliability[Doc21]. As a second priority of the sort function, the timestamps of the pods are considered[Kub21e]. Oldest pods are favored here in a Operating System (FIFO) queue manner implementation. Once a pod has been picked, the scheduler determines which nodes are possible candidates for a pod according to resource and property constraints[Aut21b]. Property constraints such as annotations for instance can be different labels attached to nodes. These can indicate certain features available there such as a specific availability zone or different hardware requirements. In a second stage the list of possible nodes is then prioritized by a rank function, which in the ideal case returns at least one node. Later the best node is picked and the pod is then bound by the cluster to this node. After this action, the scheduler has finished its job, and the cluster manages the rest of the deployment process.

Algorithm 3.1 k8s Scheduler Heap Sort Function

```
// Less is the function used by the activeQ heap algorithm to sort pods.
// It sorts pods based on their priority. When priorities are equal, it uses
// PodQueueInfo.timestamp.
func (pl *PrioritySort) Less(pInfo1, pInfo2 *framework.QueuedPodInfo) bool {
    p1 := corev1helpers.PodPriority(pInfo1.Pod)
    p2 := corev1helpers.PodPriority(pInfo2.Pod)
    return (p1 > p2) || (p1 == p2 && pInfo1.Timestamp.Before(pInfo2.Timestamp))
}
```

[Kub21f]

3.2 Options to Modify the Kubernetes Scheduling Behavior

In the scope of k8s there currently exist 4 different approaches to change the default scheduling behavior of a k8s cluster[Gui20][Hua20]. This section will give a brief overview about the different approaches possible. Each approach has its different advantages and disadvantages, so it is not possible to rank them simply by good or bad.

custom reference scheduler This approach is using the standard k8s implementation and modifies it to match the desired result [Gui20]. A benefit here is, that potentially there is almost no limitation for implementation. However a large drawback here is, that the k8s scheduler is not a static piece of code and changes from time to time. Changes in the k8s cluster may require major adaptions in the adapted approach or even a complete rewrite. Also this approach enforces the new scheduler being a replacement of the standard scheduler, which is not available anymore, in addition the modified version has now become the default implementation to the cluster[Gui20]. Consequently this is a very tightly integrated but yet not flexible approach.

custom scheduler implementation This requires a full custom implementation of the scheduler[Gui20]. This scheduler is deployed on the cluster as an alternative scheduler. However, not the standard scheduler is replaced, so multiple schedulers are available then[Gui20]. Pods for example can explicitly specify from which scheduler they want to be scheduled. A potential drawback of this implementation style is, that this yet again produces a lot of boiler plate code that again needs to be maintained or becomes obsolete over time. However the worst case in this scenario is, that the scheduler becomes useless with a k8s update and the environment needs to be reconfigured to the default scheduler.

scheduler extender Using the scheduler extender is a rather quick approach in changing the scheduling behaviour[Gui20]. This approach applies a filter to the default scheduler, rather than requiring a complete new implementation of the whole scheduler. The scheduler extender is deployed as a basic rest service communicating over HTTP or HTTPS with the standard scheduler implementation[Gui20]. The standard scheduler receives a configuration file with the connectivity information of the scheduler[Gui20]. Benefits of this approach are, that boiler plate code is reduced to a minimum, as the filters can directly be written in the scheduler extender with very less code needed around it. As the scheduler extender is rather small, code changes can be made very rapidly so fast prototyping for academic purposes. However, as the scheduler extender is realized with a web socket, there is a potential limit in performance, as method call duration over a web socket is potentially higher, than over a local method call[Hua20]. This is no problem with smaller k8s implementations but more for large clusters managing hundreds of nodes with thousands of simultaneous running pods. This should not be a limiting factor in small to medium sized clusters. Another limiting factor is, that the scheduler extender can only extend the the standard scheduler in certain predefined phases, where the standard scheduler sends the request to the extender. This can potentially be circumvented, by yet again implementing changes in the scheduler that call the extender, but this would make the whole approach more complex and bypasses the reason of simplification.

scheduler framework The scheduling framework by k8s is the most recent approach of k8s for extending the scheduler behavior[Hua21]. It basically is trying to eliminate the shortcomings in large cluster scalability of the scheduler extender approach[Hua21]. With the scheduler framework the main code of the scheduler stays untouched. Changes of the scheduler is realized by implementing plugins and linking them to the standard scheduler implementation[Hua21]. This requires however a recompilation of the whole scheduler. A large benefit of this approach compared to rewriting the complete scheduler code is, that in theory, scheduler plugins should be less or ideally not affected of future k8s updates[Hua21]. For a production environment, this approach is more scalable, but rapid prototyping is easier with the scheduler extender.

3.3 Suitable Approach for CO_2 Scheduler Implementation

In Section 3.2 4 different approaches were described for implementing an alternate k8s scheduler. We will evaluate these approaches in terms of the scope of implementation for this thesis. The drawn conclusion here eventually does not cover the use case of a real world large k8s cluster deployment.

Programming Language This category evaluates how tightly coupled the programmer is to the programming languages he can use in every approach.

Over the 4 approaches, implementing an entire new scheduler from scratch is by far the most versatile solution. Potentially even the programming language could be changed here, if using GO as the programming language is not an option. However not sticking to GO as the scheduler language could increase the difficulty of embedding the code into k8s by a large margin. Theoretically, the scheduler extender by itself can also be written in any language, as it is a basic rest webservice[FK21]. However by using the k8s scheduler framework approach or modifying the k8s reference implementation, the programmer is forced to use go[Hua21], which is not essentially a disadvantage but definitely a limitation.

Feature Set Feature set in terms of software means, how versatile the options for implementing the scheduler are by using this approach.

Implementing a new scheduler gives basically all options for implementation with the only limit being the k8s cluster itself. Theoretically the k8s scheduler implementation can be modified to a degree, by which no limits out of the algorithm itself are induced. Implementing the scheduler with the scheduler framework is limited by plugin extension points[Hua21]. The scheduling algorithm itself here is in a core part of the implementation so only plugins can extend the default behavior when they are called. The most reduced approach is the scheduler extender[FK21]. It does not have as many extension points as the scheduling framework has, so it depends on, if the scheduler extender approach is powerful enough for the implementation or not.

Implementation Effort Implementation effort means, how much effort has to be put into the code, to make meaningful changes to the scheduling behavior. The highest initial effort by deploying a new k8s scheduler is by rewriting the whole code from scratch. Every basic functionality of the scheduler has to be implemented before using it without runtime errors is possible, in case not covered scenarios occur. A lot less cumbersome is the approach of alternating the code of the k8s scheduler reference implementation. However both of these approaches are not recommended[Hua20] Using the scheduler framework or the scheduler extender are the faster approaches because in both cases the basic core scheduler code stays untouched and only pieces of external code are attached to it[FK21][Hua21]. This keeps the potential error rate low.

Maintenance Effort Maintenance effort is similar to implementation effort. Consequently writing more lines of code increases the maintenance effort, approaches that need less lines of code are potentially beneficial in future k8s versions. This means that potentially the modified reference k8s scheduler and the rewritten k8s scheduler can be very impractical in real world deployments, as potentially major changes can even cause the need for an entire rewrite of the scheduler, which yet again would be very time intensive[Hua20]. This means that the plugin approach of the scheduling framework and the REST HTTP handler approach of the scheduler extender require a lot less changes in code, once k8s is updated. Both of these approaches rely on continued support from k8s. Since breaking changes occur, at least the work effort that was lost is by a magnitude smaller than with the rewriting approaches, these extender and framework approach are a lot cheaper to maintain[Hua20].

Rapid Prototyping Rapid software prototyping is an iterative software development methodology aimed at improving the analysis, design and development of proposed systems (p.470[LS92]). So the ability of an approach to be highly viable for iterative development in this case means, how fast different versions of the approach can be implemented and then be tested. Rapid prototyping usually contains 5 phases:

Formulating requirements, demonstrating feasibility, reduce risk of system miss-development, communicate ideas and answering questions about system properties (p.470 [LS92]). These criteria is very important in the scope of this thesis, as time is limited. The re-implementation of the scheduler and the modified version of the reference implementation both need more time for each feature to be implemented. Especially a complete custom scheduler needs a lot of effort for each feature, as flawless interaction with the k8s cluster needs to be insured. The scheduler framework streamlines the process of implementing new features, as the plugins can be written and just coupled to the scheduler[Hua21]. However this requires a recompilation of the entire scheduling algorithm and a redeployment of the whole scheduler. The scheduler extender itself is a very small REST application[FK21]. Since it is only extending an existing scheduler, just the scheduler extender needs to be recompiled and redeployed in every iteration[FK21]. This makes the scheduler extender really fast for a rapid prototyping solution.

Potential Performance Performance describes how good the potential real world deployment of this application can perform. With the scheduler rewrite and the modified reference scheduler, potentially the performance result can be extremely good, as code can be optimized to a very deep level. However the scheduler framework itself is written in go which by itself is a very good optimized programming language[Hua21]. The least performing result by a margin should come from the scheduler extender[Hua20]. This is because the method calls from the scheduler to the scheduler extender and vice versa come with a relatively large performance hit in comparison to a tightly coupled local method call. However, this should not be an issue for small to medium deployments, as this performance hit will only come into play once many multiple scheduling calls in a short period of time hit the k8s cluster.

	custom reference scheduler	custom scheduler implementation	scheduler extender	scheduler framework
Programming language	- -	+ +	+ +	- -
Feature set	+ +	+ +	-	+
Implementation effort	- -	- -	+ +	-
Maintenance effort	- -	- -	+ +	+
Rapid prototyping	- -	- -	+ +	-
Potential performance	+ +	+ +	-	+ +

Table 3.1: Benefits and Drawbacks of Scheduler Implementation Approaches with (+ +) Being the Best Possible Value and (- -) Being the Worst Value

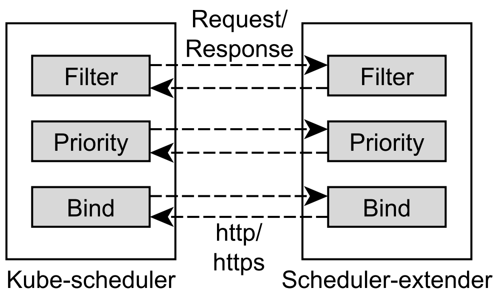
3.3.1 Conclusion of Scheduler Approach

The goal of comparing the different approaches focuses on finding a viable solution for an efficient implementation of the CO_2 scheduler. As the article *Create a custom Kubernetes scheduler*[Hua20] suggests either the custom scheduler or custom scheduler implementation are not recommended due to diminishing returns. As in Table 3.1 the scheduler extender has a lower implementation effort than the other approaches and its lightweight structure simplifies the deployment process. As it is powerful enough to implement a time shift approach for CO_2 emissions, the scheduler extender is the perfect fit for a straight forward implementation of an alternated scheduler in k8s.

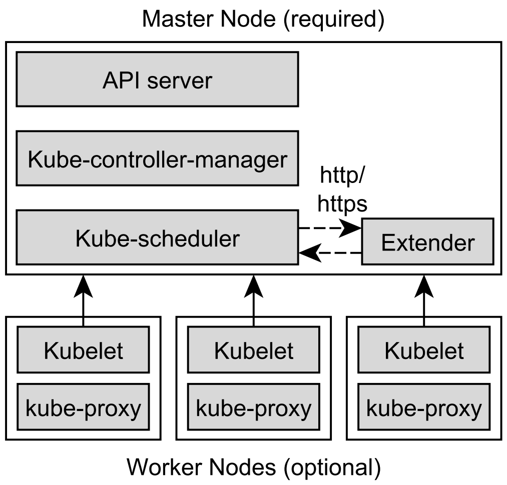
The scheduler extender is the most viable approach currently for implementing an extended scheduler[Hua20]. "The phrase “scheduler extender” simply means configurable webhooks, also known as “filter” and “prioritize”, which corresponds to the two major phases (“Predicates” and “Priorities”) in a scheduling cycle"[Hua20].

The structure of the k8s scheduler that the scheduler extender expands, works in the following way:

- start default scheduler with specific configuration in yaml file.
- standard scheduler watches scheduler API for pods that have an empty spec.nodeName field.
- the first pod of the scheduling queue gets popped and the scheduling cycle is started[Hua20].
  - function for sorting the priority queue, more detailed version with comment seen in Algorithm 3.1: `p1 > p2) || (p1 == p2 \&\& pInfo1.Timestamp.Before(pInfo2.Timestamp ))` [Aut21a].
  - p1 and p2 are priority classes, if they are undefined, which is the default case, the timestamp of the pod is used to order the queue. Oldest pods get queued first.
- hard and soft requirements of the pods are checked [Hua20].
  - hard requirements are CPU or memory requirements of the pod.
  - soft requirements are policies that for example define a class of nodes to run the pod on.
- the scheduler sets the spec.nodeNam attribute of the pod via the API server to indicate that the pod should get deployed here by the main node[Hua20].



(a) Communication between the k8s scheduler and the scheduler extender[FK21]



(b) Architecture of the main node communicating with the scheduler extender to make deployment decisions

Figure 3.1: Scheduler Extender Architecture[FK21]

3.4 Scheduler Extender

In this scheduling process multiple extension points are exposed to the scheduler extender via a REST API[Hua20]. The Architecture of the scheduler extender can be seen in Figure 3.1 The three extension points provided to the scheduler extender framework can be seen in Figure 3.1a. It shows the internal communication between the k8s scheduler and the scheduler extender deployment can be seen. The k8s scheduler and the scheduler extender get deployed in one Pod in the k8s cluster, so they represent one atomic unit only being able to exist with each other in the k8s cluster. A brief description of what the different stages are doing:

Filter The filter stage is used for determining nodes that are feasible for hosting the current pod[Hua21]. If a node is called infeasible in the stage, the later stages are skipped for this node.

Priority In the priority stage, each node that is feasible for the pod is scored. The pod with the highest score is the top candidate[Hua21].

Bind Finally the Bind plugin is called and sends the command to the cluster to bind the pod to a node[Hua21].

Figure 3.1b is a diagram of the deployment of the scheduler extender. In the top the master node is seen with all of its deployments. It has to be noted, that the scheduler extender, as well as the k8s scheduler are both deployed on the master node. This has to be the case, as pods are atomic units in k8s and therefore need to be deployed on the same node(p.46 [BBH18]). Also this effect is desired and necessary in this scenario, as the scheduler communicates with the scheduler extender via HTTPS. The performance penalty for these two components being on different nodes in the same data center or even distributed around the world would be large. Also all worker nodes do ask the master node how to allocate resource in k8s see Figure 3.1b.

4 Test Methodology

This chapter mainly describes the techniques and considerations to develop and implement a benchmarking and logging solution for the k8s CO_2 scheduler. Multiple different benchmarking tools and approaches are discussed as well as a technique to log the k8s usage data to a .csv file. Also some details are mentioned to create a fair benchmark for evaluating the implementation.

4.1 Kubernetes Benchmark Tools

In this section a brief overview of available benchmark tools for the k8s cluster is given. Not all benchmark tools for k8s are designed to measure system performance but instead other properties of the cluster. As the load applied to the cluster is important to evaluate scheduler performance, the design of the load to test the cluster is a crucial part for generating comparisons between different implementations.

4.1.1 CIS-Benchmark

The Center for Internet Security (CIS) benchmark is a k8s benchmark that performs many security checks on a deployed cluster[IBM21]. Settings of the system are reviewed and a score is given to the security level of the cluster. The cluster report shows drawbacks of the configuration and even clues how to fix them. Unlike an usual benchmark no raw system performance is measured but a metric for security is generated.

4.1.2 K-Bench

"K-Bench is a framework to benchmark the control and data plane aspects of a Kubernetes infrastructure[Tan21]." The following describes the flow chart in Figure 4.1 from flow start to flow end starting with **K-Bench config** on top and ending with **infrastructure** at the bottom.

K-Bench config file is a JSON file that contains a flow description of multiple commands that get executed during the benchmark run. The default Benchmark itself is starting creating some pods, deployments and service on the cluster and measures the latency of operations which are performed. The config.json file can be customized in an arbitrary manner[Tan21].

Kube yaml is the authorization file for being able to connect to the k8s API via CLI. The Kube yaml file is automatically generated by Minikube once the cluster is started.

Docker compose files can be optionally provided and converted to k8s spec files with the compose module[Tan21].

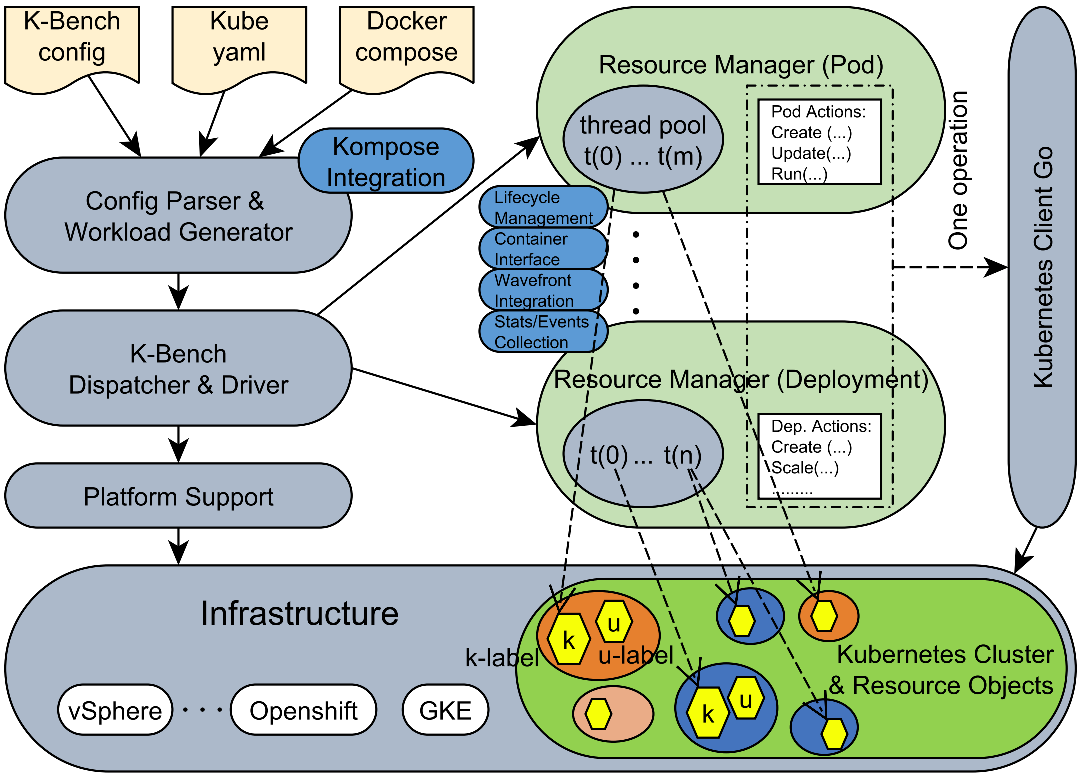


Figure 4.1: Flow Chart of the K-Bench Benchmark[Tan21]

Config Parser & Workload Generator The shell script "install.sh" installs all necessary dependencies and compiles the K-Bench config JSON file. After some modifications the workload generator can be enabled by running the "recompile.shßcript.

K-Bench Dispatcher & Driver is started by running the "run.shßcript. The dispatcher then scans the config file and plans the execution of tasks[Tan21].

Resource Managers are spawned by the dispatcher to manage the different actions for each type of resource. These can be executed in a parallel or sequential order[Tan21].

k8s Client Go The different actions specified are then executed at one operation by their Go Client. Different resources have different Go Clients so their executions can be parallelized[Tan21].

Infrastructure is hosted by vSphere a virtualization platform by VMware[VMw21]. It can manage applications that are hosted on VM and k8s at the same time[VMw21]. Openshift is a tool that provides an uniform developer experience wherever it is deployed[red21]. Openshift can be used to control the whole application life cycle[red21]. Finally Google Kubernetes Engine (GKE) is in this case a Minikube deployment.

k8s Cluster & Resource Objects The k8s Cluster is the hosting application of the containers. The different resource objects get a "k-label"and "u-label"which is either a transaction id or an operation id[Tan21]. Also some meta information about the deployed resources are stored in the label to enable detailed logging[Tan21].

Listing 4.1 Example Configuration of a Sysbench CPU Test YAML File[Kub21i]

```
apiVersion: perf.kubestone.xridge.io/v1alpha1
kind: Sysbench
metadata:
name: sysbench-sample
spec:
image:
name: xridge/sysbench:1.0.17-1
# pullPolicy: IfNotPresent
# pullSecret: null
options: --threads=1 --time=10
testName: cpu
command: run
```

[Kub21i]

4.1.3 Sysbench

"Sysbench is a scriptable multi-threaded benchmark tool based on LuaJIT. It is most frequently used for database benchmarks, but can also be used to create arbitrarily complex workloads that do not involve a database server[Kub21g]."Sysbench can benchmark a variety of parameters on a k8s cluster.

This includes CPU, Memory and Database testing[Kub21g]. The different benchmark loads can be defined in yaml as in Listing 4.1. A minor drawback of Sysbench is, that windows support was dropped with its release version 1.0[Kub21h]. This can be circumvented by using Windows Subsystem for Linux (WSL) by installing it on a Linux based runtime[Kub21g]. The installation process under the major Linux distributions is straight forward, as it is included into the most common package managers[Kub21h].

The scheme of a sysbench start is the following:

sysbench [options]... [testname] [command] Usually most default scripts include a prepare phase, e.g. setting up a database or files that are needed. A run phase to start the benchmark script and a cleanup phase that removes any temporal needed files from the benchmark[Kub21h]. There also exist several sample benchmarks, most of them include some basic launch parameters. Some useful command line parameter are:

–threads The number of threads that are created.

–time Time limit for execution, can be set to 0 for no limit.

–thread-init-timeout Wait time until the worker thread initializes

–warmup-time Execution time of the benchmark script until performance metrics of the task are logged.

For the use case of applying load to the k8s cluster, logging is not the major advantage of Sysbench, however it can still be useful to see if a modification of the scheduler creates changes in reliability and performance of the cluster. To build more realistic and less static benchmark scenarios multiple

Random Number Generator (RNG) can be used with various distributions such as Gaussian or uniform[Kub21g]. The configuration YAML of the Sysbench as for example in Listing 4.1 has an easy to comprehend and brief syntax. A potential drawback could be a limitation in terms of settings which are applied to the different loads. It is definitely equipped well enough for putting a basic load to a k8s cluster.

4.1.4 Custom Benchmark Solution

As mentioned in Section 2.4 k8s provides a CLI that can be accessed via the k8s dashboard, which is a graphical representation of it or via a Console. This CLI can be utilized to write a bash or batch script, that triggers certain scenarios in k8s during a benchmark run. Both bash and batch have the ability to trigger console commands in an arbitrary manner with time delays in between. Potentially this solution could be more or less complex than the other approaches depending on the use case. As the aim for the benchmark is to benchmark for CO_2 efficiency rather than pure cluster performance, standard of the shelf solutions are not specifically tailored to this approach. For a custom benchmark script it would be essential to have some kind of fixed pre-generated scenario, that can be rerun from implementation to implementation, to keep results comparable, reasonable and fair. Another problem is that benchmark frameworks don't come with the option to track the cluster utilization or cluster efficiency.

4.2 CO_2 Cluster Benchmark

When testing performance relevant systems, benchmarks are an important type of artificial performance evaluation. Benchmarks can test all kinds of attributes, most of the times computational power or efficiency is evaluated. In the term of the CO_2 aware job job scheduler the carbon efficiency has to be evaluated. Therefore some load has to be applied to the cluster with the modified scheduling algorithm. This load scenario is then compared against a regular scheduling implementation to measure any potential benefits of the modified implementation. An important part is that "[...]such comparisons are meaningful only if the systems are evaluated under equivalent conditions (p.8 [Fei15])". As the goal is to evaluate CO_2 efficency only, every other aspect of the test setup should be kept static to get comparable results. Following some basic properties that need to be considered for a realistic benchmark scenario:

equivalent hardware Ideally both benchmarks should be run on the same system or at least on a system with similar hardware specifications.

equivalent type of measurement The type of measurement for the performance evaluation should be equivalent or at least similar, as two different types of API could potentially return unequal kind of data. The measurement can potentially stress the system under evaluation too. To decrease any potential side effects at least the same kind of static load should be applied to the system under evaluation.

equivalent workload Finally the exact same kind of workload or at least workload with a similar pattern should be run to keep the results comparable.

reasonable workload pattern To get results that provide realistic results for real world scenarios it has to be ensured, that is the workload is similar to real world load(p.10 [Fei15]). Otherwise an artificial scenario has been proven, but potentially no impact is seen in real world scenarios.

4.3 Suitable Scheduler Testing Approach

The CO_2 benchmark as mentioned before has specific requirements for bench-marking efficiency of the cluster. As CIS Benchmark in Section 4.1.1 is only for checking the security configuration and was mentioned to show variety of k8s benchmarks. Because of this the main comparison is between K-Bench in Section 4.1.2, Sysbench in Section 4.1.3 and a custom benchmark implementation in Section 4.1.4. Sysbench has the major drawback for this project, because its only deployable under a Linux environment but not for windows. Also another major drawback of Sysbench is, that it is designed for benchmarking cluster performance. In theory, complex scenarios can be configured in Sysbench, but it is not really tooled for scheduling performance benchmarking. K-Bench is better in creating custom scenarios for workloads as Sysbench. However K-Bench is still rather complex for creating simple deployments that are needed to test the scheduler in a Minikube setup. As both benchmark frameworks in consideration seemed to be too complex and inflexible for this kind of use case, the decision is to utilize the k8s CLI to implement a small benchmark script. This script makes calls to the kubectl API to automate the queue of different deployments.

4.4 Cluster Utilization Logging

For evaluating k8s cluster utilization during the benchmark run, performance metrics has to be requested from the cluster. The k8s cluster has its own panel in the self hosted k8s dashboard (p.241[BBH18]), which shows current resource usage.

In Figure 4.2 a graphical representation of the basic data of the k8s cluster can be seen. On the top left corner the actual CPU utilization is displayed. On the top right corner the actual memory utilization is displayed. In the bottom section 5 meters are displayed indicating limits and requests. The requests (green circle) show, how many resources have been reserved by deployments on the cluster. The limits (orange circle) show how many resources can in theory be utilized by all deployments concurrently until something is shutting down. The default k8s cluster actually uses the CPU and memory request to find new deployments that fit on a worker node concerning resource utilization. These values can not only be displayed by the k8s dashboard but also retrieved via the kubectl CLI. With the command: **kubectl describe nodes** a text containing resource limits and requests. The single numbers can potentially be retrieved by filtering from the output. With **kubectl top nodes –use-protocol-buffers** the real time resource utilization of CPU and memory can be grabbed from the CLI. With a bash script these data can then be written into a .csv file to be analyzed later.

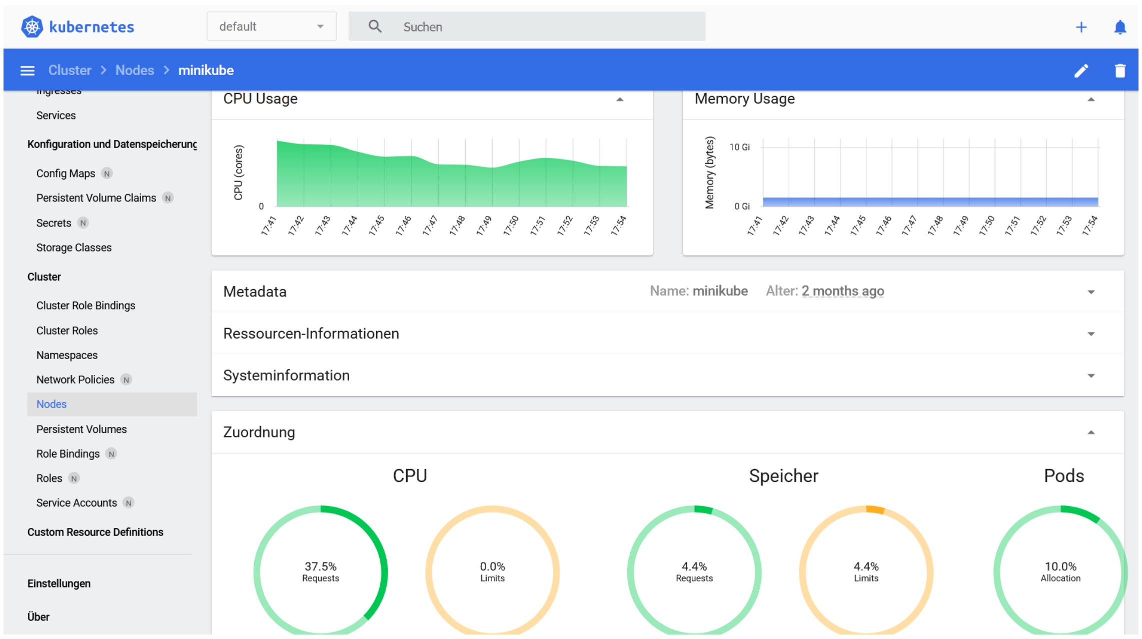


Figure 4.2: k8s Dashboard Performance Metrics

4.5 Workload Modeling

To analyze the performance of the scheduling algorithm some load has to be applied. The implementation of a custom scheduler sometimes relies on metadata to change the scheduling behavior according data provided. The CO_2 aware job scheduler needs labels assigned to the different pods to determine the type of load that has been queued. Therefore a workload has to be modeled that puts some artificial load to the k8s cluster to test the scheduling behavior. [Fei15] describes workload modeling as the following. "Workload modeling is the attempt to create a simple and general model, which can then be used to generate synthetic workloads as needed, possibly with slight (but well controlled!) modifications(p.10[Fei15])."The definition suits very well to this use case, as the scheduler is implemented for scientific research. [Fei15] states further: "The goal is typically to be able to create workloads that can be used in performance evaluation studies, and the synthetic workload is supposed to be similar to those that occur in practice on real systems (p.10 [Fei15])."For the test methodology the workload itself applied to the system should not be the focus. The aim of the workload is to get some general results that should be applicable to real world data center use cases. "It is thus of crucial importance that benchmarks be representative of real needs(p.8 [Fei15])."

5 State of the Art

Three different papers where mainly interesting for the scope of this thesis. Section 5.1 discusses a paper published by Google which presents the proprietary Borg scheduler implementation. Section 5.2 examines a scientific implementation of a k8s scheduler, which shifts load from clusters to different zones around the globe to reduce CO_2 emissions. Section 5.3 discusses a paper with a model based implementation of a load shifting scheduler, which analyzes the theoretical potential of a CO_2 optimized scheduler. Section 5.4 explains the need and benefits of the implementation presented in this thesis.

5.1 Carbon-Aware Computing for Datacenters

Google implemented a carbon aware data center load shifter named Carbon-Intelligent Computing System (CICS)[RKS+21]. The system itself does not use a specific scheduler implementation but a Virtual Capacity Curves (VCC) to constrain maximum load during carbon intense power grid times. The VCC artificially limits the clusters CPU usage by reducing the total amount of calculation power that is available at a specific hour of the day (p.1 [RKS+21]). It is also very important to sustain the reliability of the cluster. Multiple pipelines are arranged in series to calculate an optimal VCC:

Carbon fetching pipeline "which reads hourly average carbon intensity forecasts from tomorrow for each electricity grid zone where Google's data centers reside (p.1 [RKS+21])."

Power models pipeline which trains models that introduce a function to convert computation usage to electricity usage (p.6 [RKS+21]). The main power draw is in many cases implied by CPU load, as data access and GPU operations converges with CPU load.

Load forecasting pipeline that generates a forecast of the expected load of the next day. Therefore the mixture of flexible and inflexible loads is computed and an uncertainty percentage is added (p.6 [RKS+21]).

Optimization pipeline to determine the load curve an optimization pipeline was implemented that predicts the power peaks of the carbon emissions for the next day and plans the VCC according to that (p.6 [RKS+21]).

The system is monitored by a Service Level Objective (SLO) violation detection algorithm. Once the VCC is persistently exceeded the SLO flag is set and the system stops to shape the load of the cluster (p.6 [RKS+21]). The cluster then stops shaping the CPU utilization for a week to retrain the models for load forecasting(p.6 [RKS+21]).

The efficiency gains heavily rely on the load that is applied to the servers and the change of the carbon intensity of the power grid over the day. Therefore two different kind of loads that are classified:

Shapeable workload flexible workload, that does not need to be computed instantly and the computation effort stays the same after delaying

Not shapeable workload Not flexible workload that needs to be performed immediately.

The total results of carbon savings are between 1% to 2% during high carbon hours (p.12 [RKS+21]) but significant, since even a low power reduction on a large cluster can have a big impact. If the CICS was calibrated to aggressively and the VCC was set too low, many tasks were rescheduled at other clusters. This behavior however was not considered in the calculations of the power models pipeline. Comparisons to other schedulers are difficult as the CICS implementation is written for the proprietary Borg system of Google and therefore not open source (p.4 [RKS+21]).

5.2 A Low Carbon Kubernetes Scheduler

Students of the University of Bristol implemented a carbon aware scheduling policy for the k8s container orchestrator (p.1 [JS19]). An extension of the k8s scheduler was written. An important part of Demand Side Management (DSM) in their implementation is distributing load between different locations(p.1 [JS19]). The modified scheduler uses solar radiation as a scheduling metric to reduce carbon emissions, as data centers with high solar radiation are preferred. According to them, following approaches can be used to increase energy efficiency in data centers:

Virtualisation Dynamical provisioning or deprovisioning of resources(p.3 [JS19]).

Server consolidation and encapsulating application Reduces the amount of active servers by consolidating the workload of multiple servers to one(p.3 [JS19]).

Dynamic Capacity Planning (DCP) Adjusts the available resources to the current demand to prevent over or under provisioning(p.3 [JS19]).

Load Balancing Balances the workload among different servers to level out average server utilization(p.3 [JS19]).

Scheduling and VMs placement Places VM into a most energy efficient time slot and location(p.3 [JS19]).

Live migrations Migrates VMs from over-utilized and under-utilized to more efficient servers to consume less energy(p.3 [JS19]).

Renewables Replaces VM from carbon inefficient servers to servers run by renewable energy sources(p.3 [JS19]).

"The carbon intensity is calculated as the sum of the carbon intensity of the various energy sources weighted by the relative production volumes per energy source (p.4 [JS19])". For Europe the Bristol students acquired their data by the European Network of Transmission System Operators for Electricity API that allows them to calculate carbon intensity in real time. Unlike in Section 5.1 no forecasting of the data is done and the scheduler reacts to the current situation.

To calculate if a deployment should be rescheduled from cluster A to cluster B they used the following formula:

$$EC_AI_A > EC_BI_B + ER_B + EN_{AB}I_{AB}$$

Algorithm 5.1 The Low Carbon k8s Scheduler

```
Require: kubectl
Require: cloudproviderCLI
  $P = (x, y)$
  $I_d$
  $greenestregion =$
  for all P do
      getcarbonintensity
      for all P do
          if $I_D = 0$ then
              delete
          end if
          $sort\_by\_carbon\_intensity$
          if $[loc0] \approxeq I[loc0]$ then
              for all P do
                  $sortbyairtemp$
              end for
              return $topregion$
          else
              return $topregion$
          end if
      end for
  end for
  $wait30mins$
```

(p.6 [JS19])

[JS19].

$EC_{clusterindex}$ Compute energy in data center named in an index.

$I_{clusterindex}$ The carbon intensity in the region of the data center in an index.

$ER_{clusterindex}$ The energy consumed for deploying in the index cluster.

The Algorithm 5.1 sorts by carbon intensity and if multiple clusters are similar the one with lowest air temperature is picked. The process is repeated every 30 minutes. Algorithm 5.1 therefore uses the renewable approach described in Section 5.2. Unfortunately this paper does not show any numbers concerning saved CO_2 during their runs(p.8 [JS19]). Their evaluation mainly focuses to prove correctness of their implementation, rather than efficiency. Comparison to their work is therefore difficult, as source code and performance numbers remain unknown.

5.3 Let's wait awhile: How temporal workload shifting can reduce carbon emissions in the cloud

Researchers from Potsdam and Berlin tried to calculate the potential of load shifting by implementing a model in a simulator[WBS+21]. They found that 1% of global energy consumption is used by data centers and this number is expected to grow in the future (p.261 [WBS+21]). In the approach of this thesis total load was not decreased but shifted in time to take energy from the power grid in carbon efficient power cycles (p.260 [WBS+21]). Their take on saving power was achieved only by shifting load in time, but not location (p.263 [WBS+21]). During their research Googles CICS already existed but further research is still necessary, as their code is not publicly available. Most workloads in modern data centers are short running and 90% of these batch jobs are running less than 15 minutes (p.261 [WBS+21]). Examples for short running workloads are:

Function-as-a-Service (FAAS) Cloud execution of small code snippets with a constrained lifetime that are usually executed instantly.

CI/CD Rapid prototyping of code with continuous code roll outs that require contemporary execution.

Nightly backups Have the potential for large carbon savings, as their SLA can allow them to be very flexible their point of execution time.

However only 7% of jobs are long running workloads. These therefore have a very low potential for saving carbon emissions (p.262 [WBS+21]). Spontaneous workloads such as FAAS are hard to predict but nightly backups can be very easy to optimize in a real world scenario(p.262[WBS+21]). Theoretical potential on a cluster can be huge as periodic batch jobs make up about 60% of processing power on clusters at Microsoft. A critical point of the paper is the possibility to interrupt workloads. They assume that many workloads can be interrupted, for example Machine Learning (ML) training's. This may be true for the workload itself, as such calculations are maintained by themselves and do not need any input from outside during execution. However pausing and re enabling workloads on real world clusters is highly uncommon today and therefore not easy to implement. This is because applications might have to be build with this specific use case in mind. A general finding was that a large amount of regenerative energy sources leads to a larger carbon emission saving potential(p.264-265 [WBS+21]). Fossil energy sources usually have a steady output over the day in contrast to solar panels that are highly dependent on sun radiation (p.264-265 [WBS+21]). To calculate the potential of energy shifting the following formula was presented.

$$p(t, W) = C_t - min_{\forall t' \in W} C_{t'}$$

(p.265[JS19]).

W forecast window of carbon intensity data points

t time the specific job is shifted

C_t the not optimized CO_2 emission

$C_{t'}$ the CO_2 emissions of the forecast window

The paper also assumed that some workloads can be shifted on a weekly basis, as carbon emissions are usually the lowest at weekends. The simulations use a simulator program called LEAF and no real world implementation was used to verify real world performance data. A single node was simulated that attempts to shift flexible workloads in time. The model was also quite optimistic as a 5% forecast error in workload prediction seems extremely good for a real world scenario. Results of their theoretical model suggest that potential carbon savings are at around 7.4% up to 33.7%(p.267 [JS19]). Compared to a real world implementation of the google CICS Borg scheduler in Section 5.1 these results have to be taken with a grain of salt. The optimization pipeline of the CICS implementation utilizes far more optimization techniques and only achieves a decrease of about one tenth of the simulated theoretical optimum (p.267[RKS+21]). The results for this large difference are manifold. No resource constraints where considered (resources exceeded by maximum of 43% (p.269 [RKS+21]) and the load forecasting model precision was very optimistic. Technical necessities like un-deploying and redeploying load shifts are unmentioned which can add dramatically to energy consumption if utilized too often.

5.4 Benefits of the CO_2 Scheduler

Googles Borg scheduler implementation in Section 5.1 utilizes multiple different mechanisms to reduce CO_2 emissions. However their implementation itself is closed source and only usable in context of their proprietary BORG cluster manager. Therefore their work can not be taken as a starting point for academic work implementation wise. Section 5.2 is the most similar paper to this thesis. The follow the sun approach does not make its own prediction but rather uses an API to see, where CO_2 is currently cheapest. They also do not shift jobs in time, but instead in location. The implementation of thesis could be seen as a more robust test of the load shifting approach presented in Section 5.3. A key weakness of their potential CO_2 reduction is, that they used a model instead of a real scheduler implementation for calculating a theoretical maximum of CO_2 savings. This thesis could establish a connection to future research, as the implementation as well as the tools provided can be used to compare against a more sophisticated CO_2 scheduling algorithm.

6 CO_2 Emission and Workload Prediction

For the CO_2 scheduler it is essential to have a prediction of the CO_2 emissions for the current day to plan the shifting workload. As shifting the workload creates load spikes on the cluster it is essential to also predict the workload to reserve cluster resources according to the predicted load. It is important to guarantee cluster availability at every time. However also a trade off between potential CO_2 savings and availability of resources has to be made. More available resources to the shiftable workload can increase the amount of CO_2 savings but also decrease the vulnerability of the cluster which will become unresponsive.

6.1 CO_2 Emission Prediction

For the CO_2 prediction, the weighted moving average (WMA) approach was chosen. "Moving Average is one of widely known technical indicator used to predict the future data in time series analysis(p.1 [Han13])". The WMA can be applied to any time series and is still widely used for its "[...] easiness, objectiveness, reliability, and usefulness (p. 1[Han13])Äs the CO_2 data used for this project has similar characteristics from day to day, this approach seems promising to have a good estimation for an average day in in the future. In most real world applications the WMA is used for smoothing out curves for stock market prediction (p.1 [Kli11]). As both the stock market and the CO_2 power grid efficiency subordinate to periodic patterns. The WMA can be used to smoothed out the curve for future predictions (p.1 [Kli11]).

$$WMA = \frac{nP_M + (n-1)P_{(M-1)} + \cdots + 2P_{(M-n+2)} + P_{(M-n+1)}}{n + (n-1) + \cdots + 2 + 1}$$

[Han13]

For the implementation of the CO_2 prediction, whole days where parsed out of the CO_2 file and then calculated with the WMA formula. The data kindly provided by `electricitymap.org/research` contained the CO_2 data for the year 2020 and 2021 for Germany. The CO_2 efficency is measured in $\frac{gCO2eq}{kWh}$[map22]. Values are given for every day of the year in an interval of one hour (24 values per day). The following formula was used:

$$EWMA(d,h)_{21} = \frac{1 * e(d-2,h)_{20} + 2 * e(d-1,h)_{20} + 4 * e(d,h)_{20} + 2 * e(d+1,h)_{20} + 1 * e(d+2,h)_{20}}{10}$$

6.2 Prediction Accuracy

For measuring the accuracy in mean absolute percentage error (MAPE)(p.2[Han13]) the following formula is used:

$$MAPE = \frac{\sum_{t=1}^{n} |\frac{e_t}{X_t}|}{n} \cdot 100$$

[Han13]

n is the number of data, e_t is the forecasting error calculated $e_t = X_t - \ddot{X}_t$

The CO_2 prediction error in MAPE, calculated with the formula shown in Section 6.1, results in the error value **32.7 %** which can be considered as a reasonable forecasting (p.501[MPAB13]). By manual observation, it can be seen that the error of CO_2 prediction can be quite high in terms of extreme values. However results of CO_2 window calculation stay rather precise, as only the relative minimum interval prediction needs to be precise instead of the absolute CO_2 prediction value.

6.3 Workload Forecasting

The utilization prediction of the k8s cluster setup was simulated, as a workload prediction would need to be implemented and learned in a long running cluster. This cluster adapts to the workload applied to the scheduler from time to time. The workload prediction is therefore pre calculated and saved into a .csv file that is passed into the scheduler deployment. Therefore the following formulas are used on the workload scenario as seen in Section 7.1, that has to be created prior to calculating the workload prediction:

$d_p(h)$ is the time interval, at which pods get queued, t_p is the time a pod runs, therefore concurrent pods is defined as $P_c(h) = \frac{t_p(h)}{d_p(h))}$. $c_p(h)$ is the core usage of a pod in mili cores, c_s is the static reserved resources of the k8s cluster and c_{total} are the total available cores to the cluster. The hourly utilization $U(h)$ of the cluster can be calculated with the following formula:

$$U(h) = \frac{P_c(h) * c_p(h) + c_s}{c_{total}}$$

It has to be noted, that this formula looks ahead the job duration $d_p(h)$, but since average pod runtime will not be that long for the benchmark runs, the error of prediction should be negligible.

7 Test Implementation

Test implementation for a k8s cluster is rather complex, in particular with a reasonable workload scenario to a cluster. The Test implementation consists out of three major blocks. First some kind of workload scenario has to be generated and saved in a file to allow multiple runs of the exact same benchmark run to achieve comparable results. Then this workload scenario needs to be executed on the k8s cluster exactly as specified in the generated workload scenario. Lastly, the performance data of the k8s cluster has to be tracked, to allow an analyzation of the different runs with the target to gain knowledge about the performance of the implementation.

7.1 Generating a Workload Scenario

To generate a reasonable workload scenario, some steps had to be taken in consideration. The real code implementation can be seen in the project directory path: **./benchmark_scripts/workload_generator/analyseJobTraceAndGenerateWorkloadPattern.py**. A main problem that had to be solved is, that it is not possible to simply parse a .gwf file by translating the jobs to k8s pods and execute them directly on the cluster. Condition for this is that this would require a server cluster with exact same specifications and software, which is not the case. Also since the scheduler is implemented for a single node cluster, the maximum concurrent pod limit of 110 pods at a time has to be taken into consideration. The basic idea of Algorithm 7.1 is to first analyze a standardized .gwf file. For this, jobs are sorted by hour of day and the average core count, job duration and number of jobs per hour are queued is calculated. In a final step, these values get normalized to be distributed around 1. The second part of the implementation has been split into a separate pseudo code in Algorithm 7.2. In the workload generator algorithm a workload model is defined and the characteristics of the hourly normalized values are then applied to it. This leads to a similar workload pattern of the given .gwf file, compatible to the Minikube single node cluster. It has to be noted, that the specified utilization goal will not be met exactly as all parameters are only accepted as integer values. In practical however the deviance was not noticeable.

7.2 Executing a Workload Scenario on Cluster

After generating a workload scenario, a script needs to execute the workload pattern defined in Section 7.1. The workload execution script is written in bash and can be found in **./benchmark_scripts/workload_generator/workloadGenerator.sh**. The script takes a dummy pod file prepared as in Listing 7.1, changes essential parameters and sends the modified pod definition to the k8s cluster via the kubectl CLI. All parameters have been written in capital letters, to separate elements of the pod from values that need to be replaced. The following list explains the parameters order of appearance in Listing 7.1:

Algorithm 7.1 Workload Analysis

```
Require: .gwf workload file
  duration[24][]
  cores[24][]
  // contains average values per hour, per job
  avg_density[24]
  avg_duration[24]
  avg_cores[24]

  // sort job values by hour
  for all lines in .gwf do
      job_duration,job_start_hour,job_cores = parse relevant .gwf cloumns of line
      duration[job_start_hour].append(job_duration,job)
      avg_density[job_start_hour] ++
      cores[job_start_hour].append(job_cores)
  end for

  //calculate hourly averages
  for hour = 0, hour < 24, hour++ do
      sum_duration = 0
      sum_cores = 0
      for job = 0, i > density[hour], job++ do
          sum_duration = sum_duration + duration[hour][job]
          sum_cores = sum_cores + cores[hour][job]
      end for
      avg_duration[hour] = sum_duration / avg_density[hour]
      avg_cores[hour] = sum_cores / avg_density[hour]
  end for

  // calculate normalized values
  normalized_avg_density[24]
  normalized_avg_duration[24]
  normalized_avg_cores[24]
  for hour = 0, hour > 24 , hour ++ do
      normalized_avg_density[hour] = (avg_density[hour] / sum(avg_density)) * 24
      normalized_avg_duration[hour] = avg_duration[hour] / sum(avg_duration) * 24
      normalized_avg_cores[hour] = avg_cores[hour] / sum( avg_cores) * 24
  end for

  //calculate model with normalized values
  Create
```

Algorithm 7.2 Workload Generation

```
Require: Algorithm 7.1
  //define workload parameters
  total_cores
  system_cores
  average_pod_count
  average_runtime
  critical_job_rate
  utilization_goal

  avg_utilization = utilization_goal - system_cores / total_cores
  cores_available = total_cores - system_core
  avg_job_interval = average_runtime / average_pod_count
  avg_core_per_job = avg_utilization * cores_available / average_pod_count

  //average job is
  object job = (cores, runtime, interval)
  avg_job = (avg_core_per_job, average_runtime, avg_job_interval )

  // use parameters calculated by Algorithm 7.1
  job_list = []
  for hour = 0, hour <24, hour ++ do
      while time_left_in_hour do
          core = avg_core_per_job * normalized_avg_cores[hour]
          runtime = average_runtime * normalized_avg_duration[hour]
          density = avg_job_interval * 1 / normalized_avg_density[hour]
          job_list.add(job(core, runtime, density))
      end while
  end for
  write to .csv file
```

Name_LABEL is the name of the pod, followed by naming pattern testpod[index] with incremental order, as pod names must be unique in k8s.

CRITICAL-LEVEL is used as a naming label and also assigning a priority class to a pod to influence its scheduling queue

SCHEDULER-IMPLEMENTATION can be set in the workload script to select the desired scheduler implementation. It can be selected between "my-scheduler", the CO_2 scheduler implementation or "default-scheduler", the default scheduler k8s is shipped with.

TIMEOUT_DURATION is the time the deployed pod will run

CPU_MILICORES is the resource reservation of the pod during its runtime

Listing 7.1 Benchmark Pod YAML File

```
apiVersion: v1
kind: Pod
metadata:
  name: NAME_LABEL
  namespace: pod-benchmark
  labels:
    realtime: CRITICAL-LEVEL
spec:
  schedulerName: SCHEDULER-IMPLEMENTATION
  containers:
  - name: sleep-container
    image: perl
    command: ["sleep", "TIMEOUT_DURATION"]
    resources:
      requests:
        memory: "8Mi"
        cpu: "CPU_MILICORESm"
  ports:
  - containerPort: 80
priorityClassName: CRITICAL-LEVEL
restartPolicy: Never
```

7.3 Logging Performance Data

For evaluation of benchmark runs, logging the status of the k8s cluster is essential. The k8s dashboard allows users to see all relevant scheduler resource metrics in real time, however the values are only logged and visualized for the last ten minutes. Therefore a bash script needs to be developed, that utilizes the kubectl CLI to grab the performance data from the k8s metrics server over a long period of time. The script itself uses two main kubectl commands named: **kubectl top nodes** and **kubectl describe nodes**. Both commands generate a large textual description of the cluster state, one including the cluster limits, the other the cluster CPU utilization. The relevant values are then extracted by a pipeline to be later used for writing the log file. In addition, CPU limit is calculated more precisely by using the mili core number (precision of $\frac{1}{4000}$ instead of $\frac{1}{100}$). This is to get a more precise measurement of CPU resource reservation for minimizing errors in result evaluation later. The data is then piped into a .csv file to be analyzed later.

8 Scheduler Implementation

The CO_2 scheduler is implemented as a scheduler extender in k8s. For this, an empty scheduler extender dummy implementation licensed under the Apache 2.0 license has been used as a basis and modified to the degree to achieve the desired result[Omu22]. The dummy implementation basically is scheduling every pod on every possible node without applying any kind of filtering. The scheduler extender example has been modified to a large degree. The Code can be mainly seen in the **./predicate.go** file in the project directory.

The pseudo code in Algorithm 8.1 reads in the calculated workload and CO_2 prediction and grabs the current CO_2 limit state via the kubectl CLI.

Then an optimal CO_2 window of fixed size of 6 hours is determined.

If a pod is allowed to get scheduled following requirements are checked:

pod priority class means that if a pod is marked as "critical", it immediately gets queued, if classified as "not-critical"additional conditions are checked.

podage is a parameter that ensures if a pod has been waiting for over 24 hours to be allowed for scheduling. This prevents pods from getting stuck infinitely in the queue. This reduces CO_2 optimization potential, but ensures reliability.

CPU limit and CO_2 window means, that if the current CPU reservation of critical tasks is not exceeded and the time of the day is currently within a CO_2 optimal time window, the task is allowed to run.

last case means if all of the properties do not apply, the pod is delayed further, until it reaches a pod age of 24 hours or it can be executed within a 24 hour time frame.

8.1 Service Level Agreement

The Term SLA has originated multiple definitions by literature, but a good general definition used as a starting point for this thesis is the following:

"A Service Level Agreement is a formal negotiated agreement which helps to identify expectations, clarify responsibilities, and facilitate communication between a service provider and it's customers (p.18[Ber05])."

It has to be noted that the term customer is used as a group of individuals not a single person, as the effort of a SLA would be too high for a small group of stakeholders. In general this focuses about service parameters that represent an agreement between provider and customer based on a metric scale (p.20[Ber05]). In the scope of this thesis a complete SLA would inflate the thesis too much. Also the implementation is based on a modified version of the default k8s scheduler and

Algorithm 8.1 CO_2 Scheduler

```
Require: metrics-server plugin
Require: CO2 prediction file
Require: workload prediction file
  parse workload and prediction file
  j = (podname, priority_class, milicore_Reservation)
  q = list of j
  for all j in q do
      for i = 0, i < 24 - 6, i ++ do
          calculate average CO2 emissions for 7 hour window
          return optimal minimum CO2 window
      end for
      var workload_limit = 0.95 - workload_prediction[hour]*0.1
      cpu_limit = get_from_kubectl
      if priority_class = critical then
          schedule job
      else if podage > 24h then
          schedule job
      else if cpu_limit < workload_limit && inside CO2 window then
          schedule job
      else
          keep job in queue (do not schedule)
      end if
  end for
```

many guarantees are provided by default of the k8s environment. Therefore this SLA definition will in any means not be complete and focus on the more technical definition rather than the economical point of view. Therefore the focus will be on the following point of [Ber05]: "Description of the provided service through the definition of classification numbers which quantify the relevant service properties and service levels[1]"

As seen in Figure 8.1, as service level agreement includes multiple services each of which can again include several quality attributes. In the ideal case, these are quantifiable by classification numbers as inFigure 8.1. Each classification attribute then has an agreed service level and an actual service level.

agreed service level means the service level that the Service provider and customer have agreed on(p.89[Ber05]).

actual service level means the service level that is measured on the real implementation to be verified with the agreed service level indicator(p.89[Ber05]).

These two indicators in combination then form the score of the service level value(p.89[Ber05]).

[1]translated quotation from (p.21 [Ber05])

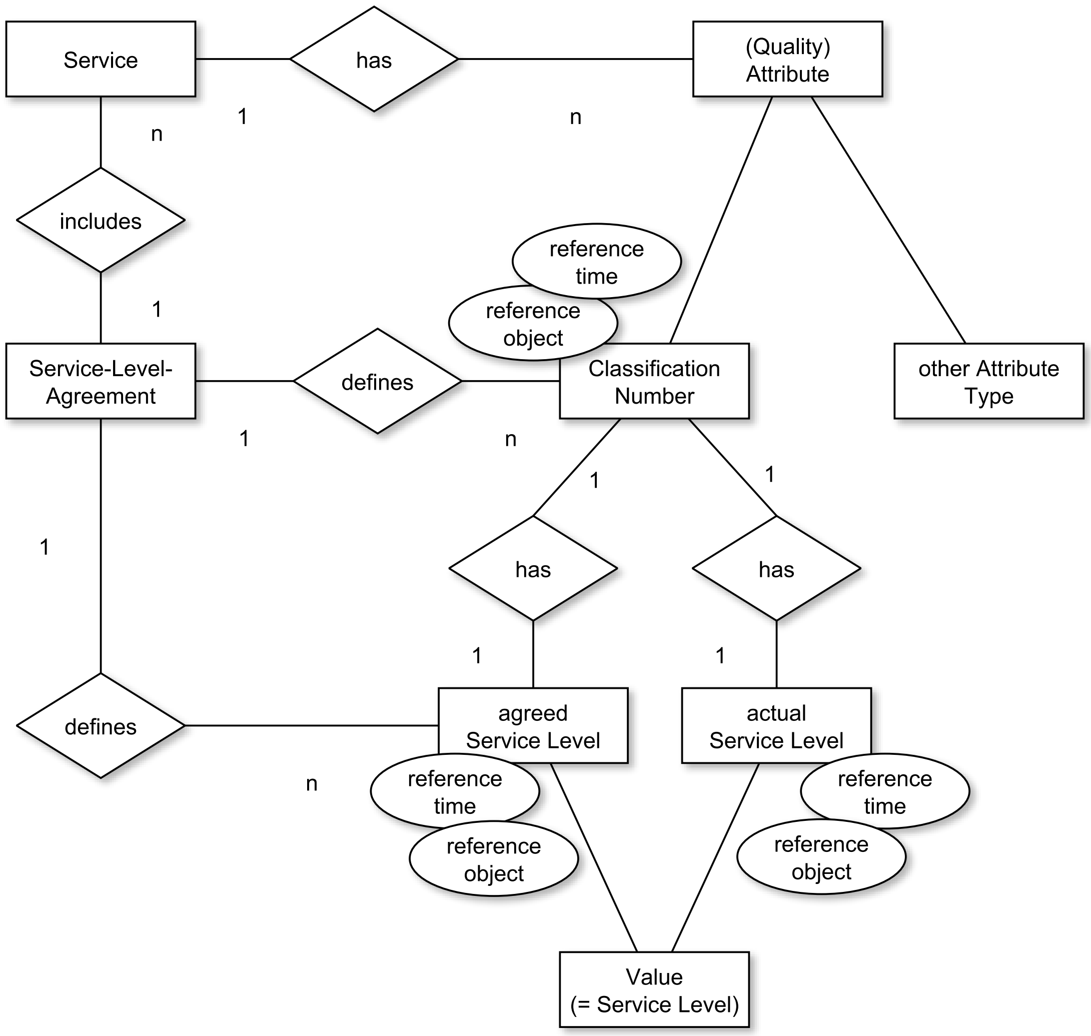


Figure 8.1: ER Diagram of SLA Classification[Ber05] p.27

8.2 CO_2 Scheduler SLA

The CO_2 scheduler is a service deployed in k8s to efficiently schedule k8s pods to reduce absolute CO_2 output of the kuberentes cluster. The main goal is to efficiently shift labeled k8s pods in execution time to time frames with more favorable CO_2 efficiency in the power grid. For achieving this in an acceptable manner, the scheduler ensures, that all jobs scheduled are scheduled within a 24 hour period.

These main goals apply to the scheduler:

- shift workload to CO_2 efficient time window
- ensure that all jobs get scheduled within a 24 hour time limit
- correctly queue pods that are scheduler agnostic

To achieve this, following needs to be done by the user:

- pods that are shiftable in time, are assigned the priority class "not-critical"
  - refer to Test Pods in project directory for examples
- a CO_2 emission log of the last year is provided for the current region
- reasonable workload, that leaves headroom is applied to the cluster
- be compliant with the installation manual at Chapter 9

9 Test Environment

To use the CO_2 scheduler implementation, certain system requirements need to be meet. This chapter gives an overview about what hardware requirements should be meet for running and testing the CO_2 scheduler implementation. Also the Minikube setup for testing is described in detail, as installing Minikube in the correct way and configuration is critical for the environment to run at its full potential.

9.1 Hardware Requirements

According to the Minikube documentation the basic system requirements for running Minikube are at least two CPU cores, 2GB of free memory, a internet connection and 20 GB of free disk space[Aut22].

However these requirements mark a minimum and are not suitable to any use case. During development most of these requirements stayed the same. The memory requirement and disk space requirement are enlarged in comparison to minimum requirements. The hardware and software requirements for running the source code of this project are therefore:

CPU: 2 cores or more

memory: 16GB +

free disk space: 30 GB or more

internet connection

OS: windows 10 pro Build 21H1 or newer/better

9.2 Software Requirements

To be able to run the CO_2 scheduler, some tools need to be installed to be able to compile, install and log the behavior of the cluster.

Git Bash git bash can be downloaded here. It is mandatory for executing bash scripts that are run for logging and executing benchmark runs.

Pyhton 3 Environment for generating performance diagrams and generating workload scenarios. The recent Python version can be downloaded here

Docker for building the scheduler extender container and uploading it to docker hub for k8s installation. Follow this guide for optimal installation.

WSL enabled for performance reasons the installation of docker using WSL is highly recommended.

Minikube a detailed description of the installation is explained in Section 9.3. Version of Minikube is fixed to 1.18.1. It has to be noted, that other versions might also work, but changes in scheduling architecture are not uncommon for k8s.

A Linux based operating system might work, but code has not been tested for this operation.

9.3 Minikube Setup

The description of installing a matching k8s setup mainly follows the *Minikube get started guide* [Aut22]. The prerequisites of successfully installing the Minikube environment are the following programs mentioned as above: git bash, python 3, docker with WSL enabled for performance reasons. The CO_2 scheduler is not implemented for the latest k8s release. The main reason is, that a stable version of k8s had to be picked, as k8s changes versions quite frequently. The version of Minikube, and therefore the requirement of k8s itself is version 1.18.1. This Minikube version can either be downloaded from the release repository or directly following this Link: minikube 1.18.1 installer.exe. Minikube can be launched out of directory directly with a powershell with admin privileges or anywhere, if the minikube.exe is bound as a local variable. For binding the local variable, create a folder named "minikube"directly in the "C:̈directory. rename the downloaded executable file to "minikube.exe". Then the command for binding the path variable in *Minikube get started guide*[Aut22] can be used to bind Minikube. An instance of Minikube can then be simply started in a powershell in the **C:/minikube** directory by executing **minikube start –cpus 4**.

Attention!, it has to be noted, that on first start of Minikube, docker desktop needs to be inactive, as Minikube needs to run directly on WSL instead through the docker daemon for performance reasons(p.30 [BBH18]). This is because docker itself is shipped with its own k8s version, that is performing worse, than the Minikube standalone installation(p.30 [BBH18]). Shutting down the docker deamon ensures, that the setup is performed in the correct way. If warnings of high latency occur while opening the dashboard or enabling the required addon, this is very likely due to a wrong installation on the docker daemon version. Please execute **minikube delete** and try to reinstall as described. Finally the Minikube metrics server needs to be enabled to make the scheduler and logger scripts operative. Enable the metrics server by entering the following command in console **minikube addons enable metrics-server**. A potential problem here might be, that Minikube will state, that the user performing this action does not have enough permissions to do so. Either change the user performing this command, or giving the current user admin privileges for the Windows Hyper-V service. Performing this fix will resolve this issue like mentioned. For manually inspecting cluster state and correctness of the setup, check the log of the **minikube start** command and/or start the k8s dashboard by entering "**minikube dashboard**"to look the current state of the cluster in a Graphical User Interface (GUI).

10 Evaluation of Test Run Results

This chapter describes what the system specifications of the benchmark system where in Section 10.1. All the different containing the Power Model in Section 10.2 and the calculation from CPU utilization to CO_2 savings are explained detailed for the first benchmark run in Section 10.4.1. Results are then shown described in Section 10.5 with a very brief summary of the benchmark runs in Table 10.2.

10.1 Benchmark System Specifications

For benchmarking the test setup was installed on a windows PC with windows 10 pro as its operating system. Windows 10 home or lower will not work, as virtualization is required for the Minikube setup. To keep energy consumption during benchmark runs as low as possible a low power Intel processor was used for running the k8s cluster. Important system specifications were:

CPU: i5 6200U (dual core @2.3GHz, 15 Watt tdp)

RAM: 16 GB

OS: Win 10 pro 21H2 on SSD

Also since the power consumption of the CPU is so low, it can be neglected in the power consumption evaluation later.

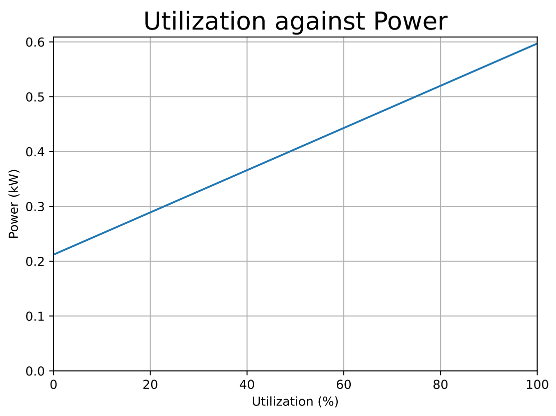


Figure 10.1: Utilization against Power Consumption[FWB07]

	scenario 1	scenario 2	scenario 3	scenario 4
avg. job duration	10 min 54s	32min 45 s	10 min 2 s	5 min 0s
avg. utilization	51.24%	49.81%	44.00%	49.70%
avg. jobs / hour	184.2	60.5	185.2	378.3
uncritical jobs rate	20%	40%	40%	40%

Table 10.1: Workload Scenario

10.2 Power Model

As measuring power consumption directly is time consuming, a conversion from CPU utilization to power consumption via a simple CPU power model is taken. Its converting CPU utilization fairly well, because RAM and CPU utilization are the main dynamic scaling power consumers in PC architecture[FWB07]. Therefore a static baseline of a server can be measured with the device in idle and the power consumption at maximum load. The CPU to power consumption function is a simple linear extrapolation between the baseline and the maximum power consumption resulting in the following formula converting utilization in percent to power consumption in Watt:

$p(u) = p_{base} + (p_{max} - p_{base}) * \frac{u}{100}$

A power estimation for a single server rack of the supercomputer of University of Stuttgart consumes 212 Watt as a baseline and 597 Watt at peak performance. This leads to the following formula:

$p(u) = 212 + (597 - 217)\frac{u}{100}$

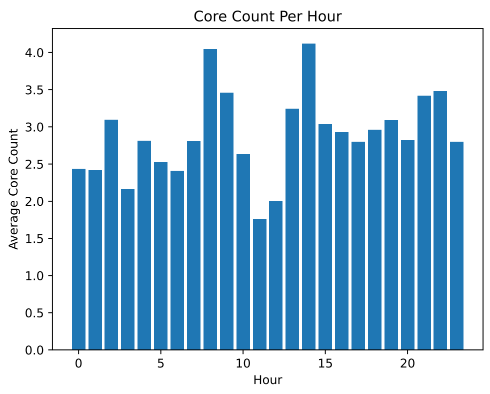
which is shown in Figure 10.1.

10.3 Workload Scenarios

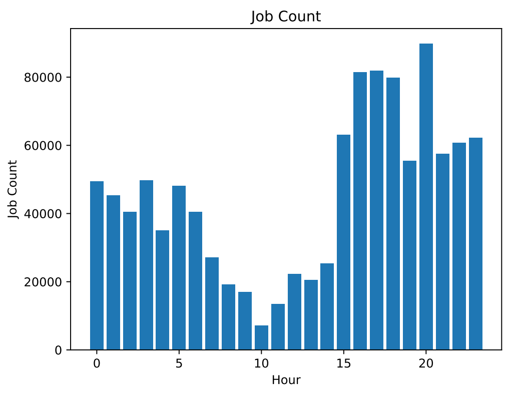
Four different benchmark runs with different workload characteristics where used. Two different .gwf files where analyzed in the process to make the scenarios reasonable. Job execution duration was kept rather short, this is because usually about 90% of batch job workloads run under 15 minutes (p.261 [WBS+21]). Creating too long workloads can also potentially increase difficulty of benchmarking, as the benchmark run can only be stopped, once the full scenario was assigned and finished by the cluster. Scenario 1 and Scenario 2 were based on the workload analysis shown in Section 10.3.1 and Section 10.3.2.

10.3.1 First Workload Characteristic

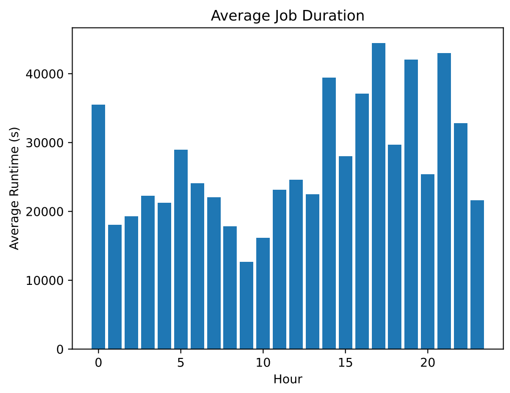
The first workload scenario used the .gwf file provided at [Kub22a]. The file contains a job trace of about one million jobs[Kub22c]. The workload was tracked for about one and a half years[Kub22c]. The average throughput per day was a total of 3000 jobs, with a maximum of 30000 jobs per day. The average job arrival rate excluding intervals with no jobs at all was about 180 jobs / hour[Kub22c]. The overall generated workload pattern as seen in Figure 10.2d shows a workload scenario, that has a rather low utilization from midnight until 14:00. This for example could suit a scenario for



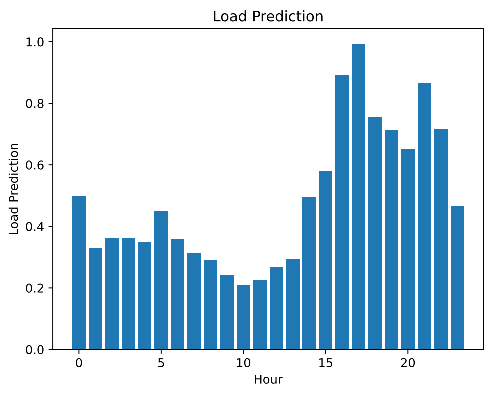
(a) Average core count of every hour



(b) job count of every hour accumulated



(c) Average runtime in seconds on every hour



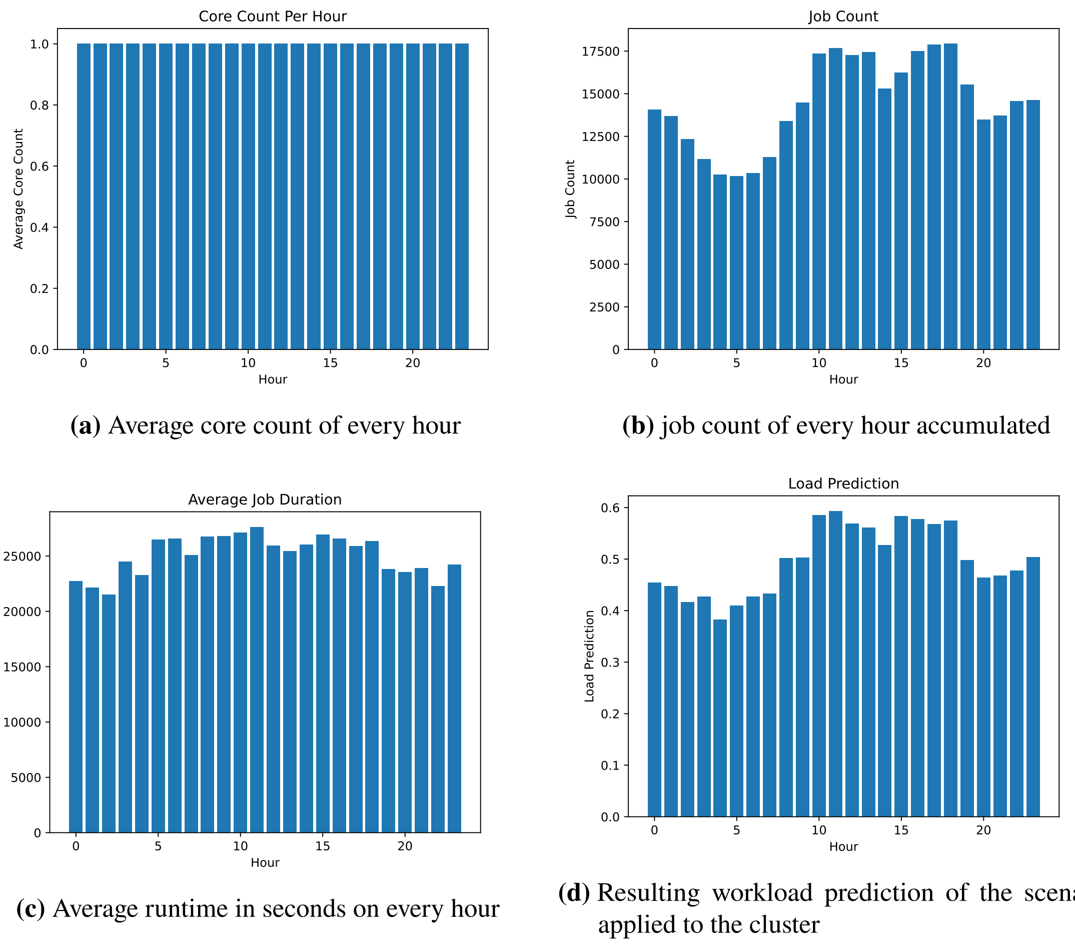
(d) Resulting workload prediction of the scenario applied to the cluster

Figure 10.2: First Workload Log Analysis[Kub22a]

educational purposes, where employees mainly develop their tasks and let them run overnight. Overall, optimization potential for this scenario is good, since cluster utilization is low during noon, where CO_2 efficiency in Germany is pretty good. Also utilization during CO_2 inefficient hours is rather high. Since this data set provided no summary about average cluster utilization, values in a range between 50% and 60% where assumed[Kub22c].

10.3.2 Second Workload Characteristic

The second workload characteristics uses a .gwf workload file provided at [Kub22b]. The statistical values can be seen in Figure 10.3. The data set contains 400.000 entries which were logged during one year[Kub22b]. The cluster used 475 CPU cores[Kub22d]. Average cluster utilization was provided by the online statistics report and was around 58%[Kub22d]. The average job arrival rate was 48 jobs per hour with a maximum of 823 jobs within a hour[Kub22d]. However all jobs seem to have consumed only one processing core, this can be seen in the workload file at [Kub22b] and is also backed up by the online report[Kub22d]. The deviation of the job arrival rate and runtime in Figure 10.3 doesn't have such a high variability as in the scenario seen in Figure 10.2. This results



(a) Average core count of every hour

(b) job count of every hour accumulated

(c) Average runtime in seconds on every hour

(d) Resulting workload prediction of the scenario applied to the cluster

Figure 10.3: Second Workload Log Analysis[Kub22b]

in a workload scenario which has a plain characteristic of CPU demand over the 24% period. Also utilization is rather high during the optimal period which most of the times is between 9:00 to 15:00 during the day. This leads to a scenario for the CO_2 scheduler that potentially has less potential for optimization.

10.4 Benchmark Runs

For analysis of the scheduler four different scenarios have been tested each with the default k8s implementation and with the CO_2 optimized scheduler implementation. The basic parameters of the workload scenarios can be seen in Table 10.1, the workload characteristics are explained in Section 10.3.1 and Section 10.3.2. It has to be considered, that the scenarios don't start at 0:00, this is because the comparison between the default and the optimized scheduler should be fair. The benchmark runs are started, so that the last jobs gets queued at the end of the efficiency time window. This is, so that both schedulers have finished all specified jobs within a 25 hour time period. This would not be necessary if the benchmark scenario would be longer. It needs to be taken into account, that for every scenario, the benchmark had to run 24 hours with each scheduler. The first scenario in Section 10.4.1 is explained in detail, to show how the results are calculated later in a visualized way. The other three are then only focused on the contents of the diagrams others not needed for analyzing are left out.

10.4.1 First Scenario

Figure 10.4a shows the CPU reservation of the k8s cluster. As in Section 10.3.1 the workload is very high at the beginning of the benchmark run at 16:00. As 20% of the jobs are marked as "not-critical"the jobs are shifted in time. Therefore the load before the CO_2 window is reached is on a lower level at the optimized curve (marked green) than the default scheduler (marked red). As the CO_2 optimal window is reached at 9:00, the CO_2 efficient scheduler spikes to a high utilization, but keeps its threshold to hold the cluster responsible until the workload debt is processed. Once this point is reached, both schedulers behave very similar.

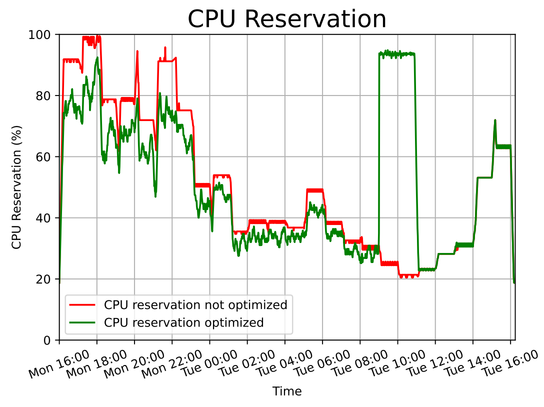
The function in Figure 10.4a is then multiplied by the power model in Figure 10.1. The resulting curve, as in Figure 10.4b, then shows an assumption about the power used by the cluster during the benchmark run. This function is the same, as in Figure 10.4a. This is because the CPU reservation function in Figure 10.4a is shifted on the y-axis by a fixed constant because of the application of the power model to it.

Figure 10.4c visualizes the core part of the CO_2 scheduler. The magenta curve shows the CO_2 prediction of 2021 that the scheduling algorithm utilizes to calculate its CO_2 window. This prediction is calculated by using CO_2 data of Germany for the year 2020 The cyan curve shows the real CO_2 data for the year 2021. For achieving a good effect on saved CO_2 emissions it is not necessary that both curves match exactly, but that they would forecast the same 7 hour efficiency time period. As seen in Figure 10.4c, both curves have their maximum efficiency period very similar, so saved CO_2 potential should be pretty high in this scenario.

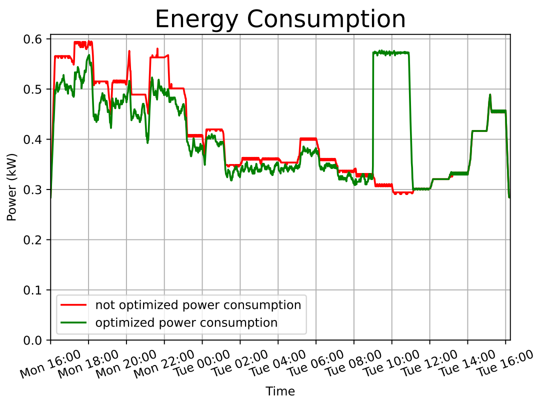
Figure 10.4d shows the CO_2 output of the cluster per hour. It is the respective curve of Figure 10.4b with same color multiplied by the cyan curve shown in Figure 10.4c. The maximum CO_2 output is not performed during the CO_2 efficiency window, although the cluster is at 95% utilization at this period of time. This is because the CO_2 efficiency function is compressed during the CO_2 efficiency window. If the CO_2 emission looks small on the CO_2 emission diagram, this is another good indication for a successful CO_2 emission optimization, as the high power utilization is canceled out by good CO_2 efficiency.

Figure 10.4e can be understood as the integral of Figure 10.4d. It shows the total CO_2 emitted by both runs. It can be observed that the difference between both implementations grows until the optimal window has been reached. As the shifted workload then needs to be catched up, the CO_2 advantage decreases, until the shifted workload has been processed. As the saving potential of CO_2 is rather small by shifting workload as seen in the other scenarios the difference is hard to be visualized with this view.

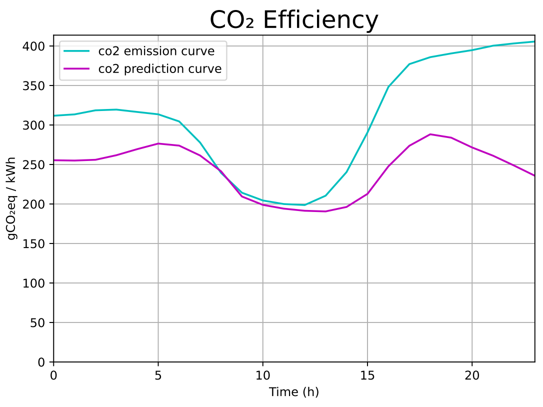
Figure 10.4f is the final diagram, that shows how much CO_2 is saved over time. Essentially it is the subtraction of both curves in Figure 10.4e. You can see the mod CO_2 is saved, when CO_2 efficiency in the power grid is at its lowest between 16:00 and 0:00 as seen in the magenta curve in Figure 10.4c. CO_2 is saved until the window is reached. The steep descent indicates the catch



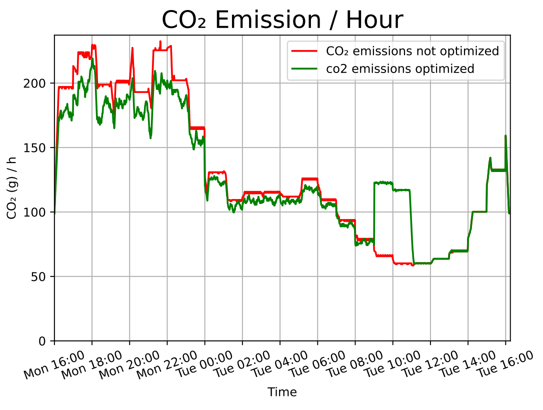
(a) Cluster CPU Reservation



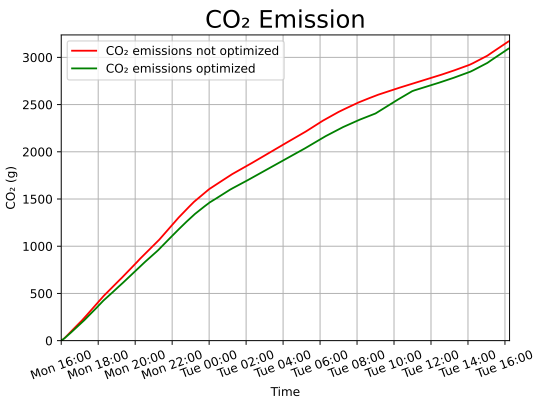
(b) Cluster Power Consumption



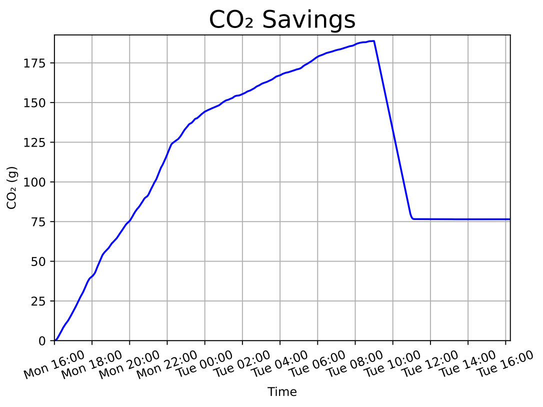
(c) Cluster CO_2 Prediction vs. Actual CO_2



(d) CO_2 Emission Rate

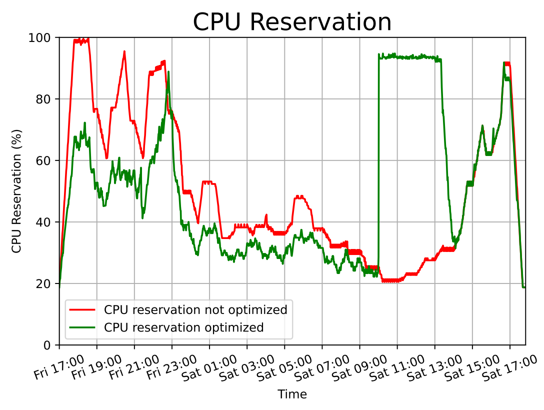


(e) Total CO_2 Emissions

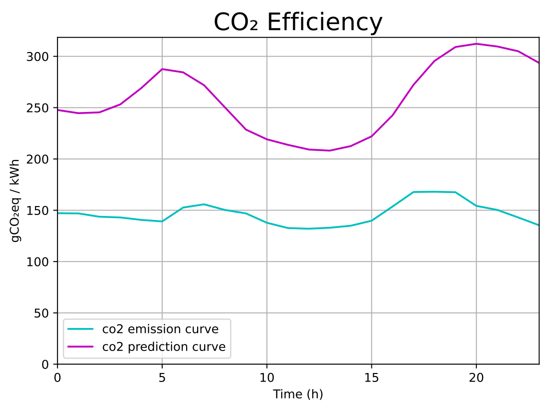


(f) Saved CO_2 Emissions

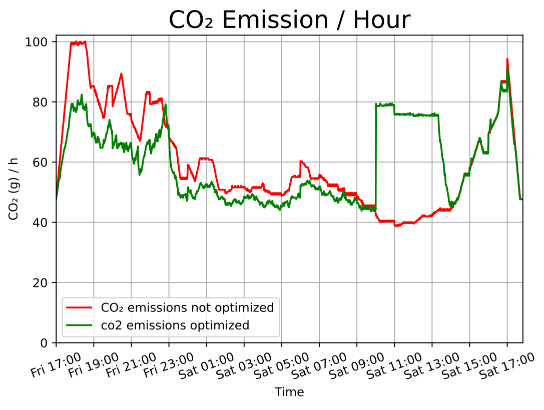
Figure 10.4: First Scenario



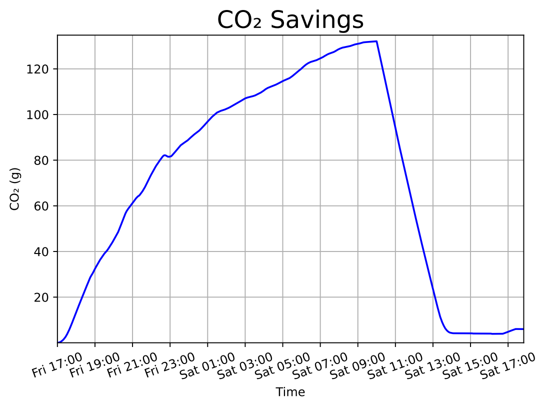
(a) Cluster CPU Reservation



(b) Cluster CO_2 Prediction VS. Actual CO_2



(c) CO_2 Emission Rate



(d) Saved CO_2 Emissions

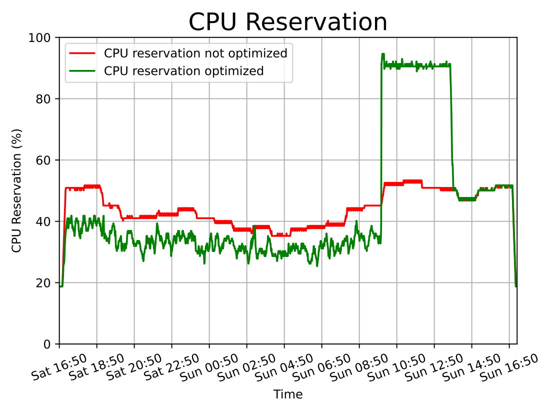
Figure 10.5: Second Scenario

up phase of the optimized scheduler. Once the shifted workload has been processed, the saved CO_2 essentially becomes a constant, as both schedulers now immediately queue jobs once they are queued. The saved CO_2 in gram can therefore be read on the y-axis at the last time stamp of Figure 10.4f which is 76.48 g. This value divided by the maximum of the red curve in Figure 10.4e then gives the percentage of saved CO_2 which is 2.4%. The summary of all results can be seen in Table 10.2 at the end of the benchmark evaluation.

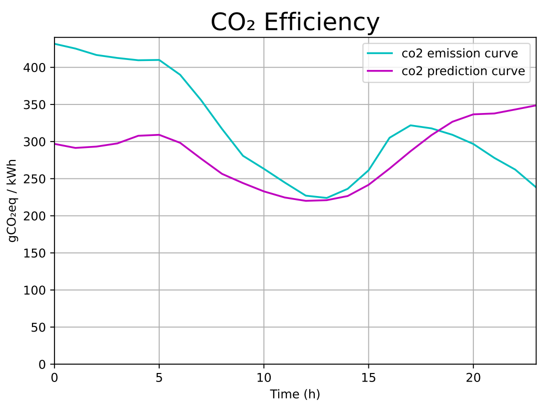
10.4.2 Second Scenario

Figure 10.5a indicated a larger discrepancy between both implementations, this is because scenario 2 has double the amount of not-critical jobs as scenario 1. Therefore the catch up phase of the optimized scheduler takes longer, as more workload can be shifted. CO_2 savings will be poor in this scenario as the variation of the CO_2 efficiency is very low as seen in Figure 10.5b.

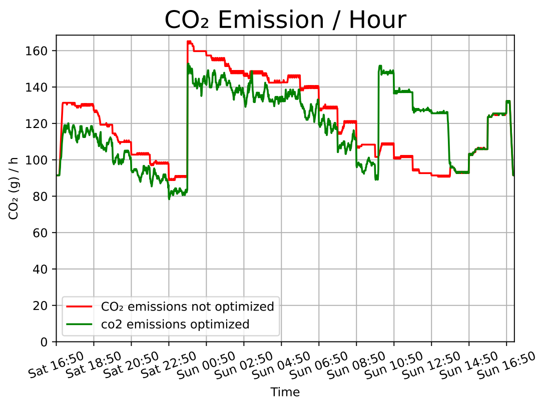
Figure 10.5b shows, that the prediction is very optimistic about the variance of the CO_2 efficiency. However the minimal turning point of both functions is very similar, although their absolute values vary by a large degree.



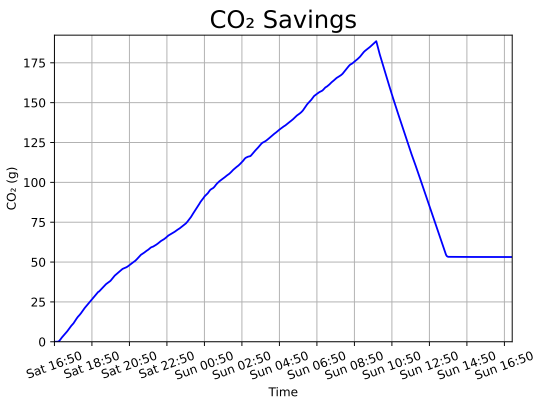
(a) Cluster CPU Reservation



(b) Cluster CO_2 Prediction VS. Actual CO_2



(c) CO_2 Emission Rate



(d) Saved CO_2 Emissions

Figure 10.6: Third Scenario

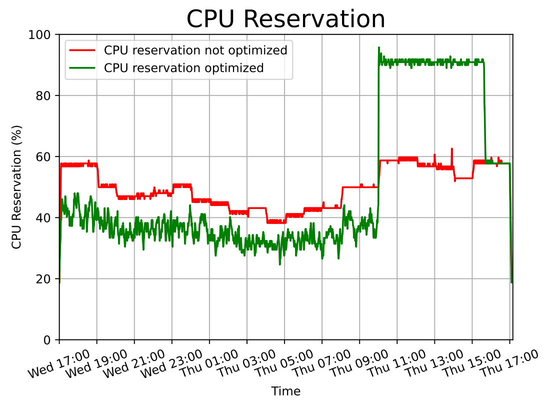
Figure 10.5c suggests that a lot of CO_2 was saved at the beginning of the scenario, as the rather flat CO_2 output during the CO_2 window starting at 10:00 looks promising.

Figure 10.5d indicates, that almost no CO_2 could be saved in the interval. A total of 6 gram was saved which is 0.4%. It is however expected that the optimization potential is small with such a low deviation throughout the day. In this scenario a scheduler implementation that shifts load between different locations around the globe could potentially provide a larger benefit.

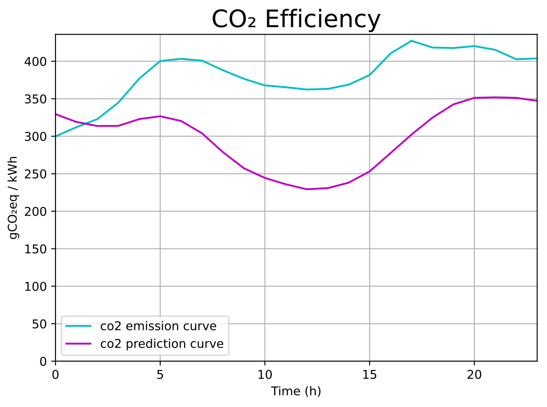
10.4.3 Third Scenario

The scenario in Figure 10.6 uses the second workload characteristic shown in Figure 10.3.

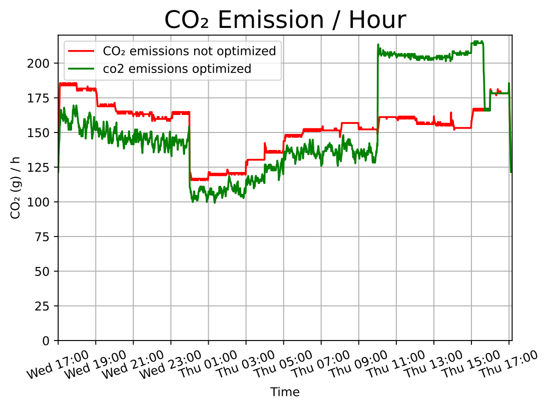
Figure 10.6a is a much more smooth utilization curve than the respective ones in Figure 10.4a and Figure 10.5a. As the uncritical job rate is 40% as in Section 10.4.2, the used CO_2 window is rather large.



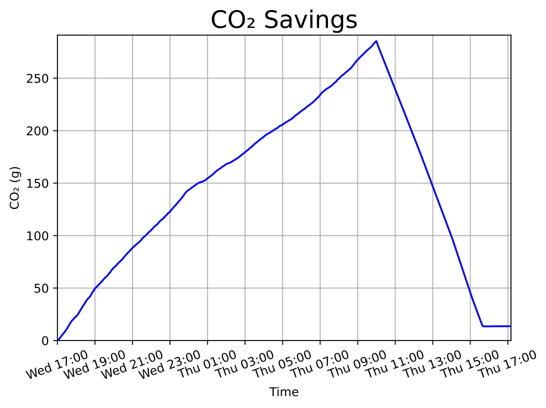
(a) Cluster CPU Reservation



(b) Cluster CO_2 Prediction VS. Actual CO_2



(c) CO2 Emission Rate



(d) Saved CO_2 Emissions

Figure 10.7: Fourth Scenario

Figure 10.6b show, that the overlap of a potential optimal CO_2 window is pretty similar, although the prediction might tend to queue jobs a little bit early here. Therefore the CO_2 prediction this time is pretty good.

Figure 10.6c proves that workload has been queued too early during this run, as CO_2 emission is pretty high during the opening of the CO_2 window. Although the cluster produces less CO_2 per hour at 9:00 with full utilization than at 22:00 with about 50% utilization.

Figure 10.6d shows that the relative CO_2 saved is pretty good in this scenario. The rather large amount of not critical workload, the high variance of the CO_2 emissions and the pretty accurate prediction contributes to this. 53.16 gram of CO_2 could be saved in this run or 1.81% of the total emission value.

10.4.4 Fourth Scenario

Figure 10.7 shows the fourth and last scenario. It considers multiple different parameters in comparison to Figure 10.6. Many short running jobs serve the purpose to put the scheduler under stress. Also average utilization was increased by 5% to extend the use of the CO_2 window. Multiple effects in theory increase the window here. A higher base load decreases the headroom to shift load from CO_2 inefficient hours to efficient hours. Also the pre calculated workload prediction leads to a lower threshold for not critical jobs, yet again decreasing the amount of computational power for not critical jobs.

Figure 10.7a is very similar in workload characteristic to Figure 10.6a. But as the increase suggests overall cluster utilization is higher.

Figure 10.7b shows a sub optimal example for a CO_2 prediction. Although the curves might look similar, the prediction suggests a CO_2 window placed around noon. Compared to that for the real CO_2 data, queuing jobs starting from midnight at 0:00 would have been the best solution. However shifting load from the spike that occurs at 17:00 is still possible with this prediction.

Figure 10.7c is the only CO_2 emission rate diagram out of the four scenarios, where the peak CO_2 output occurs during the CO_2 efficiency window. Essential this does not directly conclude, that the CO_2 emission has increased but can potentially suggest that the prediction might have not been on point.

Figure 10.7d shows that the amount of saved CO_2 is really small. A total of 13.64g of CO_2 was saved within the time interval or 0.37%. With the prediction being this much off, the result seems surprisingly good.

10.5 Summary of Results

Although many different parameters were chosen for the benchmark run as in Table 10.1, CO_2 savings were between about 0.4% and 2.4%. The scheduler is able to work with jobs with an average duration between 5 minutes or 30 minutes. Also the the implementation is performing well enough to be deployed on large cluster. The implementation is able to manage job arrival rates similar to the real workload logs used for creating the scenarios. For maximizing shifting potential about 50% of the total workload needs to be marked as not critical to have the maximum potential for this kind of CO_2 saving technique. In the worst case, a job is 24 hours long which would make this approach useless. Since most not critical workloads are rather short running and frequent batch jobs(p.262 [WBS+21]), the parameters set for these benchmarks can be considered reasonable. For the implementation to work efficient, the job duration of the shifted jobs should not be too long.

In real world applications results for CO_2 reduction between 0.5% and 2.0% should be be realistic. Compared to the BORG scheduler presented in Section 5.1 this result is pretty good, as their implementation achieved CO_2 savings in a range between 1% and 2%. However the benchmark used here might be too optimistic, as the BORG scheduler implementation shifted jobs in time and in location to achieve such good results. Because of this the numbers of the performance analysis need to be taken with care. Although the benchmark scenario has been design to be as realistic as possible, it is not a replacement for performance data that was acquired in everyday use as in the Google implementation in Section 5.1. To achieve larger CO_2 savings, shifting workload between different locations and in time could increase CO_2 reduction at locations where CO_2 variance is rather low within a day.

	scenario 1	scenario 2	scenario 3	scenario 4
CO_2 window start	9:00	10:00	9:00	10:00
CO_2 window end	16:00	17:00	16:00	17:00
total CO_2 not optimized	3174.22 g	1489.48 g	2939.83 g	3725.28 g
total CO_2 optimized	3097.74 g	1483.48 g	2886.67 g	3711.64 g
absolute CO_2 reduction	76.48 g	6.00 g	53.16 g	13.64 g
relative CO_2 reduction	2.41 %	0.40 %	1.81 %	0.37 %

Table 10.2: Benchmark Results

11 Conclusion and Outlook

This thesis investigated the question how CO_2 can be saved on a k8s cluster by implementing an intelligent carbon aware scheduler. To implement the scheduler it is essential to understand basic architectural concepts of k8s.

Therefore in Chapter 2 fundamental knowledge of k8s was explained. It should be highlighted that the k8s kubectl CLI as well as the Minikube test environment were essential for implementing, testing and benchmarking.

Four different approaches of implementing a scheduler in k8s were discussed in Chapter 3. The scheduler extender approach was finally picked as the optimal solution in the scope of this thesis, as it is fast to deploy, has a sufficient feature set and is sufficient in its architectural performance.

An important preparatory step for implementation was the development of a testing setup suitable for testing a k8s scheduler. Chapter 4 discussed multiple different approaches for test implementations, while Chapter 7 shows the practical implementation in code. As the overhead of existing k8s benchmark tools is large, a specialized workload scenario generator program, a workload execution script as well as a logging tool were needed to test and validate the correct functionality of the implementation. The emerging tools for testing the scheduler implementation will be for future implementations, as creating and executing these was a rather complex part in implementing the scheduler. The tool itself can create a high variety of different scenarios for execution and gives the possibility to show good and bad use cases of different approaches.

Chapter 6 discussed the core part of the scheduler for saving CO_2 emissions. WMA was used as the time series forecasting approach. It is wide spread, easy to understand and able to forecast CO_2 emissions to a certain degree in the use cases of this thesis. Chapter 8 showed the practical implementation which utilizes logged CO_2 data provided by `electricitymap.org`.

For full transparency about the concluding numbers, Section 10.1 introduces the four different benchmark scenarios that were passed through the custom CO_2 scheduler implementation of this thesis and the default scheduler of k8s. Essentially different parameters for average job duration, concurrently running jobs an average process utilization and day, at which the benchmark was performed were set. This gave a broad understanding about the practical variance and performance uplift in terms of CO_2 saving of this scheduler implementation.

The analysis of the scheduler suggests that in average during deployment CO_2 savings between 0.5% and 2.0% can be expected. This can be considered as a good result, since the BORG scheduler presented in Section 5.1 produces results between 1% and 2% in real world scenarios. Efficiency of the load shifting approach showcased depends on the accuracy of the CO_2 window prediction, as well as the type of workload that is put on the cluster. The implementation heavily favors days at which a high variance of CO_2 efficiency is taking place within a 24 hour frame. As the scheduler is only responsible for starting jobs but not freezing or undeploying them, short running not critical workloads give a good granularity to keep the workload shaping approach precise. Especially long running workloads (>24 hours) would not show any effect.

The implementation presented this thesis is efficient, if the CO_2 emission has a high variance throughout the day and enough workload is labeled as not critical. The scheduling implementation needs a good amount (around 50% of total load) of not critical jobs to have a sufficient enough potential of modifying the load curve of the CPU, which essentially manipulates the power draw of the entire cluster.

Overall the CO_2 scheduler is successful in saving CO_2 in all benchmark scenarios provided without putting the reliability of the cluster in danger. All workloads were performed in the same interval by both clusters and the CO_2 scheduler was successful in keeping the cluster responsive during peak load when CO_2 efficiency in the power grid was best.

Outlook

Multiple approaches for extending or rewriting a new scheduler are possible to enhance CO_2 efficiency. A more advanced ML algorithm could be used to improve the precision of the CO_2 prediction. A better prediction of the CO_2 emission ultimately leads to a better prediction of the CO_2 window which enhances the average amount of CO_2 saved, as wrong predictions become more sparse.

As an additional mechanism for saving CO_2, the scheduler code can be extended to not only for shifting workload in time but in node location as well. In scenarios where CO_2 variation throughout the day is rather low, this could increase the potential optimization outcome. Shifting load however also introduces a new magnitude of complexity, since some workloads may be unable to shift in a location because of data protection laws and/or environmental requirements that need to be met by the respective node.

Finally a true real world implementation of the scheduler in a scientific or productive industrial environment could be insightful about potential weaknesses in everyday usage of the CO_2 scheduler. This would be necessary to validate the findings of this thesis, as a theoretical approach can only strive to a real world use case, but never replace it in its significance to the effectiveness. To keep users of the scheduler motivated a financial incentive for marking jobs as being not critical could be introduced to be attractive for end users that have their jobs potentially delayed in time.

Bibliography

[Aut21a] T. K. Authors. *Kubernetes default scheduler pod priority queue*. 2021. URL: https://github.com/kubernetes/kubernetes/blob/46c072d9d9d8bd42aa56aceb8159b108fb1e7c67/pkg/scheduler/framework/plugins/queuesort/priority_sort.go#L45 (visited on 02/16/2022) (cit. on p. 31).

[Aut21b] T. K. Authors. *Scheduling Framework*. 2021. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/ (visited on 01/24/2022) (cit. on p. 27).

[Aut22] T. K. Authors. *Minikube get started guide*. 2022. URL: https://minikube.sigs.k8s.io/docs/start/ (visited on 02/27/2022) (cit. on pp. 57, 58).

[BBH18] B. Burns, J. Beda, K. Hightower. *Kubernetes*. Dpunkt, 2018, pp. 1, 13–14, 28–30, 32–39, 45, 51, 56–58, 59, 63–66, 77, 241 (cit. on pp. 17, 19–27, 32, 39, 58).

[Ber05] T. G. Berger. "Konzeption und Management von Service-Level-Agreements für IT-Dienstleistungen". PhD thesis. Technische Universität, 2005, pp. 18, 21, 27 (cit. on pp. 53–55).

[Doc21] K. Documentation. *The Burgeoning Kubernetes Scheduling System*. 2021. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/ (visited on 03/26/2022) (cit. on p. 27).

[Fei15] D. G. Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015, pp. 9, 10 (cit. on pp. 38–40).

[FK21] W. Q. (Fan), Z. Kai. *The Burgeoning Kubernetes Scheduling System*. 2021. URL: https://www.alibabacloud.com/blog/the-burgeoning-kubernetes-scheduling-system-part-1-scheduling-framework_597318 (visited on 01/19/2022) (cit. on pp. 29, 30, 32).

[FWB07] X. Fan, W.-D. Weber, L. A. Barroso. "Power provisioning for a warehouse-sized computer". In: *ACM SIGARCH computer architecture news* 35.2 (2007), pp. 13–23 (cit. on pp. 59, 60).

[Gui20] C. B. Guinevere Saenger Eduar Tua. *Scheduler extender*. 2020. URL: https://github.com/kubernetes/design-proposals-archive/blob/main/scheduling/scheduler_extender.md (visited on 01/20/2022) (cit. on p. 28).

[Han13] S. Hansun. "A new approach of moving average method in time series analysis". In: *2013 conference on new media studies (CoNMedia)*. IEEE. 2013, pp. 1–4 (cit. on pp. 47, 48).

[Hua20] W. Huang. *Create a custom Kubernetes scheduler*. 2020. URL: https://developer.ibm.com/articles/creating-a-custom-kube-scheduler/ (visited on 01/20/2022) (cit. on pp. 28–32).

[Hua21] W. Huang. *Scheduling Framework*. 2021. URL: https://github.com/kubernetes/enhancements/tree/master/keps/sig-scheduling/624-scheduling-framework (visited on 01/20/2022) (cit. on pp. 28–30, 32).

[IBM21] IBM. *CIS Kubernetes Benchmark*. 2021. URL: https://cloud.ibm.com/docs/containers?topic=containers-cis-benchmark (visited on 11/30/2021) (cit. on p. 35).

[JS19] A. James, D. Schien. "A Low Carbon Kubernetes Scheduler". In: *ICT4S*. 2019 (cit. on pp. 17, 42–45).

[Kha17] A. Khan. "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application". In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933 (cit. on pp. 19, 20).

[Kli11] F. Klinker. "Exponential moving average versus moving exponential average". In: *Mathematische Semesterberichte* 58.1 (2011), pp. 97–107 (cit. on p. 47).

[Kub21a] Kubernetes. *Cloud Controller Manager*. 2021. URL: https://kubernetes.io/de/docs/concepts/architecture/cloud-controller/ (visited on 12/21/2021) (cit. on pp. 21, 22).

[Kub21b] Kubernetes. *Considerations for large clusters*. 2021. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (visited on 01/11/2022) (cit. on p. 26).

[Kub21c] Kubernetes. *Getting started (lower table with providers listed)*. 2021. URL: https://kubernetes.io/de/docs/setup/ (visited on 12/20/2021) (cit. on p. 20).

[Kub21d] Kubernetes. *Getting started (lower table with providers listed)*. 2021. URL: https://github.com/kubernetes/kubernetes (visited on 12/20/2021) (cit. on p. 20).

[Kub21e] A. of Kubernetes Open source code. *Kubernetes open source repository heap sort function*. 2021. URL: https://github.com/kubernetes/kubernetes/blob/46c072d9d9d8bd42aa56aceb8159b108fb1e7c67/pkg/scheduler/framework/plugins/queuesort/priority_sort.go#L45 (visited on 01/24/2022) (cit. on p. 27).

[Kub21f] A. of Kubernetes Open source code. *Kubernetes open source repository initialization of pod heap sort function*. 2021. URL: https://github.com/kubernetes/kubernetes/blob/0f3e6609388defe7d0149216cb0409531f8d33b3/pkg/scheduler/internal/queue/scheduling_queue.go#L102 (visited on 01/24/2022) (cit. on p. 27).

[Kub21g] Kubestone. *Sysbench - Scriptable database and system performance benchmark*. 2021. URL: https://kubestone.io/en/latest/benchmarks/sysbench/ (visited on 12/02/2021) (cit. on pp. 37, 38).

[Kub21h] Kubestone. *Sysbench project documentation*. 2021. URL: https://github.com/akopytov/sysbench (visited on 12/04/2021) (cit. on p. 37).

[Kub21i] Kubestone. *Sysbench sample benchmarks*. 2021. URL: https://github.com/xridge/kubestone/tree/master/config/samples (visited on 12/04/2021) (cit. on p. 37).

[Kub22a] Kubestone. *gwf source download first scenario*. 2022. URL: http://gwa.ewi.tudelft.nl/datasets/gwa-t-10-sharcnet (visited on 03/08/2022) (cit. on pp. 60, 61).

[Kub22b] Kubestone. *gwf source download second scenario*. 2022. URL: http://gwa.ewi.tudelft.nl/datasets/gwa-t-4-auvergrid (visited on 03/30/2022) (cit. on pp. 61, 62).

[Kub22c] Kubestone. *Sysbench sample benchmarks*. 2022. URL: http://gwa.ewi.tudelft.nl/datasets/gwa-t-10-sharcnet/report/ (visited on 03/29/2022) (cit. on pp. 60, 61).

[Kub22d] Kubestone. *Sysbench sample benchmarks*. 2022. URL: http://gwa.ewi.tudelft.nl/datasets/gwa-t-4-auvergrid/report/ (visited on 04/02/2022) (cit. on p. 61).

[LS92] L. Luqi, R. Steigerwald. "Rapid software prototyping". In: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*. Vol. 2. IEEE. 1992, pp. 470–479 (cit. on p. 30).

[map22] electricity map. *electricityMap - Database property*. 2022. URL: https://docs.google.com/spreadsheets/d/e/2PACX-1vQymR9eNK7U9bDSUBlyegx0y6FPhpe-mVBGniPzGtWDjZyHb8gI2NHSx-S49EXBhCkDe8dqfJAvsi3C/pubhtml?urp=gmail_link (visited on 02/18/2022) (cit. on p. 47).

[MPAB13] J. J. M. Moreno, A. P. Pol, A. S. Abad, B. C. Blasco. "Using the R-MAPE index as a resistant measure of forecast accuracy". In: *Psicothema* 25.4 (2013), p. 501 (cit. on p. 48).

[Omu22] S. Omura. *k8s-scheduler-extender-example*. 2022. URL: https://github.com/everpeace/k8s-scheduler-extender-example (visited on 02/17/2022) (cit. on p. 53).

[red21] redhat. *Red Hat OpenShift*. 2021. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift (visited on 12/01/2021) (cit. on p. 36).

[RKS+21] 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". In: *arXiv preprint arXiv:2106.11750* (2021) (cit. on pp. 17, 41, 42, 45).

[Tan21] V. Tanzu. *K-Bench Documentation*. 2021. URL: https://github.com/vmware-tanzu/k-bench (visited on 11/30/2021) (cit. on pp. 35, 36).

[VMw21] VMware. *vSphere*. 2021. URL: https://www.vmware.com/products/vsphere.html (visited on 12/01/2021) (cit. on p. 36).

[WBS+21] P. Wiesner, I. Behnke, D. Scheinert, K. Gontarska, L. Thamsen. "Let's Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud". In: *arXiv preprint arXiv:2110.13234* (2021) (cit. on pp. 17, 44, 60, 68).

[web21] D. website. *What are containers*. 2021. URL: https://www.docker.com/resources/what-container (visited on 11/30/2021) (cit. on pp. 19, 20).

All links were last followed on April 14, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature