A Parameterizable Test Bed for Carbon Aware Job Scheduling

Ein parametrisierbares Testbed für kohlenstoffbewusste Jobplanung

Vincent Opitz

Arbeit zur Erlangung des Grades “Master of Science” der Digital-Engineering-Fakultät der Universität Potsdam

A Parameterizable Test Bed for Carbon Aware Job Scheduling
(Ein parametrisierbares Testbed für kohlenstoffbewusste Jobplanung)
von Vincent Opitz

Arbeit zur Erlangung des Grades "Master of Science" der Digital-Engineering-Fakultät der Universität Potsdam

Betreuer:in: Prof. Dr. rer. nat. habil. Andreas Polze
Universität Potsdam,
Digital Engineering-Fakultät,
Fachgebiet für Betriebssysteme und Middleware

Datum der Einreichung: 15. Oktober 2024

Abstract

Hallo!

Write Abstract

Fix Gutachter on titlepage

Zusammenfassung

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

1 Introduction and Motivation

In times of climate change, the need to reduce greenhouse gas emissions is prevalent. One area of interest is carbon dioxide produced via electricity used in data centers. For Europe, total usage of electricity for datacenter is placed at 2.7% in 2020 [Web [3]]. This is projected to increase to 3.4% in 2030. However, not all energy is produced equally: while a data center may source its power from the public grid, it itself is sourced from different producers. These include high-carbon intensive technologies such as coal and gas but also include low-carbon sources such as wind and solar. The latter, follows a diurnal rhythm over the day as shown in Figure 1.1.

As the sun shines more during the day, a bigger part of the total power production comes from solar, reducing the overall *carbon intensity* of the grid. This can be used for *carbon-aware scheduling*. By planning work in data centers to be executed during such low-intensity times, the emissions for a given workload can be reduced.

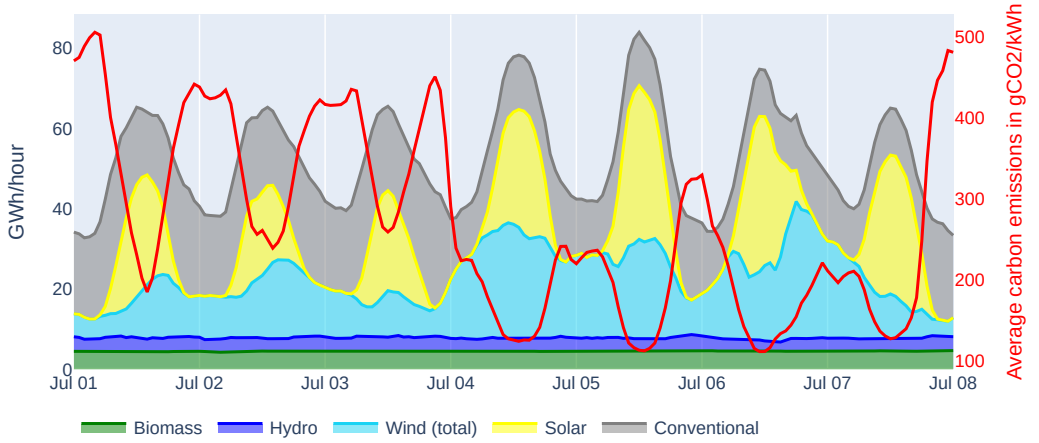

Figure 1.1: Mix of energy production in Germany for the first week of July 2024 with the resulting hourly average carbon emissions per kWh [Web [20]]. Solar production peaks at noon, reducing carbon intensity, creating times, where jobs may be scheduled more carbon efficiently.

Current work on carbon-aware scheduling focuses on three approaches: Shifting workloads *temporally*, deferring them until carbon-intensity is lower. Moving workloads *spatially* by executing them in lower carbon regions. *Scaling* resources vertically to carbon-intensity [0]. Another strategy for executing jobs during low-carbon timeframes is to *suspend & resume*: a job may be, for example, stopped as carbon intensity is increasing and resumed when a certain threshold is reached.

One common theme in related work, however, is that the workload models do not include program heterogeneity: while real-world programs may include high-powered

times for computation and low-powered times e.g. I/O, this is not reflected in literature. Suspending and resuming a workload also carries no overhead in prior work.

A prominent example of suspendible workloads is machine learning. Currently, there are major pushes to make AI more environmentally friendly [10].

Structure of this Thesis In this work, we will first conduct a structured literature review to outline the state of the art of carbon-aware scheduling. We then conduct power measurements on a select machine learning workload. These will be used to define a new workload model, STAWP, that includes startup costs and has phases of different power levels. We then modify an existing testbed, GAIA, for our implementation of CARBS, a carbon scheduling testbed under the changed assumptions. In the end, we will evaluate how the carbon-emissions of our new heterogeneous jobs with startup costs compare against the workload model used in literature.

1. Are there carbon savings under a workload model including resume-overhead and power-heterogeneity?
2. How does this compare to previous work in the field using the homogeneous model?
3. Which jobs are better suited for carbon-aware scheduling?

2 Background

In this chapter, we will introduce some basic terminology as well as provide further information surrounding carbon-aware scheduling.

2.1 The Composition of the Public Grid

As outlined in the previous chapter, the energy production of the public grid is supplied by different producers. Renewables create orders of magnitudes less emissions than conventional technologies.

Technology	gCO_2/kWh
Nuclear	5
Hydro, Wind	12
Solar	35
Gas	530
Coal	1079

Table 2.1: The carbon intensity of different energy sources [Web [22]]

As supply and demand in power need to be balanced, the composition of the grid changes according to demand. On the daily scale, there are at least two dimensions to this.

For once, there is the aforementioned diurnal rhythm of solar production. Zooming into one day, there is a mismatch between demand and renewable supply. During the morning hours, additional power needs to be generated as people wake up, prepare coffee and commute to work. The same happens during the evening when they return home (minus the coffee). Those two times, however, are when solar is not yet produced at full capacity so that additional demand is answered with conventional power generation. This is called the *duck curve*.

In addition to that, there are also seasonal effects: As shown in Figure 2.1, during warm seasons (e.g. July), more solar is produced than in colder seasons and the share in renewables rises accordingly. During the Christmas holidays in December, however, people spend more time at home, requiring less overall energy and less conventional power needs to be produced.

The power demand also affects the overall composition. During times of low demand, less conventional power needs to be produced and renewables have a higher share as well.

All the above is a local or national view on the public grid. When looking at energy production globally, a spatial dimension is added. This takes effect in the form of different countries having peak solar production at different times (following the earth's rotation)

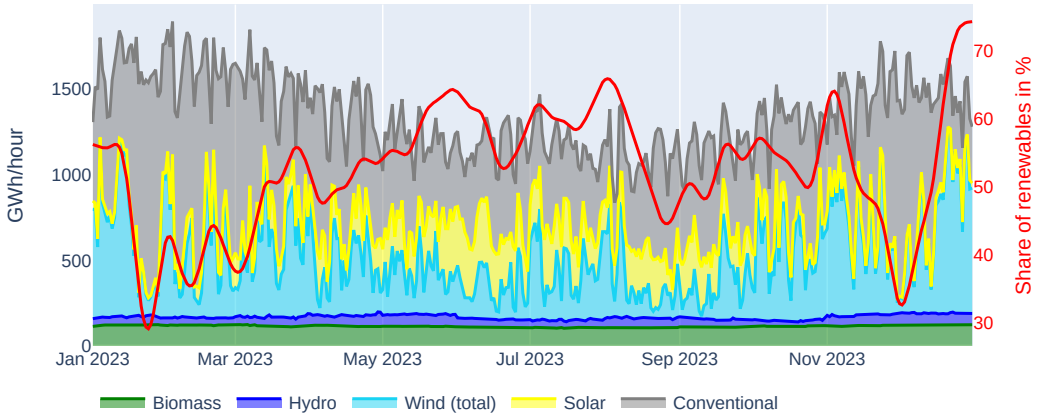

Figure 2.1: Mix of energy production in Germany in 2023 [Web [20]]. Seasonally, renewables will be highest during summer but may also spike during holidays. Thus, yearly time scopes can also be considered for carbon-aware scheduling.

but also takes effect in the composition of production capabilities. Some countries may have less investment in renewables or use nuclear power plants, which are also deemed low-carbon are largely[1] independent of the weather.

Thus, national public grids have a certain affinity for carbon-aware scheduling[16]. Generally, the higher the renewable capacities, the higher the potential carbon savings from such an approach.

2.2 Power Grid Signals

Carbon-aware scheduling commonly uses one of following metrics or signals: *average emissions* are the metric describing the amount of carbon per unit of power. These are calculated using the weighted average of all power supplies at a point.

Another metric is *marginal emissions*. Marginal emissions are closely related to the way energy producers in- or decrease production. Due to *merit-ordering*, additional demand on the grid is dispatched to the cheapest power plant with remaining capacity. These power plants, also called *marginal power plants*, correlate with emissions. With the cheapest power coming from renewables, additional demand is supplied with more expensive, more carbon-intensive, conventional power plants.

A highly simplified example is provided in Figure 2.2. In the real-world, market mechanics, over- and underproduction prevention etc. play into the way the grid is made up of. The bottom line is that the marginal emissions use a *direct causal approach* between deciding to use power and the carbon emissions associated with producing that power [Web [33]].

Two providers of signal data are *Electricitymaps* [Web [22]] and *WattTime* [Web [45]]. Electricitymaps has a free access API for historical and current average emissions. Both

[1] that said, atleast in the last 3 years, France needed to throttle nuclear power plants during heatwaves. The reason for that being that cooling water was either too hot or missing. [Web [0, 27, 0]]

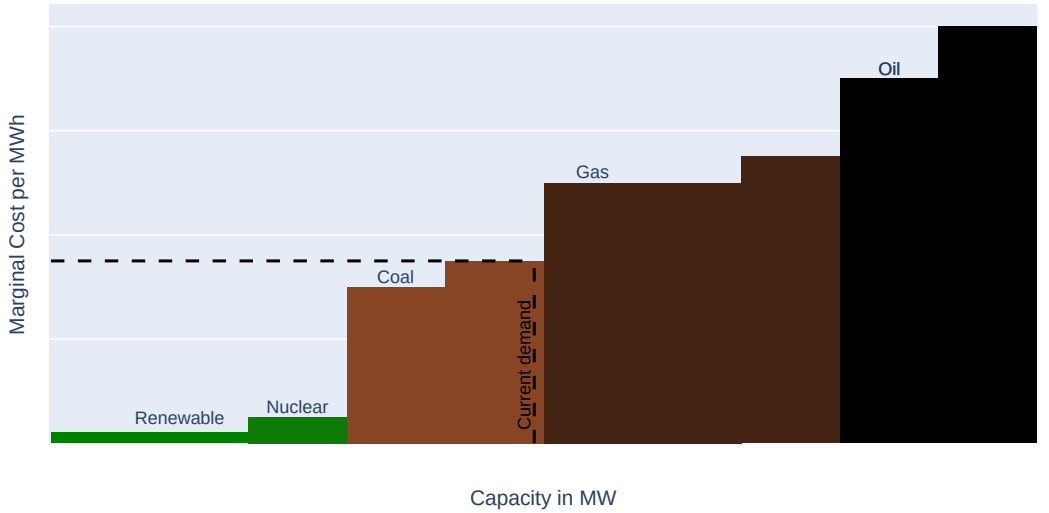

Figure 2.2: Idealized energy grid displaying how demand is met by power plants. Additional demand is dispatched to the cheapest plant with remaining capacity. In this example, increased demand will be dispatched to a, not yet utilized, gas plant. Thus, the individual decision to increase demand *now* is reflected in high marginal emissions. The costs and carbon emissions loosely follow [Web [44]] and [Web [32]]

providers only offer the marginal signal commercially, however. In the scope of this thesis, the average signal will be used. Note that current literature is still split on which signal is best. A recently released paper by Sukprasert et al. [12] may be read for further information on signals and arguments for either signal.

One reason for using the average signal is *curtailment*. Curtailment encompasses any methods that reduce production of renewable energy. Conventional power plants take more time than renewables to adjust their production. Thats why, if there is a sudden lack of demand, curtailment methods such as turning off wind turbines, selling power at a loss, or charging batteries may be used. Carbon-aware scheduling via the average emissions signal lowers the amount of curtailment needed, because otherwise left-over renewables get utilized due to better matching demand. Another argument made, for example by Fridgen et al. [4], is that by increasing power demand during times when renewable production is high, investments in renewables would be promoted. Lastly, as renewable energy is generally cheaper in production than non-renewable energy, scheduling work on low carbon periods coincides with cheaper energy prices as well. While this would not reduce costs on most energy contracts, some contracts do use dynamic pricing [Web [24]].

Workloads in a Datacenter According to Tanenbaum and Woodhull [13], there are three environments in which scheduling may take place. *Batch systems* describe environments in which there is no user interaction. Scientific simulations and computations, machine learning workloads, and data processing tasks are examples of such workloads [12]. A user may submit their job, and it would be executed according to the scheduler at some point in time. On the contrary, in *interactive* settings, a user expects quick responses to their inputs. An example of interactive workloads are web requests. In a recent work by Souza

et al. [11], web requests were answered carbon-awarely by using spatially distributed cloud services, dispatching requests to regions with lower carbon intensity. Regulations, such as the European GDPR [Web [35]] play into this, however. The last environment is *real-time* systems. There, deadlines and predictability dictate how a scheduler operates.

For this work's topic, (temporal) carbon-aware scheduling, only batch systems will be considered as they allow more freedom when scheduling workloads.

2.3 Measuring Power

There are multiple options for measuring the power of a given computer. One way of classifying these options is categorizing them under *logical measurements* or *physical measurements*.

Logical ones create a model on some metrics and derive the used power. One example is using Linux's *perf* tool to read hardware performance counters, then assigning an energy cost to select counters and multiplying that.

Advantages of choosing a logical approach are that no external hardware is needed and that the overhead of the measurement is low, as the hardware counters are being kept track of anyway. Disadvantages on the other hand are that such a model has to be created or chosen and includes some form of error as all models do.

Physical measurements follow another route; measurement devices are put between the operating hardware and the power supply. The point where a power measuring device is inserted dictates what could and could not be measured, a wall-mounted measurement device could only measure all power going into a computer and not differentiate between individual programs.

The advantages of physical measurements are that they can give a more holistic measurement of a system as would be the case for a wall-mounted measurement device. Performance counters inform only about parts of an overall system. By measuring all ingoing power, side effects such as increased fan speeds can also be captured. Portability can be an issue, though. Measurement devices could be limited to specific vendors, require specific interfaces, or may only be rated up to a specific power level.

3 Related Work

In this chapter, we will present the prior work that we build upon. Our approach to finding that related work will be subject of the first section.

3.1 Systematic Approach

We conducted a structured literature review to get an overview of the current state of carbon-aware scheduling: First, two groups of keywords are brainstormed for use in querying academic search engines, one group corresponds to carbon-awareness, and the other deals with computing environments. The specific keywords used for each group are listed in Table 3.1.

Group	Keywords
carbon-awareness	energy efficiency, energy consumption, carbon impact
computing environments	datacenter, load balancing, scheduling, job shop, job management, compute cluster, hpc, placement, cloud

Table 3.1: List of keywords for each group used in the literature study

Using these two groups, we then created Google Scholar queries via the cross-product between them. Use of the double-quotation feature restricts the results further. For each query, we then read the abstracts of, on average, the first 5 results. Depending on if their titles seemed subjectively unfit, some results were skipped.

These are then entered into a spreadsheet. With this initial basis, we further explored papers through *connected papers* [Web [23]] or looked up individual authors on a subjective basis. Some papers from the HotCarbon Workshops [Web [28]] also proved to be related. We then sort each paper into one of the following categories:

1. Green, meaning that they seem very connected and are good first entries into the topic
2. Orange indicates they are somehow connected to the paper and might be read at a later date
3. Red, the paper is either irrelevant or has some other flaw. These will not be used again in the course of our work

Results The results are exported as `literature_study.csv` via the appendix .2. According to its evaluation using the `literature_study_evaluation` Jupyter Notebook,

145 abstract were read. Using the above categorization, 99 are marked red, 31 orange, and 15 green.

3.2 State of the Art

Overall, through this approach, multiple unrelated papers were found and excluded. Nevertheless, with the green ones, the current state of carbon-aware scheduling can be outlined. Out of those, the following are of particular interest.

Carbon-Aware Scheduling There are multiple implementations in recent literature. Wiesner et al. [16] use a simulation approach to examine temporal shifting via their *WaitAWhile* algorithm. Their workload model consists of known length jobs with constant power and they explicitly rule out resumption costs. Jobs can then either use suspend & resume or not, both scenarios are evaluated. These are then tested under the assumption of different job deadlines, meaning that each job has to be completed by a certain timeframe. As outlined the in previous section, different regions of the world have different carbon intensity curves. So testing multiple countries is part of their evaluation. Their main takeaways are that increased deadlines lead to reduced carbon emissions, but that this effect also has diminishing returns. They also deduce that regions such as California, with high amounts of solar power, have a higher potential for carbon savings in comparison to nuclear-heavy regions such as France.

Another closely related paper was made by Sukprasert et al. [12]. They add on the above workload model by incorporating CPU requirements and differing arrival times for jobs. In the previous paper, job arrivals were assumed to be spread out evenly. Here, they use three real-world traces resulting in a spread of lengths, requirements, and arrivals. Due to the latter, weekly observations on carbon emission curves play into the result as well. For example the circumstance that power demand is lessened on weekends. Jobs are scheduled on *spot* instances (cheap VMs that seek to increase cloud utilization), *on-demand* instances (expensive short-notice VMs) or *pre-bought* VMs (medium cost, but may not get utilized). The paper then discusses balancing performance as well as carbon- and dollar costs.

The two mentioned simulations assume perfect knowledge for near-future carbon emissions and schedule jobs in foresight. Hanafy et al. [9] use an algorithmic approach based on the *Online Pause and Resume Problem (OPR)*. Suspend & resume scheduling is reducible to OPR. The Problem involves deciding to buy a resource for each time slot or not to. Switching between these decisions (suspending or resuming a workload) carries an added cost. They thus also work on the lack of overhead in workload models. In our approach, resuming or starting a job carries a *time* cost, which in turn results in emissions.

There is another paper building upon WaitAWhile by Hanafy et al. [5]. Their goal is to examine the differences between energy efficiency and carbon efficiency. In one scenario, they add different overheads for resuming a job according to WaitAWhile. Under their model, carbon-efficiency still improved with increasing deadlines, but they assumed the scheduling to be independent of the overhead. In our work, the scheduler will take overhead into account.

With many testbed implementations in literature, Wiesner et al. [15] aim to generalize one for comparability. Their *Vessim* is a simulation framework for carbon-aware testing aimed at extensibility and software architecture.

Slurm Schedulers Of the analyzed papers, the one made by Goiri et al. [18] seems to be among first papers to deal with carbon-aware scheduling by implementing it as a *Slurm* plugin called *GreenSlot*. Section 5.1 contains more information on Slurm and its plugins. In contrast to our scenario, where we try to optimize carbon emissions via the public electricity grid, GreenSlot is about data centers having their own renewable energy production (e.g. via having solar panels on the roof). Using weather data, GreenSlot predicts when solar energy production is high, scheduling jobs preferably in those time frames. Under their workload model, jobs may also have deadlines. If those deadlines cannot be met using only renewables, Greenslot will additionally schedule jobs on high-carbon times, based on electricity cost.

Springborg et al. [1] propose an architecture for incorporating Python plugins into Slurm. They implement an exemplary energy-efficiency plugin for a single-node cluster environment.

Summary There are multiple approaches to testbeds for carbon-aware job scheduling. A common theme is to examine job lengths, arrival times, and deadlines for effects on carbon emissions. Overhead from resuming jobs is considered in some prior work but, atleast to our knowledge, none of them use a time-based overhead for scheduling as well. Power heterogeneity was not included in any of the examined papers.

Mention GAIA again here

4 Defining a New Workload Model

We believe that the workload model using constant-power used in prior work is too big of a simplification. In a workload model that supports different phases, high-powered phases could, for example, be matched to carbon-intensity valleys. Through this chapter, we will first conduct power measurements on a machine learning workload. By that, we will infer a new model, StawP, that can represent our findings of different power levels and the cost associated with resuming said workload.

4.1 Power Measurements on Machine Learning Jobs

Test Environment The experiments are run on a personal computer, its specification being outlined in Table 4.1, the values being determined with the `hwlist` and `lshw` commands.

Operating system	Ubuntu 24.04 LTS
Kernel	Linux 6.8.0-39-generic
CPU	AMD Ryzen 5 1600X Six-Core Processor
Memory	16GiB
GPU	GeForce GTX 1070

Table 4.1: Environment parameters of the power measurements

Measurement Tool As outlined in Chapter 2, multiple measurement options exist. As the HPI has the *Microchip MCP39F511N Power Monitor* (henceforth called *MCP*) on-site, one will be used as a physical measurement device. The MCP is placed between the device to test and the wall-mounted power supply. A picture of it can be found in Figure 4.1. It can report the current power consumption in 10 mW steps, each 5ms. The software used for reading out MCP data is *pinpoint* [7]. In our case, pinpoint us run on the device-to-test.

Benchmark Machine learning (ML) was used as the main motivation for suspend & resume scheduling in prior works [16] and thus was also chosen by us to be measured and modeled.

The concrete model and framework are secondary to our measurement. In our case, a small model, RoBERTa [Web [25]], is chosen. It is a natural language model, that due to its size generates fewer data points for processing, and allows faster iterations for the measurement script.

There is a vast amount of machine learning frameworks. For our high-level model, the feature set of the framework only needed to support checkpointing, resuming, and

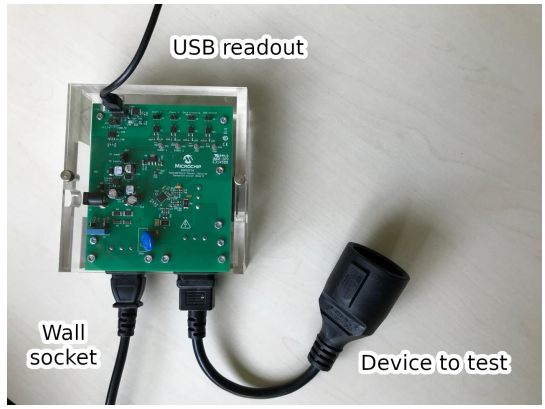

Figure 4.1: The *Microchip MCP39F511N Power Monitor*. It can be inserted between a devices power supply and the wall socket. Data is output via USB, which can be read by an external device or the device-to-test.

some basic form of logging. A glance at the documentation of popular frameworks such as *torch*, *tensorflow*, and *huggingface* shows that these features are commonly supported. With little bias towards any framework, huggingface was chosen. The huggingface trainer supports callbacks, we modified a basic ML script to also log timestamped events when a training iteration, for example, starts or ends. These events are then saved into another `.csv` file for each experiment.

Conducted Experiments A script, `measure_roberta.sh`, was used to execute each experiment. On a high-level view, the following experiments were conducted:

1. Run the whole program from start to finish
2. Run it partially, checkpointing after some steps, sleeping, and resuming from that step
3. Run it partially, checkpointing after some steps but aborting before the next checkpoint. Then resume as above.
4. Run only the startup phase up until the ML starts
5. Do nothing, measure the system at rest

Experiment 1 gives a baseline for what the job looks like without suspending. Numbers 2 and 3 will be used to determine the overhead of checkpointing the job. Experiment 4 is used to validate the other ones, as it is a subset of the others. The last experiment is necessary to determine the baseline energy consumption of the environment.

To execute these experiments inside a repeatable bash script, additional command line parameters were added to the program. For example, a boolean parameter `--resume_from_checkpoint` and an integer parameter `--stop_after_epoch` are used for experiment 1 to 2. The way of conducting experiment 4 was to copy the script, and delete everything after the imports.

Creating Repeatable Measurements As this is being run on standard hardware on a standard operating system, all experiments are subject to noise. For example, *Dynamic frequency and voltage scaling (DFVS)*, the OS technique of increasing CPU frequency for performance needs, adds power in an uncontrolled way. Also, background tasks may happen *randomly*, increasing power usage. To reduce noise, we reduced the number of processes to a minimum. We also used the Linux tool `cpupower`, as shown in the snippet below, to set the CPU frequency to a set value of 3.6 GHz, which is the maximum frequency:

Listing 4.1: Used operating system information

```
MINFREQ=$(cpupower frequency-info --hwlimits | sed -n '1d;p' \
    | awk '{print $1}')
MAXFREQ=$(cpupower frequency-info --hwlimits | sed -n '1d;p' \
    | awk '{print $1}')

cpupower frequency-set --min ${MAXFREQ} &>/dev/null
cpupower frequency-set --max ${MAXFREQ} &>/dev/null

# ... conduct experiments

cpupower frequency-set --min ${MINFREQ} &>/dev/null
cpupower frequency-set --max ${MAXFREQ} &>/dev/null
```

As machine learning makes use of available GPUs, its frequency should also be similarly set to a defined value. NVIDIA provides a guide on how to conduct power measurements on GPUs [Web [19]]. Our used GPU, the NVIDIA GTX 1070, is not capable of fixing the frequency as of the time of conducting these experiments. While it is supposed to be possible, there seems to currently be driver issue preventing this [Web [43]]. Thus, the frequency of the GPU was not fixed. To reduce the effect of frequency scaling here, the time between experiments was increased generously so that any impact from such scaling reoccurs throughout each run, reducing dependencies between runs.

For the training, data is downloaded and saved, which is however deleted before the next experiment. The Python process is also not kept between runs, forcing a reload of any libraries.

Conducting each Experiment Each experiment was repeated ten times to be able to average out noise later. Between each run, there is a `sleep` period of 10 seconds and one of two minutes in the partial executions. Additionally, `pinpoint`'s feature of measuring before and after the actual program-to-test was used. This leads to a period of 30 seconds being measured around the program execution. Plotting these additional time frames gives a quick visual indicator of whether experiments are sufficiently isolated from each other, ergo when the power draw is at the baseline as the actual program starts.

Collected Data For each experiment, a named and timestamped folder is created in the `/power-measurements` folder of our repository. Each folder then holds a `.csv` with pinpoint's timestamped power measurements. The added timestamped logging is saved into another `.csv`. On the way to the final measurements, we plotted each experiment early and visually checked if there were any obvious errors or mistakes.

Determining the Baseline Power Draw A sample run of the baseline experiment is shown in Figure 4.2. The blue dots represent each data point. The red line is a smoothed Gaussian trend line with $\sigma = 2$.

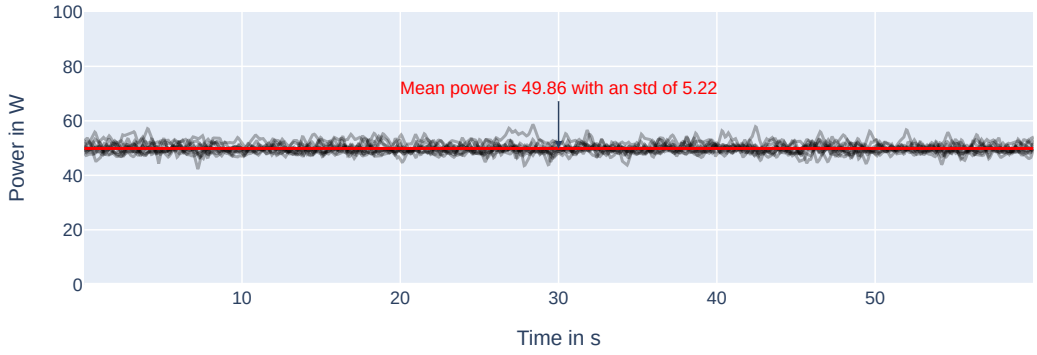

Figure 4.2: In the baseline experiment of measuring the system at rest, the average power draw is about 50 W. The black lines are Gaussian smoothed trend lines with $\sigma = 2$.

Across all ten runs, the average baseline power draw is calculated via the mean of all data points. This comes out as an average of 49.8 W with a standard deviation of 4.4 W.

The baseline power draw will be less interesting going further but will put perspective on the other experiments. The standard deviation gives a broad idea of the environment noise.

The Full Run For the unsuspended machine learning, Figure 4.3 shows the stacked trend lines of the 10 different runs. A single sample run is provided in Figure 4.4. The times of each event occurred within a standard deviation of less than 1 second. Thus, for simplicity's sake, we refrained from doing a more elaborate statistical analysis of the different runs as the visual check of them being very similar seemed enough.

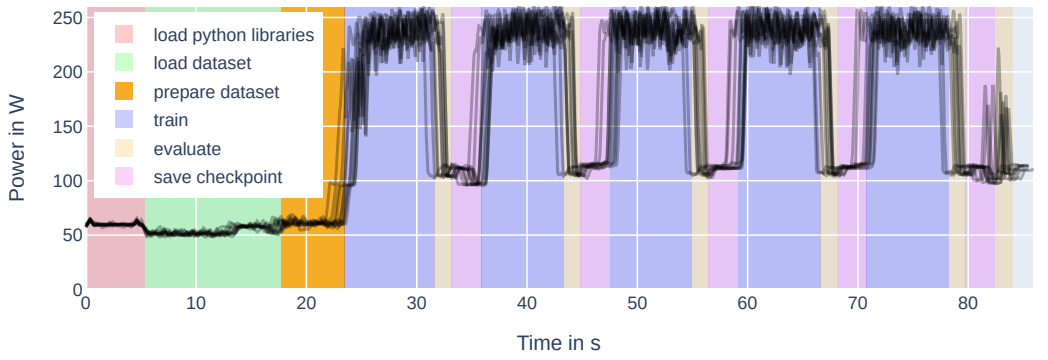

Figure 4.3: Each black line is a Gaussian-smoothed trend line of a run. We stack all 10 measurements into one time series, showing that each is similar to the others. The background represents our derived phases. Their borders are when our manually added events occurred on average. The timestamp of each event had a standard deviation of less than 1 second, meaning that there was little variation.

The main takeaways from these measurements are:

1. There is a long (about 25%) start-up phase, which is spent in starting Python, loading libraries, and setting up the training data.
2. There are periodical work phases; a high-power training phase is followed by low-power evaluation phase and a low-power checkpointing.
3. A higher variance in measurements occurs during the training phases in comparison to the others.

This already shows, that improvements upon the constant-power model used in [16] are possible. For example, in this case, the start-up phase has a much lower power-draw than the work-phase.

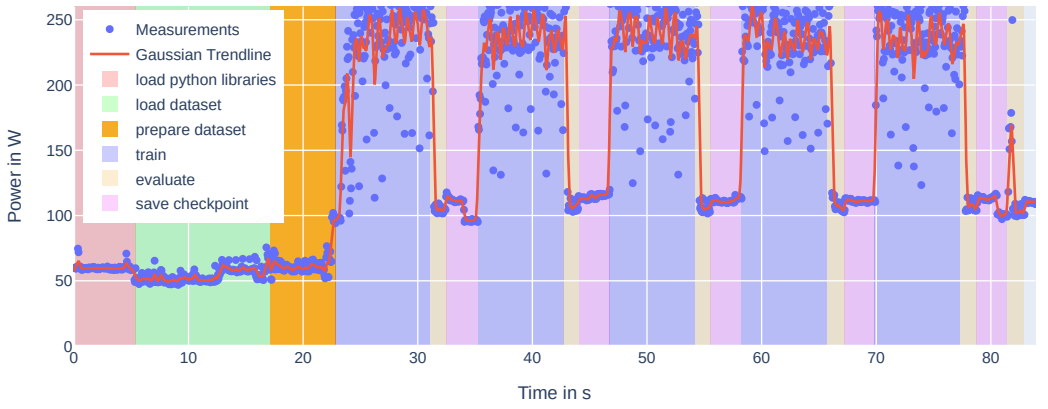

Figure 4.4: Sample run of the full run experiment, the background indicates each phase. These phases are derived from our manually added logging.

Results of Checkpoint and Restore Similarly to before, results will be discussed using the stacked plots of Figures 4.5 and 4.6.

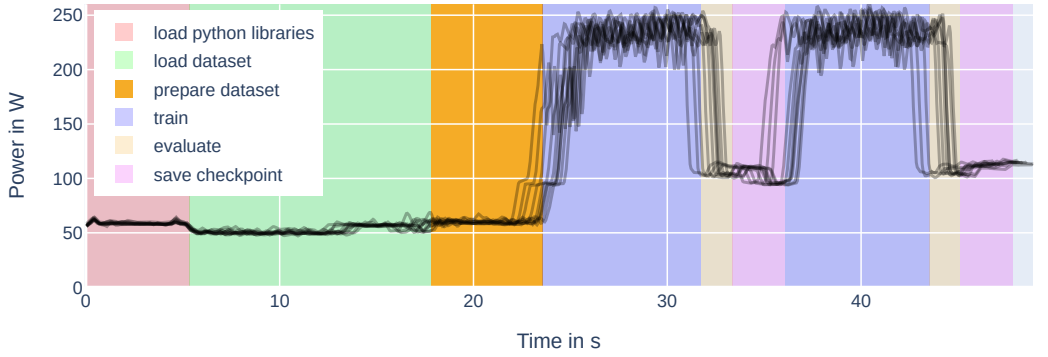

Figure 4.5: Stacked trend lines of the experiment for stopping after the second epoch

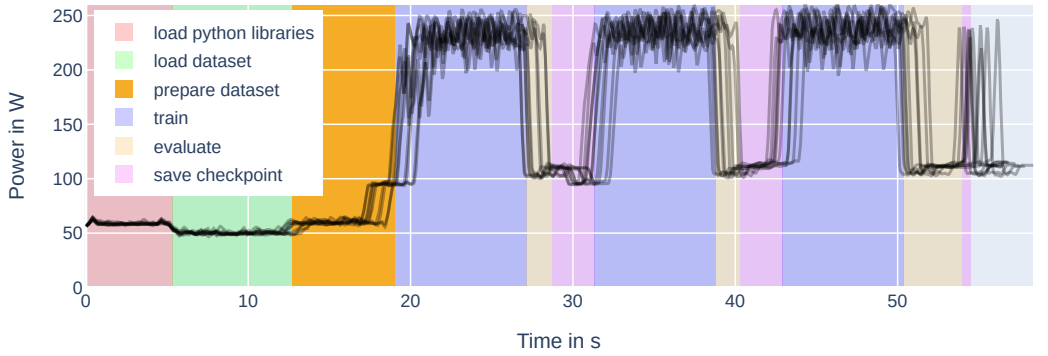

Figure 4.6: Stacked trend lines of the experiment for continuing after the second epoch

Here we can observe that:

1. The amount of work done is the same. Similarly to the full-run experiment, the ML training still takes the full 5 epochs and has the same work-phases
2. There is no overhead from checkpointing itself, as the checkpoints are being created regardless of them being resumed from later.
3. Resuming the jobs results in an added start-up phase. This phase is slightly shorter by a few seconds than the ones in the full runs, perhaps due to not needing to download the dataset again.

Results of Aborting the Checkpoint and Resuming Unlike the previous experiment, where the training is stopped as soon as a checkpoint is created, this time the program will be stopped just before a checkpoint is created (in this case just the second checkpoint would be saved). Ideally, this represents the maximum overhead from a suspend & resume strategy. Again, the results are visualized in Figures 4.7 and 4.8.

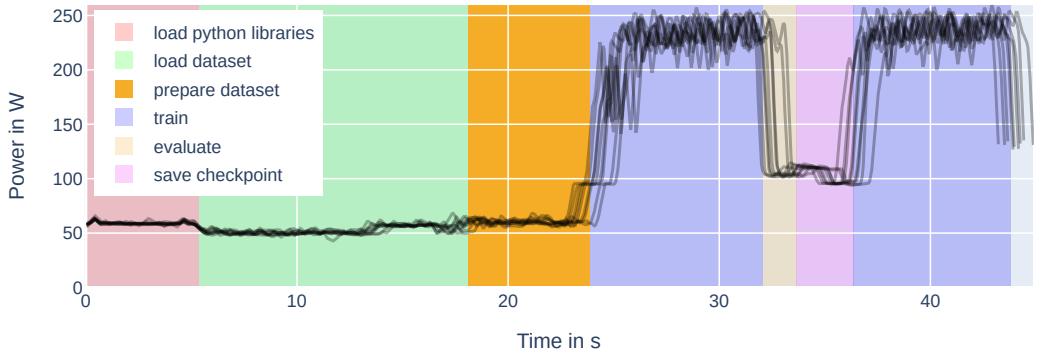

Figure 4.7: Power draws of the ML up until stopping after epoch 2

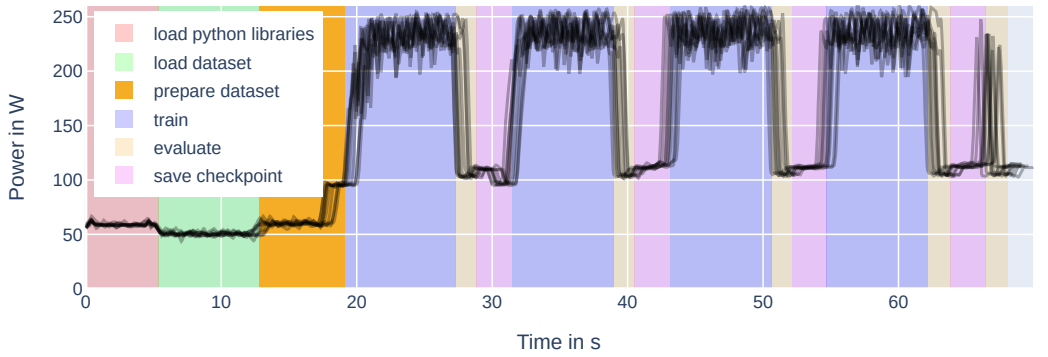

Figure 4.8: Power draws after continuing from the second checkpoint

Attention should be paid that,

1. The behavior of the repeated start-up phase is kept
2. There is now a full additional training and evaluation phase added to the overall work, the aborted checkpoint is also repeated.

While this may sound artificial, it could happen in environments where the interaction between the scheduler and the job is not well orchestrated, for example in an environment where jobs are stopped *at random* like in a cloud spot instance. The average overhead from stopping the job at random vs. stopping after a checkpoint will likely fall at half the cost of an epoch.

Calculating the Energy Costs of each Run The energy costs of each experiment are contained in Table 4.2. Showing that there indeed is an overhead created from suspending at different parts of the program execution.

Experiment	average energy cost	standard deviation
1, whole run	12.97 kJ	0.04
2, suspend after checkpoint and resume	13.94 kJ	0.1
3, abort early and resume	15.72 kJ	0.07

Table 4.2: Energy costs of the different experiments. Unlike the no-overhead assumption in prior work, we can show that suspending a workload does carry overhead.

4.2 Introduction of STAWP

Now that we know what a high-level job looks like, we can pick it apart and reduce the real-world measurements of one program to a more generic model. Summarizing the findings from the previous paragraphs; it was shown that

- The given job has phases that have different power draws
- Checkpointing & resuming carries overhead in the form of startup costs and possible wasted work.

We thus propose STAWP: a model combining startup, work, and phases. Its definition is given the form of the Python implementation in Listing 4.2.

Listing 4.2: Definition of STAWP as a Python class

```
class Stawp(TypedDict):
    startup: List[Phase]
    work: List[Phase]

class Phase(TypedDict):
    name: str
    duration: float
    power: float
    is_checkpoint: NotRequired[bool]
```

Some simplifications are made: the duration of each phase is well known and the power per phase is constant. Phases can also be named for later reference. These phases essentially define a step function, i.e. a piecewise constant function. Unlike a traditional step function, the start- and endpoints of each piece are encoded implicitly by the previous phases. With this, a simple time-to-power function can be defined, that looks up the input time by traversing the phases in order.

Initially, we considered allowing any expression instead of a constant value for power and then using Python's `evaluate()` to e.g. allow a function-per-phase model. In Section 6.3, having a rather restrictive step function will be of advantage, however.

Fitting STAWP to RoBERTa We use the following pseudocode:

Listing 4.3: Pseudocode for turning RoBERTa's power measurements into STAWP

```
1  phases = []
2  iterate with event_1 and event_2 pairwise through the events:
3          # do this in respect to each run:
4          phase_measurements = measurements between pair of events
5          phase_power = average of phase_measurements
6          phase_duration =
7              average time of all event_2
8              - average time of all event_1
9          phases += {phase_duration, phase_power, name of event_2}
10
11 manually reorder phases into startup and work
```

In line 4 we query each run for phase-associated measurements based on that runs event timings. The durations of the phases are calculated similarly by taking the average time the logging occurred during the measurements.

Using this strategy on the ten complete runs results in Figure 4.9, which shows the derived model in black with the previous Figure 4.3 in less opacity. The startup phase looks well approximated, visually however there is some error during the work phases. The training phases each have a high variance, which is not captured by the constant power approximation. After each training phase, the power goes down seemingly linearly, which is also approximated by the constant. One notable point: this model is a superset of the previous constant-no-overhead model used in the related work. Previous jobs can be modeled with just one phase of work, resulting in constant power over its execution. Leaving the startup phases empty creates no overhead on resuming.

Model Error Analysis To analyze the error of this model, we cross-validated the power's RMSE and the total energy using `scikit-learn`'s `LeaveOneOut` strategy. The first one gives a measure of the model's accuracy on a short-time (sub-second) scale, the latter tells the long-time (whole job) scale accuracy of the model.

Each of the ten runs is taken as the ground truth while the other nine are used to create the model. The results are the following: the power between the prediction and remainder has an RSME of 39.3 W while the difference in total energy is calculated as -0.1 kJ.

Interpreting this, it seems that the model performs poorly as a predictor of the exact power used at some time point as the RMSE is rather large (think of the maximum power drawn being about 250W). However, in the context of carbon-aware scheduling, this should not be too big of an issue as the time frames for carbon emissions are orders of magnitudes larger. For example, Electricitymaps has a resolution of 1 hour for carbon emissions intensities. The high error likely comes from the high variance during the training phases which is not captured in the model.

The total energy predicted by the model is very close to the actual real-life experiment, this should mean that the total carbon calculated on the model should also be close to the carbon emitted by the real program.

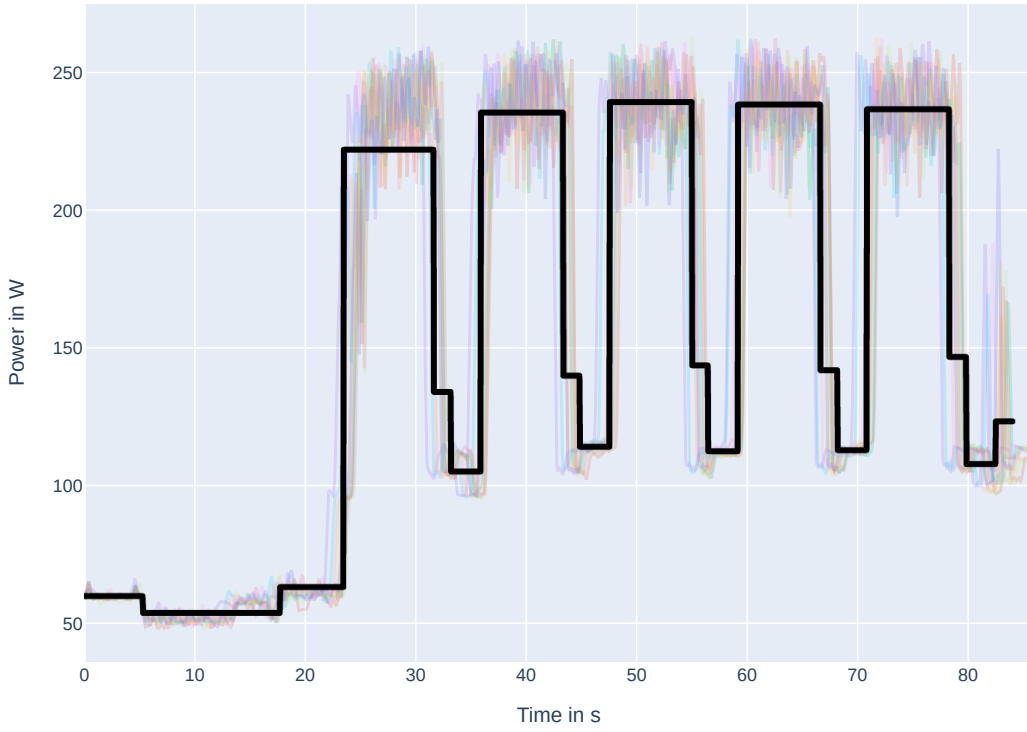

Figure 4.9: StawP vs RoBERTa. The black line is the new model we propose and the others lines are individual runs. Each of StawP's phases is the result of averaging the runs measurements between two events. The length of a phase is similarly derived from the average time of each event.

5 Choosing an Implementation Approach

5.1 Viability of a Slurm Plugin

Our first idea was to use a non-simulation approach. The HPI's Data Lab [Web [30]] runs a *Slurm* cluster and also has some nodes with power measurement infrastructure included. Slurm is an "open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters" [Web [40]]. The job scheduling part is important here, it also supports a plugin infrastructure that includes scheduling plugins. One of the highlighted papers [18] in the related work section specifically used Slurm for its carbon-aware scheduler implementation and thus seemed like a good starting point for our work.

Installing Slurm Locally For our purposes, a local installation sufficed as we did not need to run heavy workloads but instead only needed the scheduling part of Slurm. While there is a Slurm `apt get` package for Ubuntu, extending this with plugins is not possible. Plugins need to be included during the Slurm compilation, meaning we had to do the same.

Slurm's documentation provides some guidelines on how to install Slurm, which we followed. We tried cloning the main branch, but the compilation process encountered an error midway. However, using the predefined released versions worked. Installing `munge` [Web [31]] is also necessary, which is used for authentication in Slurm.

One problem arose as we tried to start the `Slurmd-` and `Slurmctld` services. The first one is the worker service that later executes jobs submitted to Slurm. The latter is the main controller that, for example, schedules jobs on workers. While the command to start them did not fail, when inspecting all nodes via Slurm's `scontrol` command, it showed that all nodes were `DOWN` instantly.

Dealing with Slurm's problems usually leads to inspecting its logs. In this case, the logs showed the following:

Listing 5.1: Slurm's cgroup configuration is missing according to the logs

```
$ less config.log

error: Couldn't find the specified plugin name for cgroup/v2
    looking at all files
error: cannot find cgroup plugin for cgroup/v2
error: cannot create cgroup context for cgroup/v2
error: Unable to initialize cgroup plugin
error: Slurmd initialization failed
```

Slurm uses Linux's `cgroup` feature to manage the submitted job's hardware resources. The log hints at some problems related to Slurm's usage of it. The solution was to provide a `cgroup.conf` file to Slurm. In our use-case of getting Slurm to simply start, this did not need to be very sophisticated, so we just used an off-the-shelf configuration file [Web [21]]. Running `scontrol` again, we were now able to see idling nodes, meaning that Slurm was successfully installed from source.

Creating a Scheduler Plugin The Slurm documentation provides a short guide on how to add a plugin to Slurm [Web [39]]. As a start, we simply copied Slurm's default scheduler, which is also a plugin, to the specified directory under a new name, and added that new name to Slurms build files. It was then time to recompile Slurm. Now, however, during the recommended `autoreconf` step, an error occurred:

Listing 5.2: Plugin recompilation errors

```
$ autoreconf
auxdir/x_ac_sview.m4:35: warning: macro 'AM_PATH_GLIB_2_0'
    not found in library
configure:25140: error: possibly undefined macro: AM_PATH_GLIB_2_0
      If this token and others are legitimate,
      please use m4_pattern_allow.
      See the Autoconf documentation.
autoreconf: error: /usr/bin/autoconf failed with exit status: 1
```

The solution, while not very obvious, was to install the `libgtk2.0-dev` library [Web [29]]. We then added a *hello world* logging to the new plugin. Upon inspecting the logs after scheduling a hello-world-script, that log message showed up as well - confirming that our new scheduler plugin was running.

Adding More Logic to the Scheduler Plugin One very helpful step for developing inside Slurm is to enable the debugging flags. This must be decided before compilation by using the `--enable-developer` and `--disable-optimizations` flags during the `configure` step. With that, debug symbols are added to the outgoing binaries. As we were using VS Code, we could then attach its debugger to the running Slurm thread with full functionality.

The code of the plugin runs in its own thread and there is no sandboxing or similar around it. Thus, there are seemingly no limitations on what can be done inside the plugin. For testing, we read information on the incoming jobs such as set constraints or the user-supplied comments. Terminating the jobs was also possible inside the plugin.

Problems of a Scheduler Plugin One big problem manifested in that not all jobs *showed up* inside the plugin's job queue. If we submitted 6 jobs, via Slurm's `squeue` command, only a part, such as the last 3, logged inside the plugin. A possible explanation for this could be Slurms' scheduler architecture: while there is a scheduler plugin, there also is a scheduler inside Slurm's main loop. These both work in parallel [Web [41]] and they use the same job queue.

To hack around this, we tried disabling Slurm's main scheduling loop by setting `sched_interval=-1` inside the Slurm configuration file. While this had the effect of being able to access all incoming jobs inside the plugin, it also had the side effect of disabling all logic concerning starting the jobs. So by choosing this route, the plugin needs to re-implement a lot of extra logic, which conventionally is not inside the scheduler plugin.

We also looked into whether there were any API hooks that are exposed to the plugin. Up until Slurm version 20.11, scheduler plugins had callbacks[Web [37]] such as a job just getting submitted. There also was support for *passive* schedulers that were invoked when determined by Slurm. The version we used, however, 23.11, removed all such functionality and documentation [Web [38]]. Now, all plugins are implemented via threads that only have callbacks for when they are started and stopped.

Thus, since there was no apparent way of getting around this scheduler race condition between the plugin and the main loop, the scheduler approach was dropped. While we would not say that a plugin approach is impossible, the effort to implement one from scratch seems very high. The public documentation for developing Slurm is scarce. There is a mailing list that can be searched, but it looks to be mostly aimed at administrating Slurm and not developing it.

Other avenues that could be explored are Slurm's Lua plugins. There is also a *Slurm simulator* [Web [29]] which could potentially be used for carbon-aware scheduling simulation. But we did not look into it much for reasons of little documentation and a seeming lack of continued support.

5.2 Simulating with Carbs

At that time during out work, the previously discussed paper by Hanafy et al. [6] was released. Our testbed, Carbs, is a fork of their implementation.

Description of the Existing GAIA Simulator We first describe their testbed and make it clear which part is our work and which is not. A class diagram is provided in 5.1 which is heavily inspired by the existing architecture diagram in their paper.

The main part of GAIA is the scheduler. At program start, it takes parameters that determine the scheduling, such as whether tasks can be interrupted, how to use the carbon information (e.g. use a perfect-information / oracle approach or use a running average), and how to balance the dollar cost of scheduling. The latter we removed for simplicity, which is marked in the figure via the red color of the reserved and spot instances. In our case of carbon-aware scheduling, we did not need that feature.

In GAIA, the scheduler had multiple queues of tasks. The paper presented an approach of a short queue (tasks under 2 hours) and long tasks (all other), short tasks are scheduled on spot instances. Again, as we already removed the spot instance, feature, the queue multiplicity was removed likewise. The task queues are based on historical traces. GAIA provided multiple traces that are described in the paper.

In their workload model, tasks have a predetermined length, as saved in the trace. Tasks also contain some data that is used for scheduling such as their arrival time in the system, and how long they should wait until execution. Some properties such as required CPUs were removed.

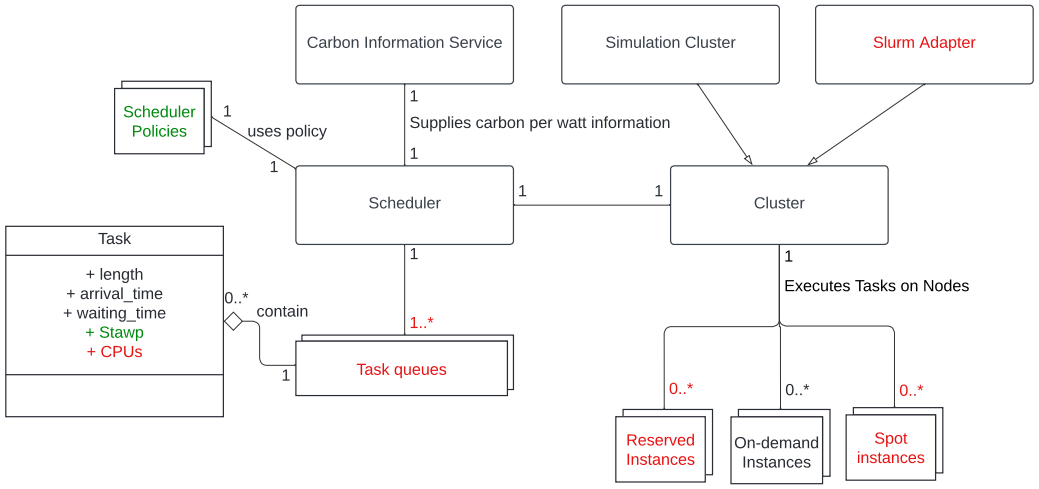

Figure 5.1: Class diagram of CARBS, which is a fork of GAIA (black). Green indicates addition. Red indicates removal of the original GAIA features.

The Scheduler policies are also marked green, as we added scheduler policies that could use STAWP. Specifically, we added a (non-) suspend & resume scheduler that works on the assumption of perfect carbon intensity knowledge, this will be further explained in the latter sections.

Scheduled tasks are submitted to a cluster. In our case, this is a simulated, already implemented, cluster that simply logs when each task is entered and finished. In the original paper, they also used an open-source adapter that submits the tasks to an actual Slurm cluster. While this approach could have been used, we decided against it based on not adding further complexity. Thus, we also removed the Slurm adapter from our implementation.

6 Schedulers with Stawp

We will first summarize the assumptions that GAIA makes on the problem of carbon-aware scheduling:

1. job lengths are known
2. all considered jobs are batch jobs and can thus be shifted temporally based on user deadlines
3. carbon information is provided for near future periods, no error is considered
4. there are no hardware limitations or considerations, all jobs are executed in an isolated manner
5. jobs have a constant power draw and can be suspended & resumed for free

In Carbs, the last point will be modified going further. As shown in Section 4.2, jobs now have a predefined, variable power draw according to Stawp. Resuming a job will also carry an additional startup phase with it.

6.1 Using Stawp in Carbs

The first step to be taken was to add the described model to the jobs as outlined in Figure 5.1. In GAIA, jobs were generated from `csv`-formatted traces, one example being provided in Listing 6.1.

Listing 6.1: Excerpt from the Alibaba-PAI cluster trace

```
arrival_time,   length,     cpus
0.0,            6302.0,     1
68.0,           1000.0,     1
463.0,          1570.0,     3
838.0,          23549.0,    1
```

A jobs' `arrival_time` signifies when it is added to the simulation in seconds. The `length` of a job similarly encodes the seconds a job needs to fully execute. The `cpu` column was not interesting for our work, as previously discussed.

The chosen way of supplying the model information was to add two new columns to the trace files, which entail the type of job and a column for arguments. As of now, the following types are supported: `constant`, `constant-from-stawp`, `stawp`, and `ml`. The first two `constant` types are for backward compatibility with GAIA, and are for the evaluation later.

Use this in the implementation

Type `stawp` is the main way of creating a job with phases such as the one used in the experiment. The supplied argument in that case would entail a Python-readable `dict`,

essentially a JSON-like definition of the model as described in Listing 4.2. As this was just a string, we then used Python's `eval()` to read it back into a Python object. For example, to model `roberta.py` via Stawp, the definition in Listing 6.2 could be used:

Listing 6.2: Simplified definition for a job similar to the experiment

```
{
    'startup': [
        {'name': 'Start', 'duration': 5.34, 'power': 59.9},
        {'name': 'Finish Imports', 'duration': 12.36, 'power': 53.77},
        {'name': 'Data loaded', 'duration': 5.75, 'power': 63.17},
    ],
    'work': [
        {'name': 'Train', 'duration': 8.17, 'power': 221.93},
        {'name': 'Evaluate', 'duration': 1.54, 'power': 134.0},
        {'name': 'Save', 'duration': 2.72, 'power': 105.1,
            'is_checkpoint': True},
    ] * 5
}
```

Instead of supplying the phase information via `.csv`, we also added a parameter to set a value for all jobs.

To use Stawp and read out the power of a job at a given time, a function was created that essentially traverses all phases in order, keeping track of the time, and returning the power of the phase when the requested time is reached. This was also used to create Figure 4.9.

A generalizing helper type is `ml`, which takes a dictionary of parameters and converts it to Stawp. A similar job as to the one above would be created with the following:

Listing 6.3: Generic definition for machine learning jobs

```
{
    'start_duration': 23.45
    'start_power': 60
    'training_duration': 8.17
    'training_power': 221.93
    'evaluate_duration': 1.54
    'evaluate_power': 63.17
    'save_duration': 2.72
    'save_power': 105.1
    'epochs': 5
}
```

6.2 Uninterrupted Oracle Scheduling

This is the first scheduler to be adjusted for Stawp. In this setting, jobs are not able to suspend & resume and the carbon trace is fully known. GAIA already implemented this under the constant-power assumption. The algorithm for which is pseudocoded in Listing 6.4.

Listing 6.4: Pseudocode for GAIA's non-interrupt oracle scheduler

```
1  function schedule_job_no_checkpointing_resuming(job):
2      possible_starts = []
3      for every possible starttime with respect to deadline:
4          calculate carbon_emissions_of_job at starttime
5          add carbon_emission, starttime to possible_starts
6
7      return starttime with least carbon emissions
8
9  function carbon_emissions_of_job(job, carbon_trace):
10     sum_of_emissions = 0
11
12     for each second of job:
13         sum_of_emissions += carbon_trace at timepoint * constant_watt
14
15     return sum_of_emissions
```

In order to adjust this to Stawp, where power is a function of time, only line 13 needed changes. Thus, the previously mentioned time-to-power function replaced the Watt constant. Evaluating this change will take place in Section 7, after another scheduling algorithm is introduced.

6.3 Suspend & Resume Scheduling for Heterogeneous Jobs

Under the constant-power, no startup-overhead assumption of GAIA, the algorithm worked like the following:

1. order all time slots until the deadline by their emissions ascending
2. execute the job on the first length many time slots

This minimizes emitted carbon, as only the best time slots are used. Due to the diurnal nature of carbon emissions, this essentially uses each valley, scheduling a job there. With deadlines increasing, the amount of valleys also increases accordingly.

In order to implement a scheduler for the improved job model, we chose to use *linear programming (LP)*. In a nutshell, LP is an optimization method, that given linear equations and constraints, finds a set of variables (or a vector) that minimize (or maximize) an optimization goal. Translating this to the context of carbon-aware scheduling, the resulting vector encodes when a job should be executed. The optimization goal in this case is to minimize the emitted carbon. A constraint for example could be that all time slots being

executed should add up to the job length. The big challenge is then to model this in the form of mathematical equations, which will be most of this section's remainder.

Reflections on Linear Programming Linear programming is its own programming paradigm. Unlike, for example, imperative programming, where each instruction is coded explicitly, enabling classic debugging and such, in LP, a mathematical model is written which is then solved by a *solver*. Different implementations for these solvers exist, but at least in our case, the result output by a solver will either be an error, or the result set. As the solver uses the whole model simultaneously, debugging individual parts of the model is not possible outside removing them from the overall model.

Thus, an iterative approach to LP proved useful. We encode some part of our overall problem into the model, solve it, and immediately plot the variables of the result set and check them visually. The visual check is very important: many times the solver solved the problem *so well* that jobs are scheduled in a way that the emitted carbon is minimized, but the jobs are executed in unintended edge cases of the mode. Examples of this will be given in the following parts.

Another part of programming LP is *just knowing the tricks*. Many common programming operations, especially ones for control flow, cannot be expressed directly. Instead, they need to be written via linear equations. There are some mappings for such operations, but finding out how seems to be harder than in other paradigms. The LP community appears to be smaller in comparison and online resources are limited, making finding help harder. Some *tricks* will be highlighted in the latter parts as well.

Modelling Carbon-Aware Scheduling in a Linear Program In this part, we will present the iterative steps taken to model the scheduler as an LP problem, highlight some implementation details and visualize the results of each step. We used Python's `PuLP` library, which allows creating LP models to standard file types and also helps in calling external solvers as well as querying the results.

Executing the Job Firstly, as the job has a certain length and a set deadline, we chose to use an array of time slot variables for the result set. Each such variable is a boolean signifying whether the job is executed at that time. For the first constraint, the sum of all boolean variables (which are represented as 0 and 1) should add up to the job length, effectively executing the job for length-many time slots. The necessary objective function is to be defined as each time slot being executed multiplied by the carbon-per-watt at that time slot. The code for this is shown in Listing 6.1.

Listing 6.5: LP Implementation for basic scheduling

```
1 prob = LpProblem("CarbonAwareScheduling", pulp.LpMinimize)
2 work = LpVariable.dicts("work",
3     (t for t in range(DEADLINE)), cat="Binary")
4
5 prob += lpSum([work[t] * carbon_cost[t] for t in range(DEADLINE)])
6 prob += lpSum(work[t] for t in range(DEADLINE)) == WORK_LENGTH
7
8 solver = pulp.Gurobi_CMD(timeLimit=timelimit)
9 prob.solve(solver)
```

The first line creates an LP problem and defines the optimization goal to be a minimizing one. Then, a variable "work" is defined as being `DEADLINE`-many integer-indexed boolean variables. These will be modeled as the time slots. We add linear equations via `prob +=`. Line 5 defines the optimization goal; each time slot will be multiplied by the carbon emitted there. As one factor is a boolean, this will only increase the sum if the time slot is scheduled. The dictionary `carbon_cost` contains a constant factor for each time slot. The last two lines start the solver. `PuLP` offers an open-source solver by default, but during development this proved to be very slow and also did not offer features such as time limits and retrieving intermediate results. We thus chose the *Gurobi* solver [Web [42]], a commercial solver that also offers academic licenses. That one supported the previously lamented features and increased development time by a lot.

The running example will be a deadline of 250 time slots, the job has a startup phase of 20 units and has 100 work units. The above code leads to a result set that is visualized in Figure 6.1 Looking at the figure, the results are not too surprising: the chosen time slots are just the points where emissions are lowest, essentially being equal to the result of the WaitAWhile-Algorithm.

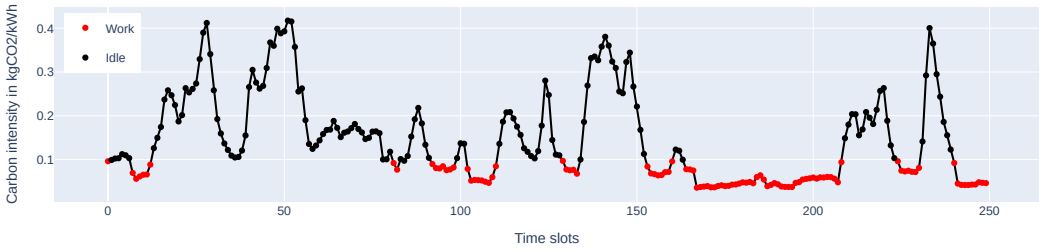

Figure 6.1: Resulting schedule using the first iteration of the LP scheduler. Each time slot, with its corresponding carbon intensity, is assigned to be worked or not. As this iteration has no notion of startup-costs, only the lowest time slots are used with many suspends in between.

Adding Overhead via Startup Phases In order to add a cost to starting a job, the code following in Listing 6.6 is complemented, some parts are excluded for brevity.

Listing 6.6: LP Implementation for overhead

```
1 for t in range(DEADLINE - 1):
2     prob += startup_finished[t] >= work[t + 1] - work[t]
3     prob += startup_finished[t] + work[t] <= 1
4     prob += starting[t] + work[t] <= 1
5
6 for i in range(STARTUP_LENGTH - 1, DEADLINE):
7     prob += pulp.lpSum([starting[i - j] for j in range(STARTUP_LENGTH)])
8         >= STARTUP_LENGTH * startup_finished[i]
```

In this snippet, two extra dictionary variables are introduced, `startup` and `startup_finished`. For every time slot (line 1), set `startup_finished` to true if and only if there is a 0 to 1 transition along the `work` dictionary (line 2). This becomes apparent when looking at Table 6.1. Notice how the numerical booleans help in this case, as negative integers get mapped to `0`, or `false` respectively.

$work[t]$	$work[t+1]$	$work[t+1]$ - $work[t]$	$startup\_finished[t]$
0	0	0	0
0	1	1	1
1	0	$max(-1,0) = 0$	0
1	1	0	0

Table 6.1: Truth table for finding when working time slots begin

Line 3 and 4 ensure that *working* time slots and *startup* time slots are mutually exclusive. The last loop defines that if (`* startup_finsihed[i]`) a startup must be finished by some time slot, the previous `STARTUP_LENGTH`-many time slots must be used for starting the job.

At this point, sometimes, the result scheduled jobs in the very first time slots. There, if the first two slots are set to work, the `startup_finished` would not be set, skipping the startup phase. This minimizes carbon but is obviously a bogus result. To fix this, we add an extra constraint that work may only be scheduled after the length of the startup phase. The emitted carbon goal is changed to also include these new startup time slots, similarly to the previous Listing 6.6. This time, the optimal schedule includes executing the job in one go, as shown in Figure 6.2. Doing a visual check, the job is also executed on seemingly the lowest carbon intensities.

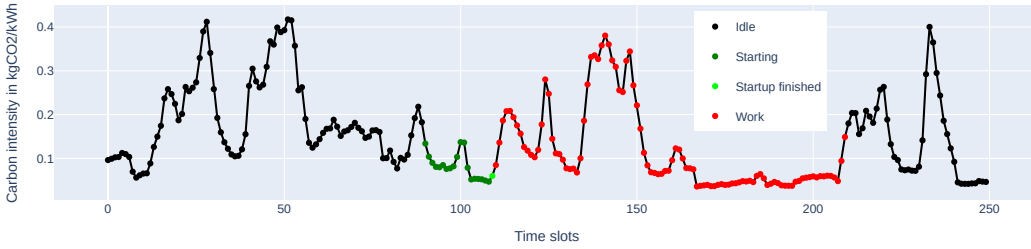

Figure 6.2: The schedule of the second iteration. In addition to time slots being assigned for work, a startup phase must also be scheduled before work can begin. The light-green mark indicates a helper variable, used for this mechanic. In comparison to the previous figure, there is now one long, connected, block where the job is worked on, as resuming a job carries an extra cost.

Adding a Notion of Progress So far, the assumption of constant power has been continued. Changing this assumption under the previous scheduler using the greedy algorithm in Section 6.2, was relatively easy, as changing the constant expression to a Python function of Stawp sufficed. This cannot be reused in LP, however, as the problem definition may only include linear equations, which a generic function is not. To add dynamic power into the linear program, we decided to *linearize* the function, meaning Stawp needed to be split up into multiple linear approximations.

As it is essentially a step function of time to power, each phase being one step, the mapping is inherently close. As such, *a lot* of equations were needed that express "if the progress in the job is t, set power to $stawp(t)$" for each time slot. For that, a notion of said progress and time is needed in our problem model. The progress inside each startup-phase needs to reset, as that can happen multiple times during the schedule. On the other hand, the progress for the productive work must not be reset between execution blocks. We add the following code to express this:

Listing 6.7: Progress Variables in LP

```
# define "work_progress" and "startup_progress" as
# DEADLINE-many integer variables

M = DEADLINE * 2
for t in range(DEADLINE-1):
    if (t>0):
        prob += startup_progress[t] >= startup_progress[t-1] + 1 - (1 -
            starting[t]) * M
        prob += startup_progress[t] <= startup_progress[t-1] + 1 + (1 -
            starting[t]) * M
    prob += startup_progress[t] <= starting[t] * M
    prob += work_progress[0] == 0
    if (t > 0):
        prob += work_progress[t] == work_progress[t-1] + work[t]
```

Basically, the idea is to count up each progress variables by 1, if a time slot is being determined for work or startup respectively (see line 12 for this).

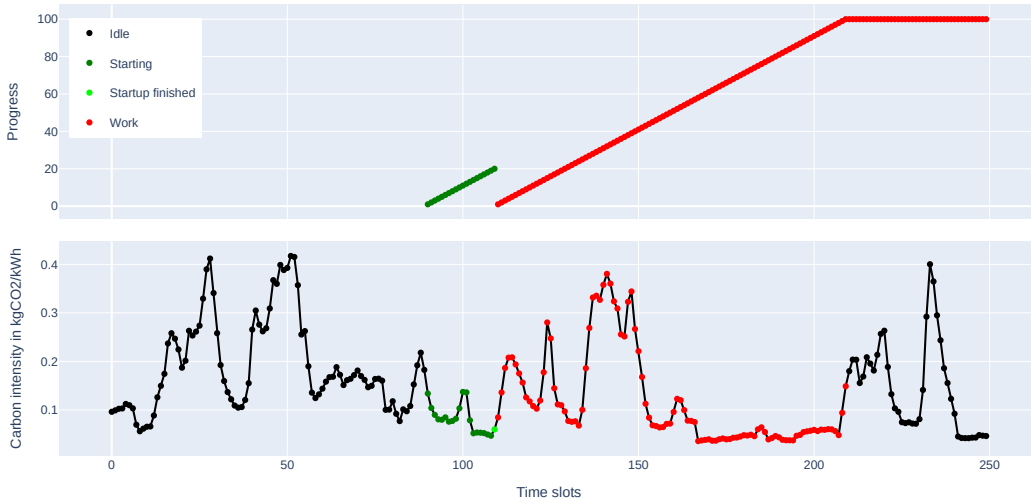

Figure 6.3: Visualisation of the result set after adding progress indicators. For either starting up or working, count up a respective integer variable. These do no yet impact the schedule itself, compared to Figure 6.2. Progress is kept even when idling.

This snippet also includes an LP "trick", namely the *Big M Method*. In line 4, a constant `M` is defined as "a *large* integer, that cannot otherwise occur in the result set", "large" being twice the amount of time slots. M can also be any other arbitrarily chosen large number.

Take line 9 as an example: remember that `starting[t]` is either 0 or 1, multiplying this by a large number means the right side is either 0 or *large*. Constraining a variable to be less than `M` effectively does nothing, called *relaxing a constraint*, as every value of that variable is less than `M` by definition of `M`. On the whole, this can expression be translated to "if a time slot is not used for starting the job, set the progress to 0, otherwise ignore this constraint", the Big M Method enables a way to add conditional constraints to a model!

While line 9 defines the `startup_progress` outside the startup phases, lines 7 and 8 are needed to increase the progress by exactly 1. Notice how depending on the type of inequality, *M* is either added or subtracted to the equation, a conditional *greater than* relation is relaxed by setting the constraint to zero.

All in all, there is now a way to keep track of a job's progress inside the model, as shown in Figure 6.3's upper graph. Attention should be drawn to the `work_progress` which keeps its value even when time slots are set to idle. The schedule found by the solver is not different from the previous one which added overhead, as the optimization goal was not changed and the progress indicators have no impact on the scheduling (yet).

Adding Power According to Stawp The goal of this part will be to have an LP variable for each phase, which indicates when the job is in that phase. When such indicators exist, the formula for calculating a time slot `t`'s carbon emissions can be changed to Formula 6.1:

$$carbonEmitted(t) = \sum_{p \in Phases} isActive(p,t) * powerOfPhase(p) * carbonEmission(t) \quad (6.1)$$

With that goal in mind, we add the pseudocode in Listing 6.8 to determine when a phase is active.

Listing 6.8: Phase detection in LP

```
1  # for each phase
2  # let phase_indicator, phase_indicator_upper, phase_indicator_lower be
       DEADLINE-many boolean variables
3  # set lower_bound to be the minimum progress this phase can occur in
4  # set upper_bound to be the maximum similarly
5
6  # do the following for each phase
7  for t in range(DEADLINE):
8      prob += progress[t] - lower_bound <= M*phase_indicator_lower[t]
9      prob += lower_bound - progress[t] <= M*(1-phase_indicator_lower[t])
10
11     prob += upper_bound - progress[t] <= M*phase_indicator_upper[t]
12     prob += progress[t] - upper_bound <= M*(1-phase_indicator_upper[t])
13
14     prob += phase_active[t] >= phase_indicator_lower[t] +
           phase_indicator_upper[t] - 1
15     prob += phase_active[t] <= phase_indicator_lower[t]
16     prob += phase_active[t] <= phase_indicator_upper[t]
```

Unwrapping this, two helper variables are added per phase and each time slot. A lower variable indicates that the previously established progress is above the threshold for a phase, while the upper variable indicates the opposite. The `phase_active` is then *active* where these two overlap (lines 14 to 17 define a logical *AND*).

The constraint of line 8 is only applied if the indicator is `false`. If it is false, the progress at the time slot must be below the lower bound (as negative numbers are equal to zero). Line 9 works on the negated indicator, applying the constraint if it is `true`. If the indicator is `true`, progress must be higher than the lower bound. Combining this, the expression `phase_indicator_lower[t] <=> progress[t] > lower_bound` is added to the model. The upper bound is formulated in the following two lines similarly.

Using this addition, the schedule now looks like the one shown in Figure 6.4. Unlike the previous times, splitting the job up into two parts is now the optimal solution. Attention should be drawn to the circumstance that the purple high-powered phase is scheduled on the lowest carbon emissions while the low-powered one is scheduled on the higher emission time slots.

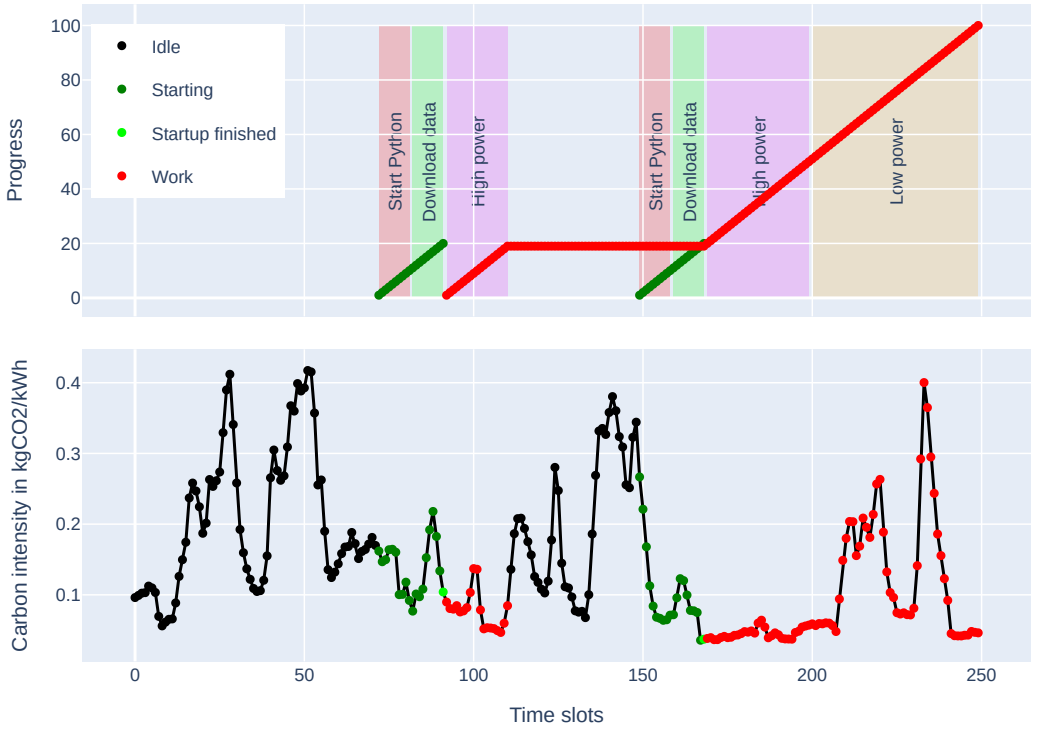

Figure 6.4: The final scheduling including differing power levels according to phases. In this example, there is a high-powered phase at 230 W and a low powered phase at 100 W. Unlike before, it is now optimal to suspend & resume the job to schedule the high-powered phase on the least carbon intensive time slots (170 to 200)

Issues of the implementation As of now, checkpointing may happen at any point in the scheduler, as the `work_progress` is never reset. Future work will focus on checkpointing at specific points, such as after specific phases like in the entry ML example in Section 8.4.

Add this to future work!

Another issue at this point is the runtime and hardware requirements for finding a solution. While the upper example of 250 time slots and a job length of 120 were found within minutes on a laptop[2] bigger problem sizes such as 5000 time slots and a length of 800 barely finish within 30 minutes with a *gap* value of 98%. The gap value is an indicator of how close the solver is to finding the optimal solution, with 0% representing the optimal solution.

Solvers are able to take advantage of multiple CPU cores; however, in our case, the issue was likely a lack of memory. Glancing into the system monitor utility on Ubuntu then shows that all 8 GB of RAM are in use and that swap space is being utilized a lot.

We integrated this scheduler into Carbs using the existing suspend & resume scheduler interface GAIA used. Previously, a binary decision whether a job ran had to be made for each time slot. This means that phase information is not yet exported into the output traces.

Decreasing the Problem Size In the previously shown Figures, each carbon emission data point corresponds to one time slot, which in turn corresponds to one unit of time in the job. Going by the resolution of the carbon emissions, each data point describes the emissions for one hour. In Carbs, job lengths are given in seconds, however. The example workload used for measuring in Section 4.1 also showed that timescales in seconds are useful.

Using a 1:1 mapping between those results in 3600 (60s * 60) time slots for each hour, resulting in very large problem sizes in turn needing long execution times and high amounts of memory.

In order to improve on the issues mentioned above, the following optimization is made: by calculating the *greatest common divisor (gcd)* between all phase durations and the time resolution of the carbon emissions (3600), all durations can be scaled down by the `gcd`. Each time slot in the model then represents `gcd`-many seconds.

The effect of this optimization heavily depends on the input phases. If there are very short phases, the `gcd` is low likewise, resulting in minimal reduction of the problem size.

[2] specifically, Lenovo T470 with 8 GB RAM and an i5-7200U CPU @ 2.50GHz

7 Evaluation

7.1 Parameter Description

We want to evaluate the carbon savings under StawP, a model containing startup overhead and heterogeneous phases, against the constant-power model used in prior work. The following approach is proposed: As the already existing job traces hold no information regarding the power or their phases, some test cases will be presented. In a rather explorative process, we use the following parameters and simulate all possible permutations of them. This is done using the `generate_evaluation_jobs.sh` script, the used parameters are listed in the following table:

Parameter	Values
Scheduling Strategy	suspend & resume, non-interrupted
Phases	alternating high- and low-powered phases, each either 30 or 60 minutes long. 200 W and 100 W respectively
Startup length	no startup, 5 minutes, 10 minutes, 30 minutes
Startup power level	100 W, 200 W
Waiting time	4 hours, 12 hours, 1 day, 2 days, 4 days
Job length	1, 2, 4, 8, and 16 hours
Carbon trace	Week from July 1. 2024 (see Figure 1.1)

Table 7.1: Overview of the parameters used for the evaluation

Another dimension will be comparing these scenarios against having no information on phases, instead only having an averaged constant wattage of the otherwise existing phases. The latter represents the previous `GAIA` implementation, where jobs had a constant amount of power at all points in time. Finding the LP solution will time-limited to 20 minutes and all jobs will be simulated to be submitted at the very first time, midnight, in the trace.

7.2 Running the Evaluation on a Cluster

As previously discussed in Section 6.3, LP scheduling tends to have high runtime and memory requirements. For that reason, the simulation was executed on the *SCORE Lab*, short for *scientific compute resources*, of HPI.

After requesting and gaining access, `rsync` was used to transfer the simulation to the cluster. Slurm could then be used to schedule all experiments individually (that were generated using the mentioned `generate_all_jobs.sh` script), which would parallelize the evaluation, leading to faster results.

One problem arose in the licensing of the used *Gurobi* solver. Under our academic license, only 2 *sessions* are allowed simultaneously. A session is defined via the hostname of the executing machine, which is communicated to the solver's licensing servers live. As such, scheduling each experiment by itself lead to many sessions being started, as Slurm distributes the jobs on the available nodes. While there is some leeway in the number of sessions, experiments crashed beyond the 5th instance as the solver did not start.

We work under this restriction by distributing the experiments across two workers via a script `run_evaluation.sh`. Each experiment has an index. One worker executes the even-numbered jobs and the other executes the odd ones. Using this even-odd strategy results in both workers having about the same amount of work.

The workers in this case are Python docker containers that were created with the help SCORE Lab's online knowledge base [Web [26]]. The command for launching one example worker, in this case, the one for even jobs, is given in Listing 7.1. Along the workload categorization in Chapter 2, `sbatch` is used to submit batch workloads.

Listing 7.1: Submitting the evaluation to SCORE Lab's Slurm

```
1  sbatch -A polze -p magic \
2      --container-image=python \
3      --container-name=test \
4      --container-writable \
5      --mem=128G \
6      --cpus-per-task=128 \
7      --time=24:0:0 \
8      --output=slurmlogs/output_%j.txt \
9      --error=slurmlogs/error_%j.txt \
10     --constraint=ARCH:X86 \
11     --container-mounts=/hpi/fs00/home/vincent.opitz:/home/vincent.opitz \
12     --container-workdir=/home/vincent.opitz/master-thesis/carbs \
13     run_all_even_jobs.sh
```

Now the simulation can be run with 128 GB RAM as specified in line 5. The `-p magic` parameter results in us using a compute node. Before submitting all jobs, we used `srun` to get an interactive session to that container, allowing us to `pip install` the requirements. Line 4, `--container-writable`, means that the named container will be reused. Restricting the nodes to be `X86` nodes via line 10 is necessary as there are other architectures available in the cluster. Without this line, Slurm can schedule our jobs on nodes incompatible to the containers' pre-compiled Python binary, resulting in an error on start [Web [36]].

7.3 Results

Running the evaluation took a combined, and very carbon-aware, processing time of 3 days and 16 hours split across two instances. In sum, 2400 jobs were simulated with differing parameters.

Effect of Suspend-Resume Scheduling In the introductory research questions, we surmised that the effects of suspend-resume scheduling may be lessened under a model that contains startup costs for resuming. To answer this, we first calculate the total carbon emissions for each job under different schedulers. The GAIA and CARBS output includes multiple entries per job if they get paused, so they get aggregated into the total.

With this, we first run an ANOVA to determine the axis under which to explore the data. Table 7.2, shows that the length and phases of a job impact carbon emissions significantly. This is not unexpected, however, as these parameters increase the time and overall energy required to run the job, in turn also increasing overall carbon emissions. The phases have a high impact as there are 3 configurations: more high-powered, more low-powered, and balanced. Waiting times being very significant as well, is consistent with prior literature [16].

Parameter	p_value	p_value (rounded)
Job length	$0.000000e+00$	0.00
Waiting time	$1.008635e-23$	0.00
Phases	$1.910669e-04$	0.00
Scheduling Algorithm	$1.827666e-01$	0.18
Startup power level	$6.641101e-01$	0.66
Phases averaged or not	$7.436032e-01$	0.74
Startup length	$9.885010e-01$	0.99

Table 7.2: Results of the ANOVA for total carbon emissions per scheduler. The lower the p-value, the higher the impact on carbon emissions.

With these findings, we will visualize jobs grouped by their length and waiting time, as their p-values are effectively zero. In Figure 7.1, all simulated jobs are plotted with their respective total carbon emissions. Blue indicates that both schedulers, with suspend & resume and the other without, had similar emissions within 1%. If the emissions are not equal, red and green dots indicate the respective scheduler performance. The remaining parameters are hashed into an integer, as that they had little effect on carbon emissions.

By eye, it can be observed that the carbon savings between the schedulers are consistently minimal for jobs of length up to at least 8 hours and for all jobs with waiting times up to 6 hours. There are however, exceptions to this: for example, 8-hour jobs with a waiting time of 2 days exhibit saving potential, but only for jobs with a startup time of less 30 minutes.

The bottom right sub-plot, showing the maximum waiting time and job length, does not follow the column trend of increased savings with increased waiting times. In a few cases, the suspend & resume scheduler performs *worse*, as can be seen by green dots being above all other dots. One explanation for this can be seen in Figure 7.2, showing the scheduling for different job lengths, with a startup time of 5 minutes, and a waiting time of 4 days. Checking the Slurm logs for the jobs highlighted in red shows that the LP scheduler stopped after hitting the time limit of 20 minutes. A result was found that is within the constraints but is not optimal. The gap value introduced in Section 6.3, for these jobs is 100%. In fact, counting the occurrences of `"Time limit reached"` in the logs shows that out of all jobs, 209 timeouts occurred.

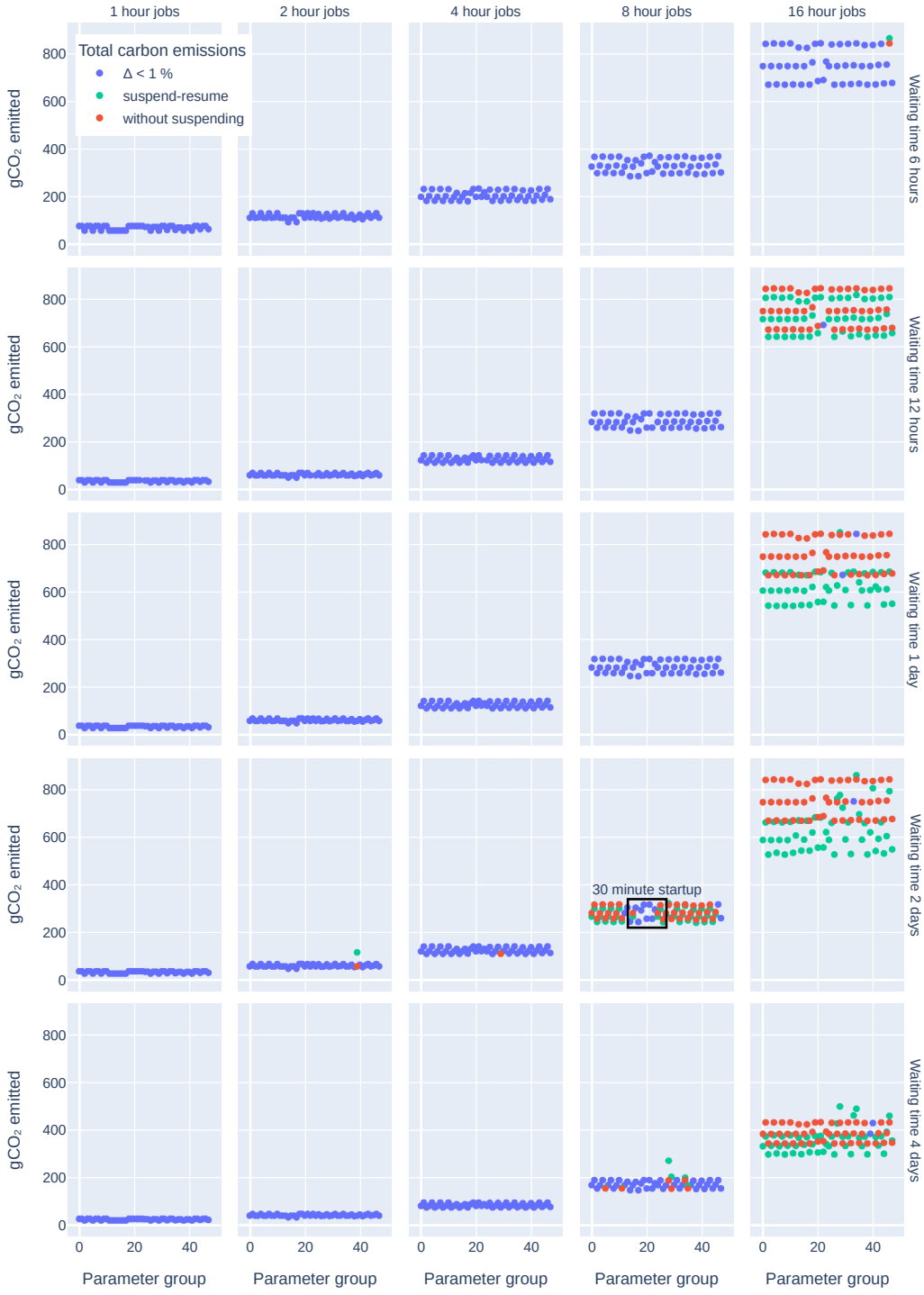

Figure 7.1: All simulated jobs and their total carbon emissions, compared between the two scheduling approaches. Blue indicates equal emissions, red and green differentiate the schedulers. The combination of startup length, startup power and phases is encoded on the x-axis. Savings from adopting a suspend & resume strategy are only noticeable for very long jobs with long waiting times.

Since half of the 2400 jobs are LP-scheduled, about a sixth of them did not find an optimal solution in time.

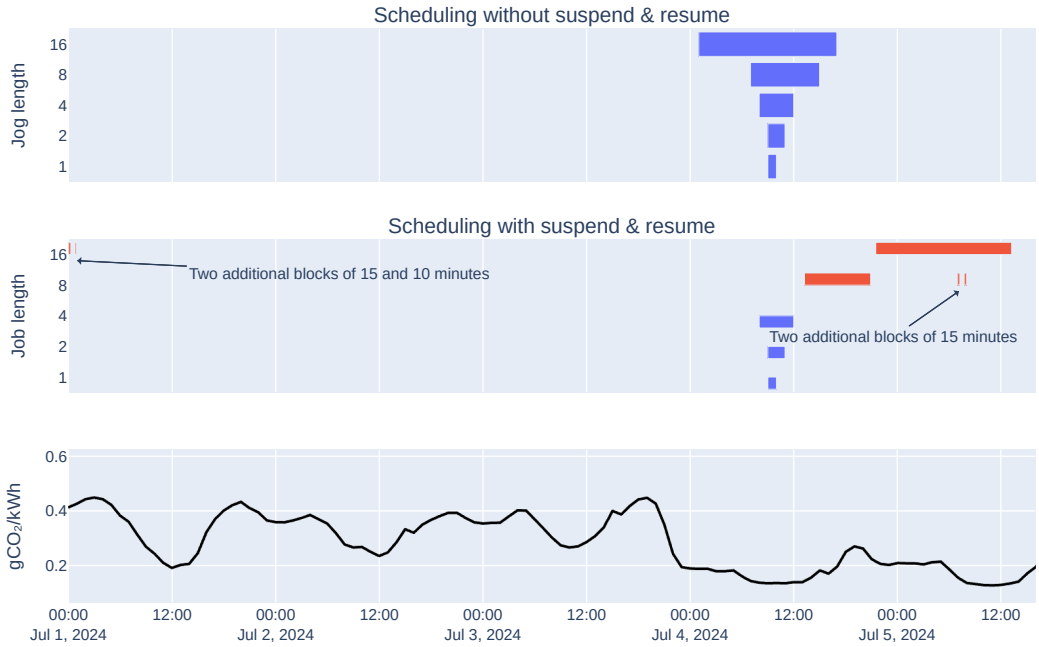

Figure 7.2: Schedules of jobs sharing all parameters but length under different schedulers. All jobs have a waiting time of 4 days and a startup time of 5 minutes and an uneven phase configuration. The red bars are suboptimal schedules that are found upon being time limited.

The cumulative distribution for the time limited solutions in Figure 7.3 shows a heavy skew towards bad solutions, if the time limit is not kept.

Figure 7.3: Cumulative distribution plot of the 209 gap values of time limited schedules. A higher gap indicates worse results. There is a heavy skew to 100%, with half of the values being above 95%

Despite this, the jobs with a constant power draw and startup costs were still being scheduled optimally according to the logs.

Effect of Phases To determine whether the differences in power introduced by Stawp enable improvements in carbon-aware scheduling, we will use a similar approach to before. We hypothesized that in suspend & resume scheduling, high-powered phases could be scheduled to the best slots, so we suspect that savings could be different between the scheduling approaches.

For Figure 7.4, we compare each parameter configuration between having a constant power and differing power. Taking the previous assumption of constant power as a baseline, the color of each data point represents the change in carbon in percent. As the scale of the figure suggests, most jobs did not exert meaningful changes in emissions.

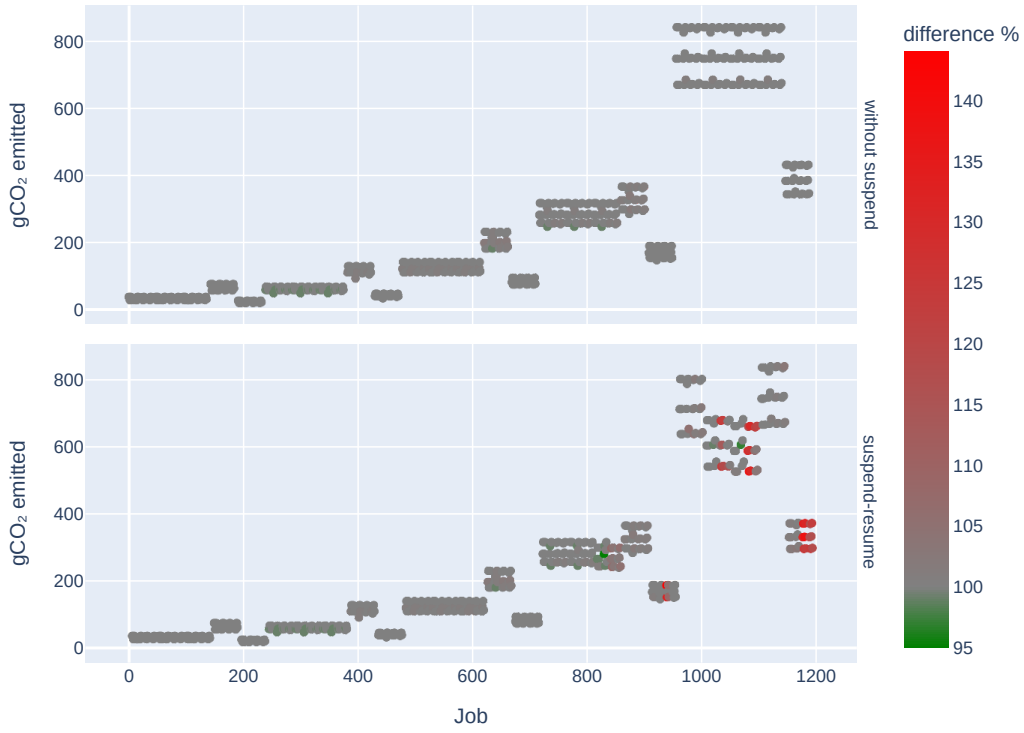

Figure 7.4: Difference in carbon emissions from having heterogeneous phases instead of a constant average. For the majority of jobs, this had an effect of ±1%. Red signifies jobs, where this increased carbon emissions. For suspend & resume scheduling, some jobs had highly increased emissions. These are likely the result of to time-outs due to increased complexity

Opposite to our intention, some jobs had highly increased carbon emissions from gaining a more detailed power profile. One explanation we suggest is the same as for the increased emissions in the scheduler comparison before: scheduling long jobs with long deadlines lead to timeout due to the increased complexity from having different phases. For each such job, the same configuration with a constant assumption would be able to find an optimal schedule, while a non-constant assumption would not.

Even for jobs that found an optimal schedule, the savings are minimal, if any. This is likely due to the design of the evaluation: with a resolution of an hour for the carbon emissions and the phases sharing that resolution, the averaged power is likely too close on this scale. Another reason could be that the chosen carbon trace exhibits a very *wavy* pattern. Unlike the Australian trace used in Section 6.3, the German trace used for this evaluation had fewer sudden shifts in carbon intensity, likely making scheduling according to phases less worthwhile.

To summarize, the effects of having heterogeneous power in simulated jobs could not be proven using the proposed evaluation.

Effect of Waiting Times Figure 7.5 shows the carbon emissions for each job across different waiting times. We decided to remove the heterogeneous jobs from these plots, as the chosen parameters had little effect on emissions, instead only causing timeouts. Across the remaining parameters, the relative carbon savings are very close, showing up as a single dot. In two instances, there were changes, however. For longer jobs with a waiting time of 2 days, having a startup time of half an hour decreased relative savings.

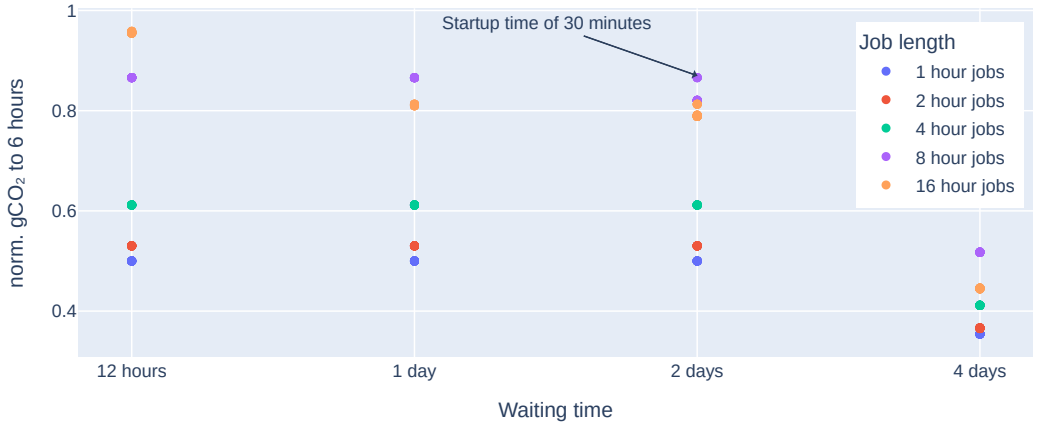

Figure 7.5: Normalized carbon emissions with the baseline being a wait time of 6 hours for each constant-power job. There are small difference for jobs with a startup length of 30 minutes but for the most part, there is no difference across the remaining parameters.

Jobs shorter than 16 hours had stagnating savings beyond a waiting time of 12 hours. It is likely, that this is a circumstance of the chosen carbon trace. The best period is around noon on July 1 (see the bottom graph of Figure 7.2), which also fits shorter job lengths. With a 4 days waiting time, shorter jobs could then again be scheduled on the new lowest carbon emissions period.

Summary As for our given research questions, answering them under the current evaluation cannot be done definitively. We found that, within a summer week in Germany, longer jobs have a higher potential for carbon savings using a suspend & resume strategy, even if resuming a job carries overhead. Increasing the deadline for a given job also displays a trend of increasing carbon reduction. Very long jobs, in this case 16 hours, exhibit higher saving potential from waiting times. The effect of workload heterogeneity is unclear, as the chosen phase scenarios, of (short) alternating phases, likely do not have significant saving potential inherently.

On the other side, we have gained insights into the limitations of suspend & resume scheduling in Carbs. Specifically, scheduling certain workloads - those with many alternating phases, long runtimes, and deadlines - has long runtimes. In Section 8, we will give ideas for a better-suited evaluation.

8 Discussion

For this, we will discuss each contribution, and draw a conclusion in the end.

8.1 Stawp

With Stawp, we propose a high-level time-to-power model for workloads. In essence, it models a workload as having startup phases and work phases. Each of these then have a constant power attached to it. This presents a superset of the workload models used in prior literature in this field. Using a constant per phase is a big simplification, however. The main motivation for this was to reduce complexity in the LP suspend & resume scheduler. We also argue that in the context of the low carbon-emission resolutions, this is appropriate.

A drawback of Stawp is that power used outside the job is not captured at all. In a real-world cloud scenario, there will be e.g. be VM startup times [17]. Also, we assume workloads to be executed in an isolated matter. Thus, each workload under Stawp increases power and carbon emissions linearly. On real-world hardware however, servers have a baseline energy demand. Sharing resources and increasing utilization of a server increases energy efficiency [2]. Performance impacts due to shared hardware is also not part of Stawp.

In the context of long workloads and parallel execution, the clean-cut phases we observed on a single machine and a short job may not hold in HPC environments.

8.2 Carbs

We iterated on an existing testbed, GAIA. In comparison to prior work, workloads in Carbs now have heterogeneous phases and their startup times are part of the scheduling. We have tried to evaluate this changed assumption for different startup costs, different phases, waiting times, and job lengths. The trend that allowing resumeable jobs to be deferred for longer increases carbon savings [16, 6, 5], appeared in our results as well. We were also able to observe the findings by Sukprasert et al. [12] that a suspend & resume strategy benefits very long jobs more.

Wholly unexplored are the effects job heterogeneity that we added. The reason being that our chosen scenarios:

- Had phases of relatively short length. It is likely that longer phases result in more pronounced results.
- Used too many phases. We chose to repeat low and high phases until the given job length is met, meaning that the amount of phases increased linearly. As suspend

& resume scheduling works better for very long jobs, they could not be scheduled within a time limit of 20 minutes.

A drawback of the LP implementation is that computation time is dependent on the shortest phase in Stawp (see Section 6.3). Thus, if startup times are short, they may need to be removed in favor of runtimes. The OPR approach described in Section 3.2, which assumes startup to have a cost but no length, does not have this problem.

We want to propose a more fit approach to the evaluation: First, a literature review of long-running workloads with long phases is necessary. Additionally, the power measurement capabilities of cluster nodes we described in Section 4.1 could be used to measure larger ML models than we were able to execute on our private hardware. Given these workloads, the evaluation is then done under different carbon traces to remove bias. In our evaluation, the days 4 and following had an influx of wind power which dominated the carbon savings for the longest waiting time.

In our case, due to the way each job was generated and indexed, the hardest problems were executed last. A better approach would be to compute the complex problems first to check whether time limits are hit, or to run them in a random order. Running a sample evaluation for select parameters should also have been done. As of now, the impact of our used parameters on the resulting LP complexity is not entirely clear. A runtime analysis on Carbs' LP scheduler could be a guide on what can be scheduled.

8.3 Conclusion

8.4 Future Work

There are several avenues for future work. Regarding Carbs, our suspend & resume implementation is not yet able to make use of Stawp's checkpoint annotations. Future work includes evaluating carbon-aware scheduling when jobs can only be checkpoint at some points.

Our LP scheduler showed long runtimes for more complex scheduling problems. Future work could deal with either improving model performance [Web [14, 34]] or evaluating whether a break-even point in scheduling-time and carbon-saving exists.

We found that suspend & resume scheduling works best for long jobs. Future work could investigate which real-world workloads exhibit power heterogeneity. Their saving potential our workload model could then be analyzed.

Instead of only testing predefined workloads, our LP scheduler could be combined with current research in the field of workload analysis. Köhler et al. [8] classify workloads based on power signatures. On live systems, such an approach could be used to opt-in to a suspend & resume strategy. We found that short jobs exhibit little potential carbon savings. If a workload then runs longer, classifying them as suspendible could lead to hybrid scheduling strategies.

Literature

[1] Anders Aaen Springborg, Michele Albano, and Samuel Xavier-de Souza. "Automatic Energy-Efficient Job Scheduling in HPC: A Novel SLURM Plugin Approach". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pages 1831–1838. ISBN: 9798400707858. DOI: 10.1145/3624062.3624265 (cited on page 9).

[2] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (Dec. 2007). Conference Name: Computer, pages 33–37. ISSN: 1558-0814. DOI: 10.1109/MC.2007.443 (cited on page 43).

[0] *EDF cuts nuclear production in reaction to soaring temperatures*. en. URL: euronews.com/business/2024/08/14/edf-cuts-nuclear-production-in-reaction-to-soaring-temperatures (cited on page 4).

[4] Gilbert Fridgen, Marc-Fabian Körner, Steffen Walters, and Martin Weibelzahl. "Not All Doom and Gloom: How Energy-Intensive and Temporally Flexible Data Center Applications May Actually Promote Renewable Energy Sources". en. In: *Business & Information Systems Engineering* 63.3 (June 2021), pages 243–256. ISSN: 1867-0202. DOI: 10.1007/s12599-021-00686-z (cited on page 5).

[5] Walid A. Hanafy, Roozbeh Bostandoost, Noman Bashir, David Irwin, Mohammad Hajiesmaili, and Prashant Shenoy. "The War of the Efficiencies: Understanding the Tension between Carbon and Energy Optimization". en. In: *Proceedings of the 2nd Workshop on Sustainable Computer Systems*. Boston MA USA: ACM, July 2023, pages 1–7. ISBN: 9798400702426. DOI: 10.1145/3604930.3605709 (cited on pages 8, 43).

[6] Walid A. Hanafy, Qianlin Liang, Noman Bashir, Abel Souza, David Irwin, and Prashant Shenoy. "Going Green for Less Green: Optimizing the Cost of Reducing Cloud Carbon Emissions". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Volume 3. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pages 479–496. ISBN: 9798400703867. DOI: 10.1145/3620666.3651374 (cited on pages 23, 43).

[7] Sven Köhler, Benedict Herzog, Timo Hönig, Lukas Wenzel, Max Plauth, Jörg Nolte, Andreas Polze, and Wolfgang Schröder-Preikschat. "Pinpoint the Joules: Unifying Runtime-Support for Energy Measurements on Heterogeneous Systems". In: *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. Nov. 2020, pages 31–40. DOI: 10.1109/ROSS51935.2020.00009 (cited on page 10).

[8] Sven Köhler, Lukas Wenzel, Max Plauth, Pawel Böning, Philipp Gampe, Leonard Geier, and Andreas Polze. "Recognizing HPC Workloads Based on Power Draw Signatures". In: *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*. Nov. 2021, pages 278–284. DOI: 10.1109/CANDARW53999.2021.00053 (cited on page 44).

[9] Adam Lechowicz, Nicolas Christianson, Jinhang Zuo, Noman Bashir, Mohammad Hajiesmaili, Adam Wierman, and Prashant Shenoy. "The Online Pause and Resume Problem: Optimal Algorithms and An Application to Carbon-Aware Load Shifting". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7.3 (Dec. 2023). arXiv:2303.17551 [cs], pages 1–32. ISSN: 2476-1249. DOI: 10.1145/3626776 (cited on page 8).

[10] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. *Green AI*. arXiv:1907.10597 [cs, stat]. Aug. 2019. DOI: 10.48550/arXiv.1907.10597 (cited on page 2).

[11] Abel Souza, Shruti Jasoria, Basundhara Chakrabarty, Alexander Bridgwater, Axel Lundberg, Filip Skogh, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. "CASPER: Carbon-Aware Scheduling and Provisioning for Distributed Web Services". In: *Proceedings of the 14th International Green and Sustainable Computing Conference*. IGSC '23. New York, NY, USA: Association for Computing Machinery, May 2024, pages 67–73. ISBN: 9798400716690. DOI: 10.1145/3634769.3634812 (cited on page 6).

[12] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David Irwin, and Prashant Shenoy. "On the Limitations of Carbon-Aware Temporal and Spatial Workload Shifting in the Cloud". en. In: (2024) (cited on pages 5, 8, 43).

[13] Andrew S. Tanenbaum and Albert Woodhull. *Operating systems: design and implementation: [the MINIX book]*. en. 3. ed. The MINIX book. Upper Saddle River, NJ: Pearson Prentice Hall, 2006. ISBN: 978-0-13-142938-3 978-0-13-505376-8 978-0-13-142987-1 (cited on page 5).

[0] John Thiede, Noman Bashir, David Irwin, and Prashant Shenoy. "Carbon Containers: A System-level Facility for Managing Application-level Carbon Emissions". In: *Proceedings of the 2023 ACM Symposium on Cloud Computing*. SoCC '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pages 17–31. ISBN: 9798400703874. DOI: 10.1145/3620678.3624644 (cited on page 1).

[15] Philipp Wiesner, Ilja Behnke, Paul Kilian, Marvin Steinke, and Odej Kao. "Vessim: A Testbed for Carbon-Aware Applications and Systems". en. In: (2024) (cited on page 9).

[16] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. "Let's Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud". In: *Proceedings of the 22nd International Middleware Conference*. arXiv:2110.13234 [cs]. Dec. 2021, pages 260–272. DOI: 10.1145/3464298.3493399 (cited on pages 4, 8, 10, 14, 38, 43).

[17] Chen Zheng and Jianfeng Zhan, editors. *Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium, Bench 2018, Seattle, WA, USA, December 10-13, 2018, Revised Selected Papers*. en. Volume 11459. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-32812-2 978-3-030-32813-9. DOI: 10.1007/978-3-030-32813-9 (cited on page 43).

[18] Íñigo Goiri, Kien Le, Md. E. Haque, Ryan Beauchea, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. "GreenSlot: scheduling energy consumption in green datacenters". en. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle Washington: ACM, Nov. 2011, pages 1–11. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063411 (cited on pages 9, 21).

Web Sources

[19] *Advanced API Performance: SetStablePowerState*. en-US. June 2022. URL: developer.nvidia.com/blog/advanced-api-performance-setstablepowerstate/ (cited on page 12).

[20] *Agora Energiewende*. en-US. URL: www.agora-energiewende.org/ (cited on pages 1, 4).

[21] APaul. *Answer to "slurmd: error: Couldn't find the specified plugin name for cgroup/v2 looking at all files"*. Oct. 2022. URL: stackoverflow.com/a/74211989 (cited on page 22).

[0] Aurelien Breeden. "'Most Severe' Drought Grips France as Extreme Heat Persists in Europe". en-US. In: *The New York Times* (Aug. 2022). ISSN: 0362-4331. URL: nytimes.com/2022/08/05/world/europe/france-drought-europe-heat.html (cited on page 4).

[22] *Build next generation sustainable products | Electricity Maps*. en. URL: www.electricitymaps.com/ (cited on pages 3 sq.).

[23] *Connected Papers | Find and explore academic papers*. URL: www.connectedpapers.com/ (cited on page 7).

[24] *Dein Stromvertrag für das digitale Zeitalter Tibber*. de-DE. URL: tibber.com/de (cited on page 5).

[25] *distilbert/distilroberta-base · Hugging Face*. May 2024. URL: huggingface.co/distilbert/distilroberta-base (cited on page 10).

[3] *Energieverbrauch von Rechenzentren*. de. URL: bundestag.de/resource/blob/863850/423c11968fcb5c9995e9ef9090edf9e6/WD-8-070-21-pdf-data.pdf (cited on page 1).

[26] *Getting started with Slurm - HPI | SCORE | Labs*. URL: score.hpi.uni-potsdam.de/docs/Slurm/index.html (cited on page 37).

[27] *Heatwave forces French nuclear power plants to limit energy output*. en. July 2023. URL: www.euronews.com/green/2023/07/13/frances-nuclear-power-stations-to-limit-energy-output-due-to-high-river-temperatures (cited on page 4).

[28] *HotCarbon*. en. URL: hotcarbon.org// (cited on page 7).

[29] *How to use the Slurm simulator as a development and testing environment - HPCKP*. URL: hpckp.org/articles/how-to-use-the-Slurm-simulator-as-a-development-and-testing-environment/ (cited on pages 22 sq.).

[30] *HPI Data Center*. URL: hpi.de/forschung/hpi-datacenter/ (cited on page 21).

[31] *Installation Guide*. en. URL: github.com/dun/munge/wiki/Installation-Guide (cited on page 21).

[32] *IPCC Report 2018, Chapter 7*. en. URL: https://www.ipcc.ch/site/assets/uploads/2018/02/ipcc_wg3_ar5_chapter7.pdf (cited on page 5).

[33] *Marginal emissions: what they are, and when to use them*. en. URL: https://www.electricitymaps.com/blog/marginal-emissions-what-they-are-and-when-to-use-them (cited on page 4).

[34] *Optimizing Gurobi Solving*. en-US. URL: assets.gurobi.com/pdfs/webinar-parallel-and-distributed-optimization-english.pdf (cited on page 44).

[35] *Regulation - 2016/679 - EN - gdpr - EUR-Lex*. en. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (cited on page 6).

[36] *SCORE Slack*. URL: hpi-scientificcompute.slack.com/archives/C05DCH62XTJ/p1709634180913929?thread_ts=1709633994.587459&cid=C05DCH62XTJ (cited on page 37).

[37] *Slurm Scheduler Plugin API 20.11.9*. URL: https://slurm.schedmd.com/archive/slurm-20.11.9/schedplugins.html (cited on page 23).

[38] *Slurm Scheduler Plugin API 23.11.10*. URL: https://slurm.schedmd.com/archive/slurm-23.11.10/plugins.html (cited on page 23).

[39] *Slurm Workload Manager - Adding Files or Plugins to Slurm*. URL: slurm.schedmd.com/add.html (cited on page 22).

[40] *Slurm Workload Manager - Overview*. URL: slurm.schedmd.com/overview.html (cited on page 21).

[41] *slurm/src/slurmctld/controller.c*. en. URL: https://github.com/SchedMD/slurm/blob/3e71fef79ca203b53e939c12a2e58c653f48c126/src/slurmctld/controller.c (cited on page 22).

[42] *The Leader in Decision Intelligence Technology*. en-US. URL: https://www.gurobi.com/ (cited on page 29).

[14] *Threads and Gurobi*. en-US. URL: www.gurobi.com/documentation/current/refman/threads.html (cited on page 44).

[43] *Unable to change power limit with nvidia-smi · Issue #483 · NVIDIA/open-gpu-kernel-modules*. en. URL: github.com/NVIDIA/open-gpu-kernel-modules/issues/483 (cited on page 12).

[44] *U.S. levelized energy costs by source 2023*. en. URL: https://www.statista.com/statistics/493797/estimated-levelized-cost-of-energy-generation-in-the-us-by-technology/ (cited on page 5).

[45] *WattTime*. en-US. URL: watttime.org/ (cited on page 4).

.1 Repository

Make a tagged release on github and link that

.2 Results of the Structured Literature Review