A parameterized 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 parameterized 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 parameterized 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

Gutachter:in: Prof. Dr. Jack Alsohere
University of San Serife
Facultiy of Computer Doings

Amelia van der Beenherelong
ACME Cooperation

Datum der Einreichung: 6. Oktober 2024

Zusammenfassung

ummarize thesis

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 carbon emissions is prevalent. One area of interest is carbon produced via electrical power used in data centers. However, not all power is produced equally: while a data center may source its power from the public grid, the public grid 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 solar and wind. The latter, renewables and especially solar, follow 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 then comes from solar, reducing the overall carbon intensity of the grid. This can be used for *carbon aware scheduling*, by planning work in datacenters to be executed during such low intensity times, the overall carbon can be reduced.

Data centers are currently projected to experience exponential growth in their power requirements, largely fueled by pushes in AI.[4]

Current work on carbon aware scheduling includes shifting jobs temporally and spatially. One common theme among them, 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 for e.g. I/O, this is not reflected in literature. Another common 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. This generally assumes, that resuming a job carries no overhead.

In this work, we will improve upon the homogeneous, no overhead workload model used in literature, by measuring an AI program and deferring a new model upon that.

The research question will be the following:

1. How high are 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?

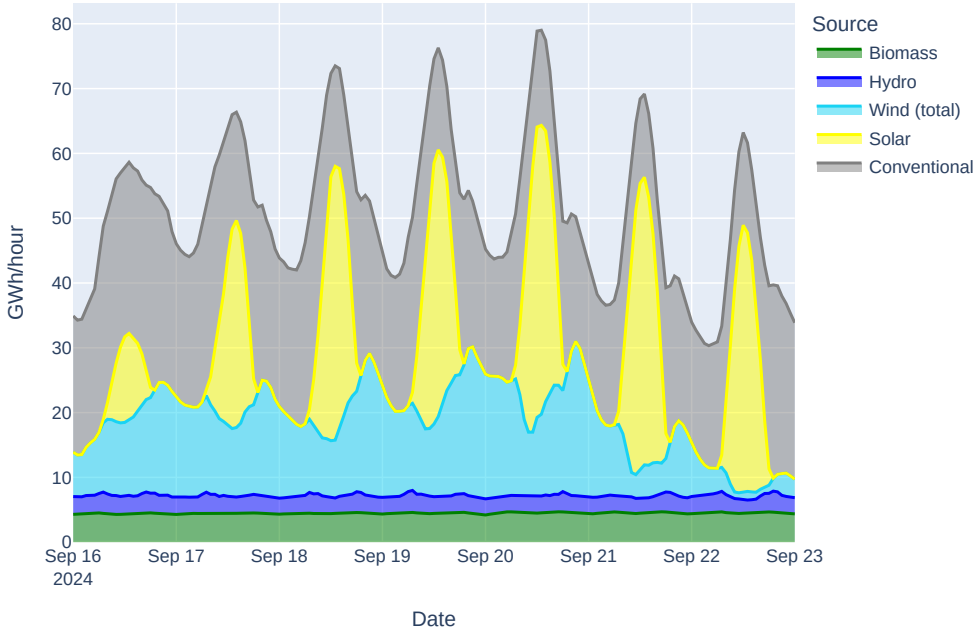

Figure 1.1: Mix of energy production in Germany between the 16th to 22nd of September (Monday to Sunday). Data provided by `agora-energiewende.org`

2 Background

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

The Composition of the Public Grid As outlined in the previous chapter, there is a diurnal rhythm to the renewable energy production. In addition to that, there is also a seasonal effect on the energy grid composition: 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.

The demand for power also has an effect on the overall composition. During times of low demand, less conventional power needs to be produced and renewables have a higher share as well. This can be seen in the seasonal Figure: during Christmas holidays in December, people spend more time at home, requiring less additional conventional energy (e.g. for industrial consumers, as people also work less).

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), but also takes effect in the composition of production capabilities. Some countries may have less investment in renewables or may use nuclear power plants, which are also deemed low-carbon but have less of a seasonal or diurnal rhythm (if they are not limited by lack of cooling water[1]).

Needs citation

Thus, national public grids have a certain affinity for carbon aware scheduling[6]. Generally, the higher the solar capacities, the higher the potential carbon saving from such an approach.

Needs to be integrated better xd

Power Grid Signals Carbon-aware scheduling commonly uses one of two possible metrics or signals: the *average emissions* are the metric describing the amount of carbon per unit of energy. These are calculated using a weighted average of all power supplies at a point int time.

Another metric is the *marginal emissions*, answering the question of "if more energy is used at this point in time, how much carbon does that cost?".

The marginal emissions are closely related to the way energy producers in- or decrease production. In order to reduce electricity prices, additional demand on the grid is dispatched to the cheapest power plant with remaining capacity. These power plants, also called *marginal power plants*, have a correlation to the carbon emissions. With the cheapest power coming from renewables, additional demand is supplied with more expensive, more carbon intensive, conventional power plants.

[1] https://www.euronews.com/green/2023/07/13/frances-nuclear-power-stations-to-limit\protect\penalty\z@-energy-output-due-to-high-river-temperatures

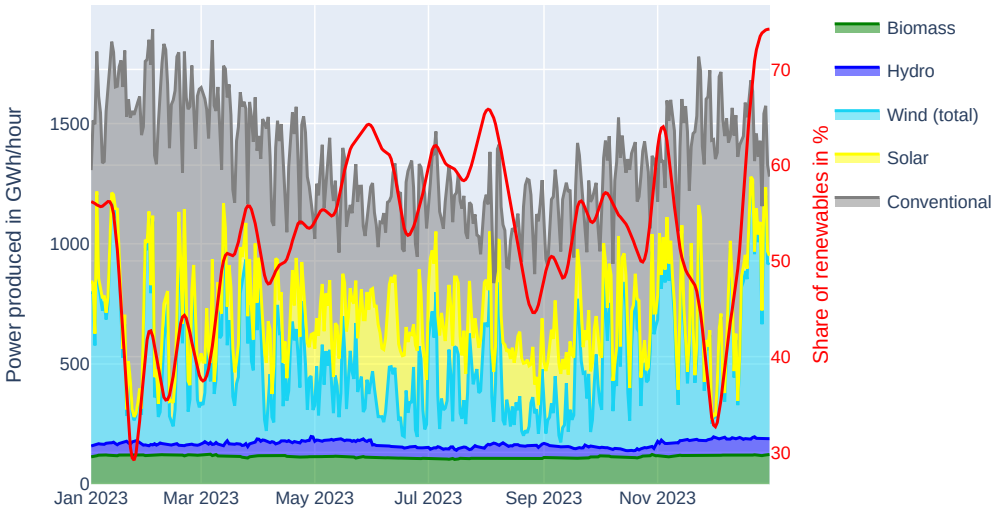

Figure 2.1: Mix of energy production in Germany in 2023. Data provided by `agora-energiewende.org`

Two providers of signal data are *Electricitymaps*[2] and *WattTime*?[3] Electricitymaps has a free access API for historical and current average emissions. Both providers only offer the marginal signal commercially, however.

Going further, the average signal will be used. Note should be taken that current literature is still split on which signal is best. A recently released and aptly named paper by Sukprasert et. al **[sukprasert_limitations_2024]** may be read for further information on grid signals, as well as arguments for either signal.

Out of these, reasons for using the average signal is, for example, *curtailment*. Curtailment encompasses any methods that reduce the amount of produced renewable energy. As the power grid always needs to have a balance between demand and production, curtailment methods such as turning of 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, as demand for energy would increase at those times. Another argument made, for example by Fridgen et al. **[fridgen_not_2021]**, 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, there are also some contracts that use dynamic pricing[4]

Need a citation here!

Workloads in a Datacenter According to Tanenbaum and Woodhull[5], 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 these workloads**[sukprasert_limitations_2024]**. 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 interacts with the system, and thus expects quick responses to their inputs. An example of an interactive job are web requests, which should be answered quickly inherently. The last environment are *real time* systems. There, deadlines and predictability dictate how a scheduler operate.

For this work's topic, (temporal) carbon-aware scheduling, only batch systems will be looked at as they allow more freedom to the times of scheduled jobs.

Power Intake of a Computer As there will be power measurements in 4.1, some basic understanding of energy and power used for computation will be provided:

- I could mostly borrow from the EBRH slides; there is some base power needed that is correlated to the hardware (dynamic and static energy)
- this also depends on frequency (which is why later we set our CPU frequency to some hard coded value [or do not do that in the case of my GPU lol])
- basically, use this paragraph to outline all things we take care of in my power measurements

[2] `electricitymaps.com`
[3] `watttime.org`
[4] One example for dynamic pricing is *Tibber*: `tibber.com`

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' *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.

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. Portability is an issue however, unlike operating-system supported tools such as perf, a measurement device needs more effort to be used on another system (or be entirely not useable, for example when such devices are only rated for a certain power level).

3 State of the Art

This chapter will outline the current state-of-the-art research surrounding carbon aware (temporal) scheduling, as well as the approach taken for determining that.

Systematic Approach We used the following system for finding related work to my topic. 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, I could then create 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 fit.

These are then be entered into a spreadsheet: for each query, 5 paper titles. Additionally, we further explored papers through *connected papers*[5] or looking up individual authors on a subjective basis. The papers from the 2023 and 2024 HotCarbon Workshops[6] also proved to be related.

We then sort each paper into one of the following category:

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

Results The results are exported as `literature_study.csv` in the appendix, according to its evaluation using the `literature_study_evaluation` Jupyter Notebook, 145

[5] `https://www.connectedpapers.com/`
[6] `https://hotcarbon.org/`

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

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

GreenSlot: Scheduling energy consumption in green datacenters[7] According to the analyzed papers, this seems to be the first paper that deals with carbon aware scheduling by implementing it as a *Slurm* plugin (See section 5.1.1 for further information on Slurm and such plugins) In contrast to our scenario, where we try to optimize carbon emissions via the public electricity grid, GreenSlot is about datacenters 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.

The War of the Efficiencies: the Tension between Carbon and Energy Optimization [1] This paper outlines the different ways of carbon aware computing. Among those are *temporal shifting*, the idea that jobs can be executed later when energy is more carbon efficient, is also the main idea for my work. They also use *spatial shifting*, moving jobs across the globe to areas where higher carbon efficiency is possible. *Resource scaling* uses dynamic amounts of hardware according to carbon emissions. In the end, *rate shifting* is the idea to also scale hardware frequencies. During carbon-efficient times, CPU speeds are increased, leading to faster processing speeds and more energy usage.

All of these techniques are then tested under various parameters. My work will only make use of the temporal shifting, and abstract away the other methods of further carbon savings.

Let's wait awhile: how temporal workload shifting can reduce carbon emissions in the cloud This paper will be built upon in my work. The authors [6] use a simulation approach, to simulate temporal shifting. Their workload model consists of known length jobs that can use suspend & resume to be executed at different time slices. They further use different traces and test these traces under the assumption of different job-deadlines, meaning that each job has to be completed by a certain timeframe, and also different regions of the world as described in the background part. Their main takeaways are that increased deadlines lead to reduced carbon emissions, but that this effect also has diminishing returns. They also deduced that regions such as California, with high amounts of solar power, have higher potential for carbon-savings in comparison to nuclear-heavy regions such as France.

Weigh this sentences with the GAIA paper as that is the one I am actually forking from

Think about linking this to the earlier part

On the Limitations of Carbon-Aware Temporal and Spatial Workload Shifting in the Cloud In a very recent paper, Sukprasert et al. [**sukprasert_limitations_2024**]

Should write about this

4 Defining a New Workload model

In order to improve on the homogenous, no startup-cost, workload model used in related work, we will first conduct power measurements on a sample program, and then use that information for the new model.

4.1 Power Measurements on Machine Learning Jobs

The test environment The experiments were run on my personal computer, the components of that are listed via the *hwlist* tool, with some columns and rows being redacted for brevity:

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

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 Like outlined in Chapter 2, multiple measurement options exists. As the HPI has the *Microchip MCP39F511N Power Monitor (henceforth called MCP)* on-site, they 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.

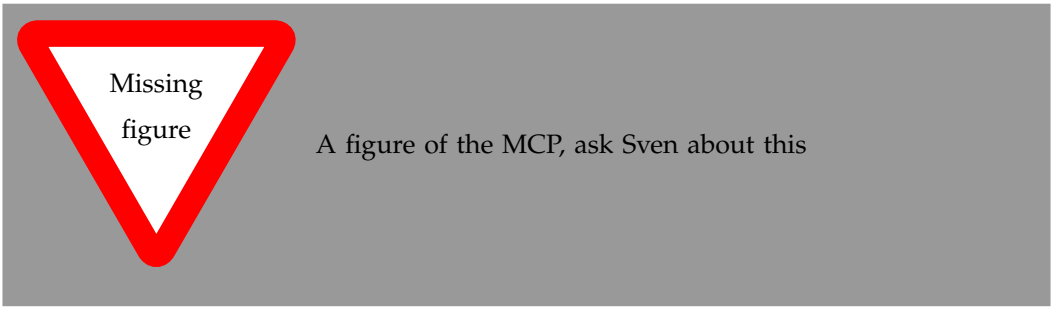

Figure 4.1: The MCP

The software used for reading out MCP data is *pinpoint*[3].

I could include an extra paragraph on why the MCP is cool, and what it does differently, perhaps. Was there anything cool? I vaguely remember some measurement devices having two capacitors to more accurately determine usage

Measured Program Machine learning (ML) was used as the main motivation for suspend & resume scheduling in the related works[6] and thus was also chosen by me to be measured and modeled.

The concrete model and framework is secondary for our measurement. In our case, a small model[7] is chosen in order to have fewer data points for processing, as well as faster iterations on the measurement script.

There is a vast amount of machine learning frameworks. For a high-level model, the feature set of the framework only needed to support checkpointing, resuming, and 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 not much bias towards any framework, huggingface was chosen. The huggingface trainer supports callbacks, we modified a given 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 start to finish
2. Run it partially, checkpointing after some step, sleeping, resuming from that step
3. Run it partially, checkpointing after some step but aborting before the next checkpoint. Then resume as above.
4. Run only the startup phase up until the ML would start
5. Do nothing, measure the system at rest

Experiment 1 gives a baseline for what the job looks like without suspend & resume. Number 2 and 3 would be used to determine the overhead of checkpointing the job. 4 would be used to validate the other ones. 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 experiment are subject to noise. For example, *Dynamic frequency and voltage scaling (DFVS)*, the OS technique of increasing CPU "speeds" according to work load add power in an uncontrolled way. Also, background tasks may happen *randomly*, increasing power usage.

[7] `huggingface.co/distilbert/distilroberta-base`

Thus, for the testing, any foreground apps are closed. We also used the Linux tool `cpupower`, as shown in 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, the frequency should also be similarly set to a defined value. NVIDIA provides guide on how to do so.[8] 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 drivers issue preventing this?[9] 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.

Maybe add a prove that it could work.

During 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 uniform load of any libraries.

Conducting each experiment Each experiment was re-run 10 times. 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 would be used. This leads to a period of 30 seconds being measured around the program exection. Plotting these additional time frames gives a quick visual indicator whether experiments are sufficiently isolated from each other, ergo when the power draw is at the baseline as the actual program starts.

As there is some data being downloaded and persisted during the execution of the ML, before each run, that data is removed,

Collected data For each experiment, a named and timestamped folder is created in the `/power-measurements` folder of my repository. Each folder then holds a `.csv` with pinpoint's timestamped power measurements. The added timestamped-logging is saved

[8] `developer.nvidia.com/blog/advanced-api-performance-setstablepowerstate`
[9] `github.com/NVIDIA/open-gpu-kernel-modules/issues/483`

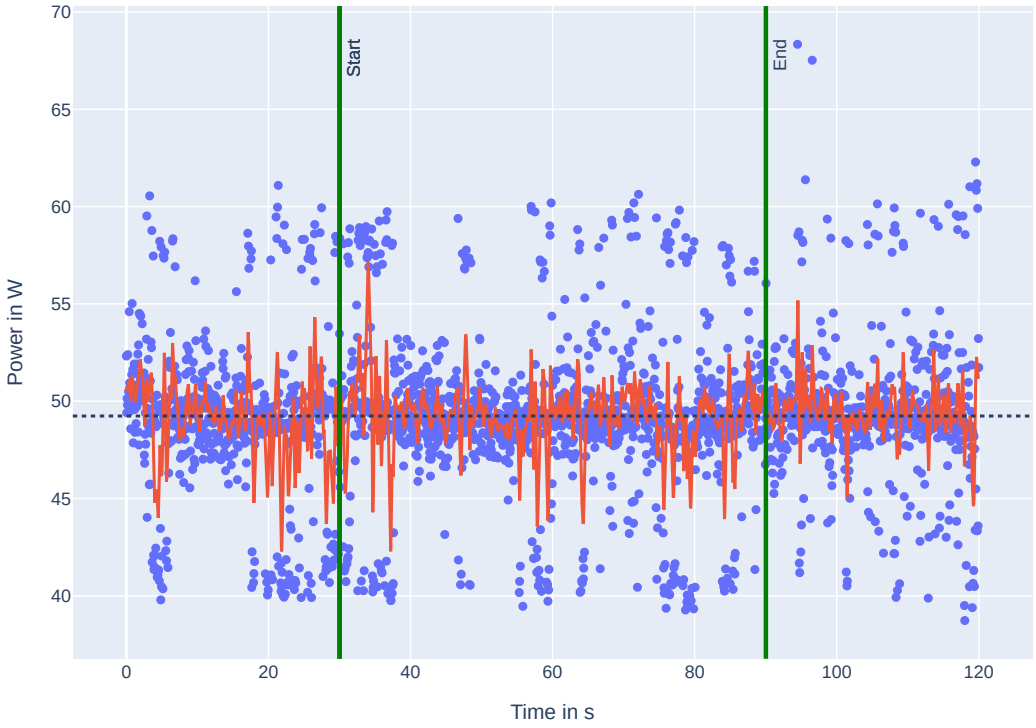

Figure 4.2: Sample run of the baseline experiment. The dotted line is the global average. The red line is a smoothed Gaussian trend line.

into another `.csv`. On the way to the final measurements, we plot each experiment early and visually spot if there are any obvious errors or mistakes.

Determining the baseline power draw Beginning with the most exiting experiment, determining the baseline and testing the amount of the underlying environment. One sample run is shown in Figure 4.2. The blue dots in figure represent each data point. The red line is a smoothed Gaussian trend line with $\sigma = 2$. Dark-green vertical lines are the logged or derived "events" for each run. In this case, nothing happens, so it is only the start and end of `sleep 120`. Notice how the trace starts 30 seconds before the start and continues for another 30 seconds because of the aforementioned `pinpoint` feature. Going further, these additional measurements will be redacted for brevity.

Across all 10 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 power draws of the other experiments. The standard deviation should give a broad idea of how much noise is in the system environment.

The non-interrupted run For the non-interrupted machine learning run, a sample run is provided in Figure 4.3. Figure 4.4 shows the stacked trend lines of the 10 different runs.

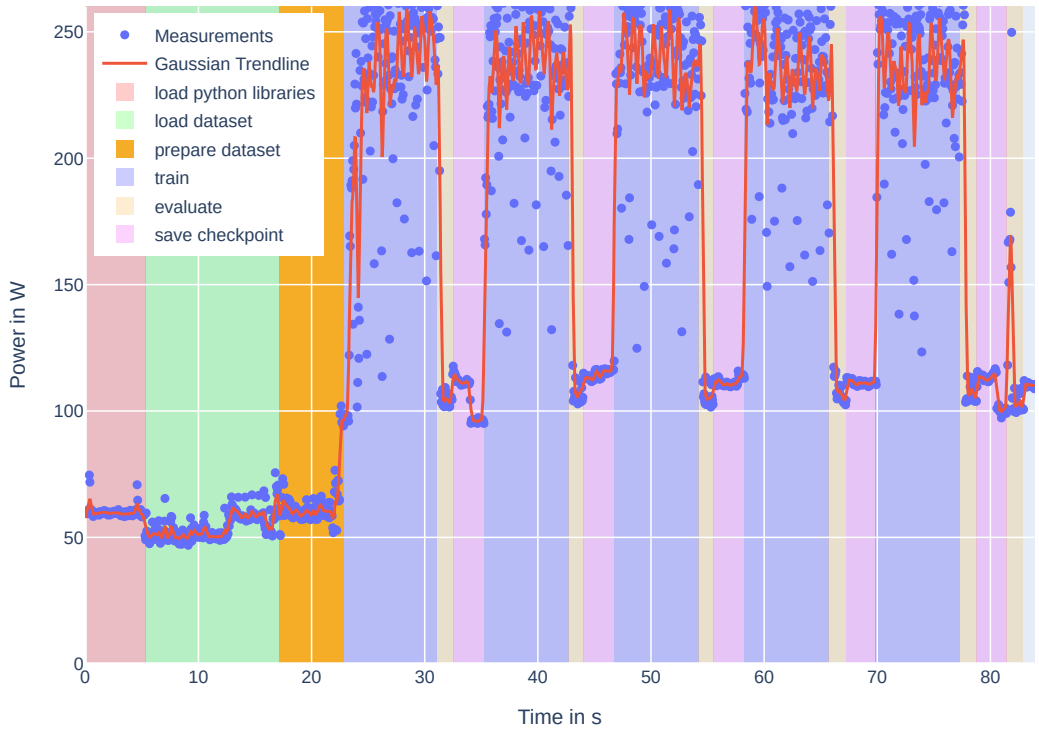

Figure 4.3: Sample run of the full run experiment, the background indicates the phase as determined by the added logging

For simplicity's sake, we refrained from doing more elaborate statistical analysis on the different runs as the visual check of them being very similar seemed enough.

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 loading data to disc.
2. There are periodical work-phases; a high-power training phase would be 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.

Uhm, where did my evaluation phase markers go?? They are mushed together as they share a label

This already shows, that improvements upon the constant-power model used in [6] are possible. For example, in this case the start-up phase has a much lower power-draw than the work-phase.

Results of the checkpoint and restore experiment Similarly to before, results will be discussed using the stacked plots 4.5 and 4.6; each individual run is plotted in the repository, however.

Here we can observe that:

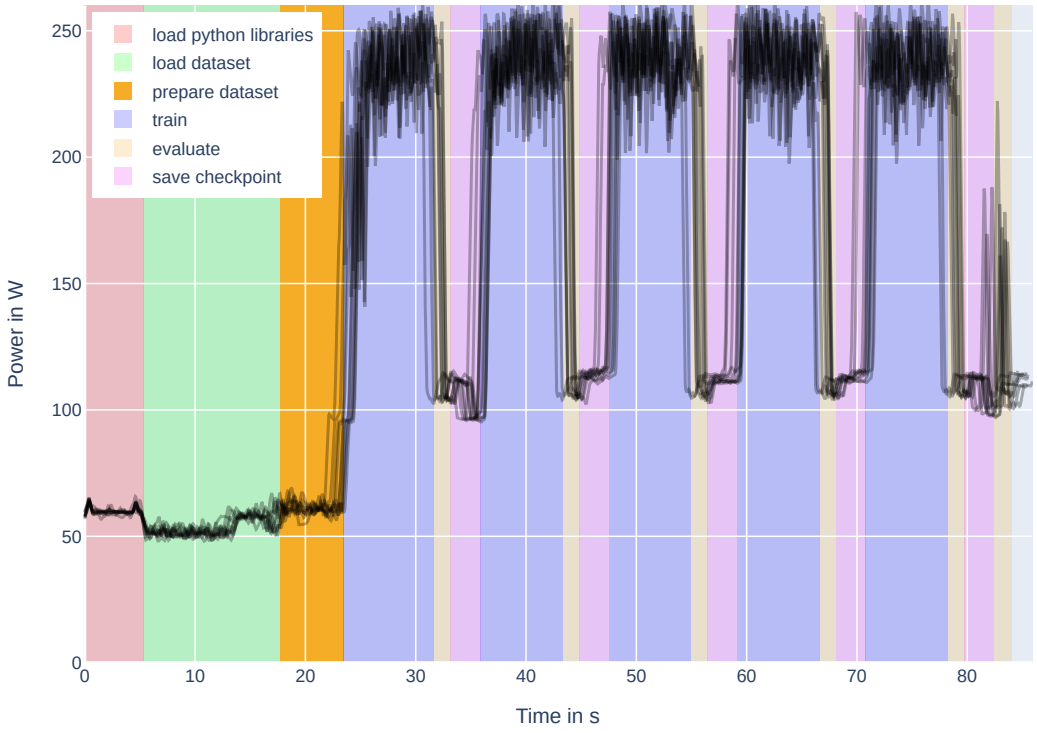

Figure 4.4: All full-run trend lines stacked.

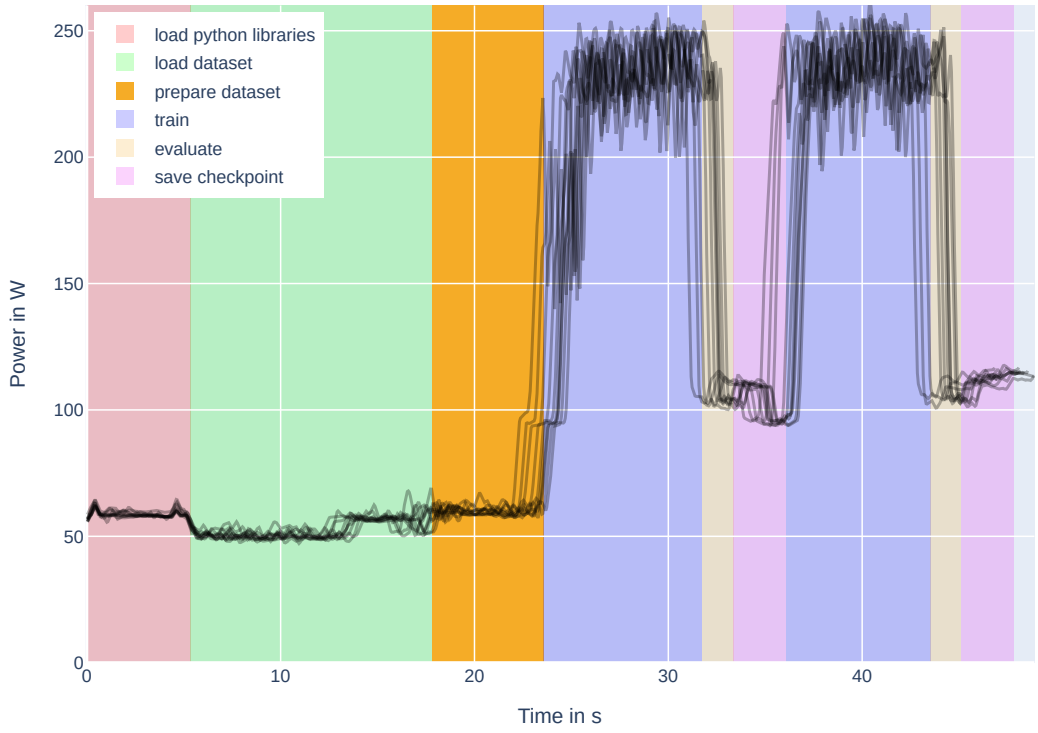

Figure 4.5: Stacked trend lines of the experiment for stopping after the second epoch

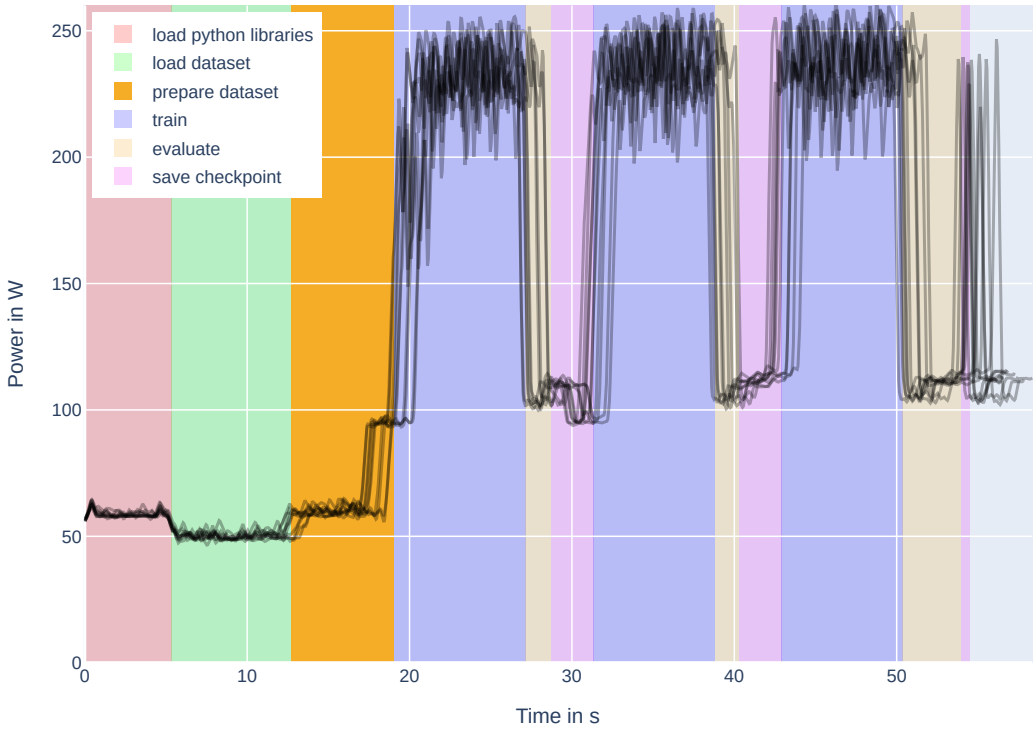

Figure 4.6: Stacked trend lines of the experiment for continuing after the second epoch

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, likely due to not needing to download the dataset again.

Results from checkpoint and resume with abort Unlike the previous experiment, where work is stopped as soon as 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 figure 4.7 and 4.8. 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 checkpointed 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 To support the argument that there is indeed an overhead from checkpointing & resuming, we calculated the energy needed for each experiment:

- unstopped costs 12.97 kJ on average with an std of 0.04
- save+resume costs 13.94 kJ on average with an std of 0.1
- save+abort+resume costs 15.72 kJ on average with an std of 0.07

4.2 Modelling

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.

The parameters for the improved job model are shown in form of the python implementation in Listing 4.2.

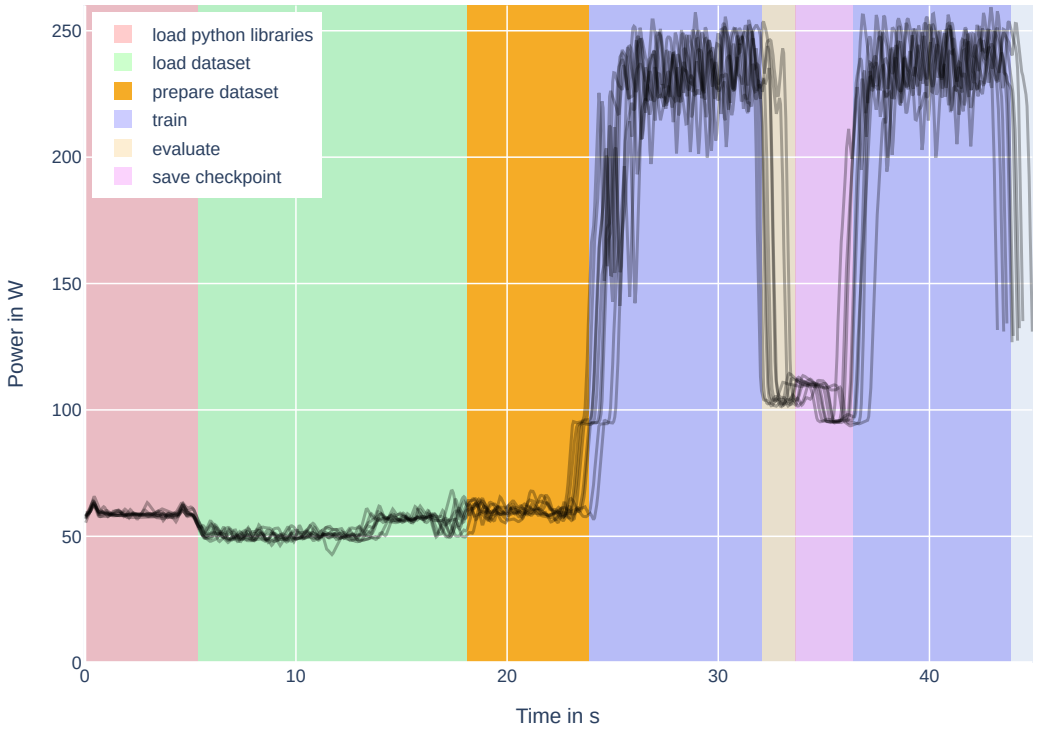

Figure 4.7: Power draws of the ML up until stopping after epoch 2

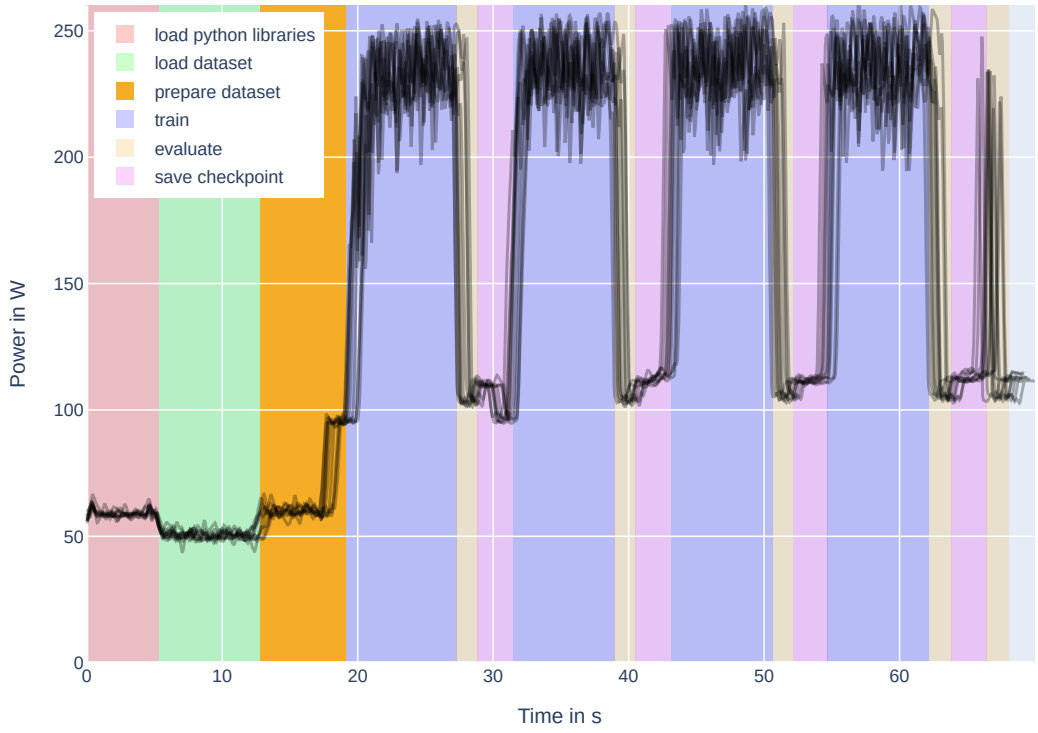

Figure 4.8: Power draws after continuing from the second checkpoint

Listing 4.2: Python Model definition

```
class ModelParameters(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 a constant. Phases can also be named for later reference. These phases essentially define a step function, i.e. piecewise constant function. Unlike a traditional step function, the start- and endpoints of each "piece" would be encoded implicitly by the previous phases. Using these specifications, a simple time-to-power function can be defined, that looks up the input time and traverses 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 5.2.3, having a rather restrictive step function will be of advantage, however.

Add some more explanation to this in that section

The measurements that have been taken can now be fit into this new model. As each measurement-point can be associated to a phase via the aforementioned added logging, the average of each phase-associated measurement is used to determine the model parameters. The durations of the phases are calculated similarly by taking the average time the logging occurred at during the measurements.

Using this strategy on the 10 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 a constant power over its execution. Leaving the startup phases empty creates no overhead on resuming.

Should probably add the phase markers back in

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 would give a measure of the model's accuracy on a short-time (sub-second) scale, the latter would tell the long-time (whole job) scale accuracy of the model.

Each of the 10 runs would be taken as the ground truth while the other 9 would be 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.

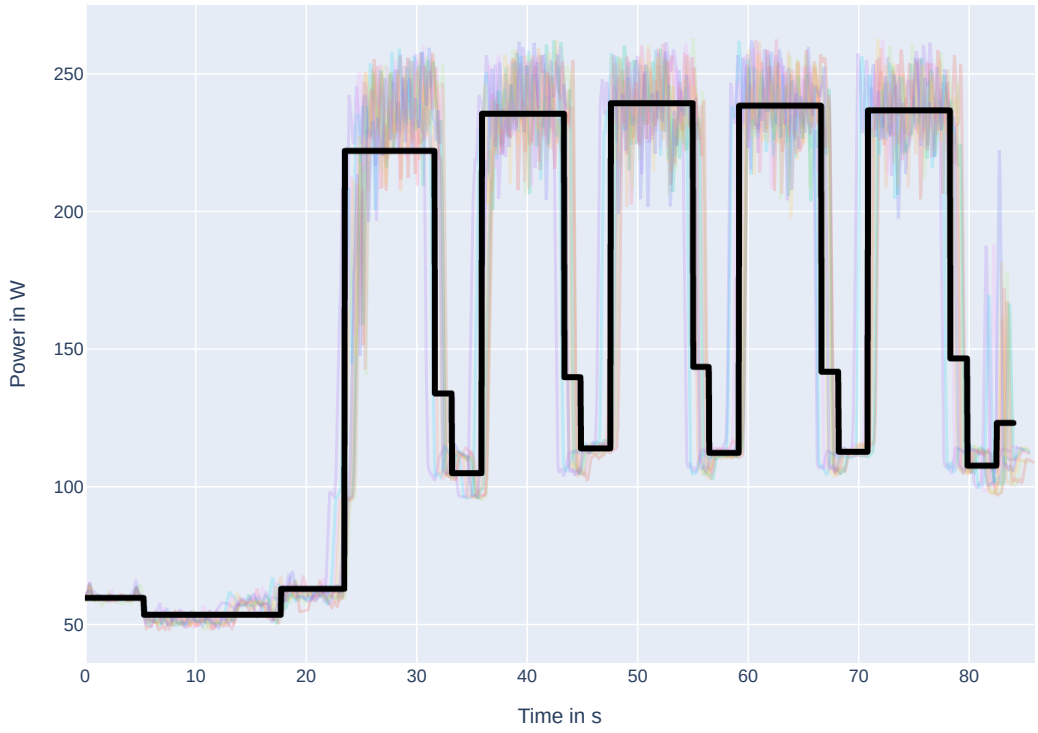

Figure 4.9: Model of roberta.py (black) vs. the measured runs

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 (*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.

5 Implementing Schedulers using the New Model

Now that an improvement on the job model has been made, the question, on how to evaluate the implications of said model, remains.

5.1 Choosing an Implementation Approach

5.1.1 Carbon-aware scheduling via a Slurm plugin

My first idea was to use a non-simulation approach. The HPI's Data Lab[10] 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".[11] The job scheduling part is important here, it also supports a plugin infrastructure that includes scheduling plugins. One of the highlighted papers[7] in the related work section specifically used Slurm for its carbon-aware scheduler implementation and thus seemed like a good starting point for my own work.

Installing Slurm locally For my purposes, a local installation would suffice as I would not need to run heavy workloads but instead just the scheduling part of Slurm. While there is a Slurm `apt get` package for my Ubuntu version, this would not work as any plugins to be included during the Slurm compilation, meaning I would have to do the same.

Slurm's documentation provides some guidelines on how to install Slurm, which I followed. I tried cloning the main branch, but I got stuck during the compilation process. However, using the predefined released versions worked.

Installing `munge`[12] is also necessary, which is used for authentication in Slurm.

One problem arose as I tried to start the `Slurmd`- and `Slurmctld` services. The first one is the worker service that would later execute jobs submitted to Slurm. The latter is the main controller that, for example, schedules jobs on workers. While the command to start them would not fail, upon node inspection via Slurm's `scontrol` command, it would show that all nodes are `DOWN` instantly. | Dealing with Slurm's problems usually leads to inspecting its logs, in my case the logs showed the following:

Listing 5.1: Slurm error logs

[10] `https://hpi.de/forschung/infrastruktur/hpi-data-engineering-lab.html`
[11] `https://Slurm.schedmd.com/overview.html`
[12] `https://github.com/dun/munge/wiki/Installation-Guide`

```
$ 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' `cgroup` feature to manage the submitted job's hardware ressources. The log hints at some problem related to Slurm's usage of it. The solution was this was to provide Slurm's `cgroup.conf` file. In my use-case of getting Slurm to simply start, this did not need to be very sophisticated so I just used an off-the-shelf (off-the-stackoverflow[13]) configuration file.

Running `scontrol` again, I was finally able to see idling nodes, meaning that Slurm was successfully installed form source:

Listing 5.2: Slurm running

```
$ scontrol
scontrol: update NodeName=vincent-Laptop STATE=RESUME
```

Creating a scheduler plugin The Slurm documentation provides a short guide on how to add a plugin to Slurm.[14] As a start, I simply copied Slurm's default scheduler (which is also a plugin) to to the specified directory under a new name, and addings that new name to Slurms build files. It was then was time to recompile Slurm. Now however, during the recommended `autoreconf` step, an error occured:

Listing 5.3: 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.:[15] I then added a simple logging which then showed up in Slurm's log files too.

[13] https://stackoverflow.com/a/74211989
[14] https://Slurm.schedmd.com/add.html
[15] https://stackoverflow.com/questions/7805815/autoconf-error-on-ubuntu-11-04

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 I was using vscode, I 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 limitation on what can be done inside the plugin. For testing, I read out 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 I would submit 6 jobs, via Slurm's `squeue` command, only a part such as the last 3 could be logged inside the plugin. A possible explaination for this could be Slurms scheduler architecture: while there is a scheduler plugin, there also is a scheduler inside Slurm's main loop. That main scheduler loop also uses the same job queue as the plugin.

To hack around this, I 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 would apparently also need to re-implement alot of extra logic, which conventionally would be not be put inside the scheduler plugin.

I 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 such as when jobs were submitted. There also was support for "passive" scheduler that would get invoked when determined by Slurm. The version I tested however, 23.11, removed all such functionality and documentation. All plugins are implemented via threads that only have callbacks on 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 as a whole was dropped. While I would not say that a plugin approach is impossible, the effort to implement one from scratch subjectivly seems very high. The public documentation for developing on Slurm is scarce. There is a mailing list that can be searched, but it looks to be mostly aimed for administrating Slurm and not developing it.

Other avenues that could be explored are Slurm's Lua plugins. There is also a *Slurm simulator*[16] which could potentially be used for carbon-aware scheduling simulation, but that I did not look into much for reasons of little documentation and seeming lack of continued support.

5.1.2 Using a Simulation approach

In a recently released paper by Hanafy et. al [2], a prototype testbed for simulating job scheduling on cloud providers is implemented. These Jobs could be executed on spot

[16] `https://hpckp.org/articles/how-to-use-the-Slurm-simulator-as-a-development-and-testing-environment/`

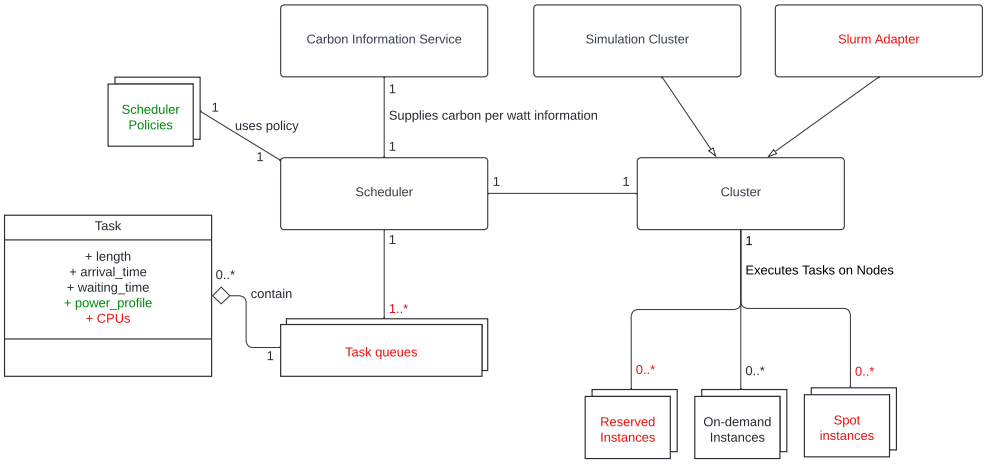

Figure 5.1: Class Diagram of GAIA, red indicates removal or reduction, green indicates additions

instances (cheap VMs that seek to increase cloud utilization), on-demand instances (short-notices VMs that are thus more expensive) or pre bought VMs (medium cost, but may be wasted), the paper then discussed balancing carbon- and dollar costs. They included a small notion notion of hardwre requirements in the form of required CPUs per job, but that was only used to scale the cost; all hardware requirements were abstracted away in the form of the cloud always having computing ressources available.

The important part is that they also included an implementation of some schedulers used in the highlighted related works. Among that an implementation for WaitAWhile[6]. This meant that I could add the improved job model to that exisiting testbed and have something to compare against aswell.

Description of the GAIA simulator I would first like to describe the existing testbed in detail to make it clear which part is my work and which is not. A class diagram is provided in 5.1 which is heavily inspired by the existing architecture diagram in the original paper.

I should probably give an overview of which parameters are supported, so I have that terminology later

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 to use a running average), and how to balance the dollar cost for scheduling. The latter I removed for simplicity, which is marked in the figure via the red color of the reserved and spot instances. In my case of carbon-aware scheduling, I did not need that feature.

In the original implementation, 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 would get scheduled on spot instances. Again, as I already removed the spot instance, feature, I did not need the queue multiplicity and it was removed.

The tasks queues are based on historical traces. GAIA provided multiple traces that are described in the paper. To those I added a trace of the HPI's Data Lab, which will be analysed in section. Link this

Tasks have a predetermined length (as saved in the trace), and they also hold 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. My contribution of the phases-and-power model was added to the tasks.

The Scheduler policies are also marked green, as I added scheduler policies that could use this new model. Specifically, I 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 my 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 would submit the tasks to an actual Slurm cluster. While this approach could have been used, I decided against it on the basis of not adding further complexity. Thus, I also removed the Slurm adapter from my implementation.

5.2 Implementing simulation-schedulers using the new model

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 time spans, 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

Especially the last point will be modified going further. As shown in Section **??**, jobs now have a predefined, variable power draw according to the defined phases. Resuming a job will also carry an additional startup phase with it.

5.2.1 Modifying the existing jobs to use the new model

The first step to be taken was to add the described model to the jobs as outlined in Figure 5.1. So far, jobs were generated from `csv`-formatted traces, one example being provided in Listing 5.4.

Listing 5.4: Excerpt from the Alibaba-PAI 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
```

In the original GAIA implementation, jobs have a `arrival_time`, which signify when the job should be 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 my work, as previously discussed.

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

Type `phases` 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, I would then use Python's `eval()` to read it back into a python object. As an example, a simplified `roberta.py`, the job that was measured in section 4.1, could look like the code in listing 5.5

Check how and if I should visualize this

Listing 5.5: 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.

In order to use this model definition and read out the power of a job at a given time, a function was crated 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 that to model parameters for the `phases` type. A similar job as to the one above would be created with the following:

Listing 5.6: Generic model 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
}
```

5.2.2 Uninterrupted oracle scheduling

There already was an existing version of this scheduling approach which assumed constant power and that the carbon trace would be fully known ("oracle"). Jobs would not be able to suspend & resume in this version. The algorithm for which is pseudocoded in Listing 5.7.

Listing 5.7: Pseudocode for the original 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 the new model where power is a function of time, only line 13 needed changes. Thus, I implemented a python function whose input is the model parameters (see Listing **??**), returning the power at a specified time. Evaluating this scheduling approach will take place in section 6, after another scheduling algorithm is introduced.

5.2.3 Suspend & Resume Scheduling for Heterogeneous Jobs

Under the constant-power, no startup-overhead assumption of the already existing implementation, the algorithm for this scheduler worked like the following:

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

This would minimize the 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 suspends & resumes also increases accordingly. In order to implement a scheduler for the improved job model, I 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 somehow decodes 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 all of this in the form of mathematical equations, which will be most part 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 would be written which is then solved by a *solver*. Different implementations for these solvers exist, but atleast 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 edge cases that were not intended, but that are possible under the model, 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 (as they need to be a linear equation). There are some mappings for such operations, but finding out how seems to be harder than in other paradigms, since the LP community appears to be smaller in comparison and online resources are limited. Some *tricks* will be highlighted in the latter parts as well.

Modelling carbon-aware scheduling in a linear program In this part, we will preset the iterative steps taken to model the scheduler as an LP problem, highlight some implementation details and visualizing the results of each step. I 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 as my 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. As an example, the code for this is shown in Listing 5.2.

Listing 5.8: LP Implementation for basic scheduling

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

Fix the stupid breaklines here

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 (in this case 300) integer-indexed boolean variables. These will be modeled as the time slots. Line 4 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 solving library. `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, 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 300 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 5.2, which on the bottom part shows the carbon emissions at each time slot and the upper graph shows the chosen time slots. 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.

Adding Overhead via Startup Phases In order to add a cost to starting a job, the code following in listing 5.9 is complemented, some parts are excluded for brevity.

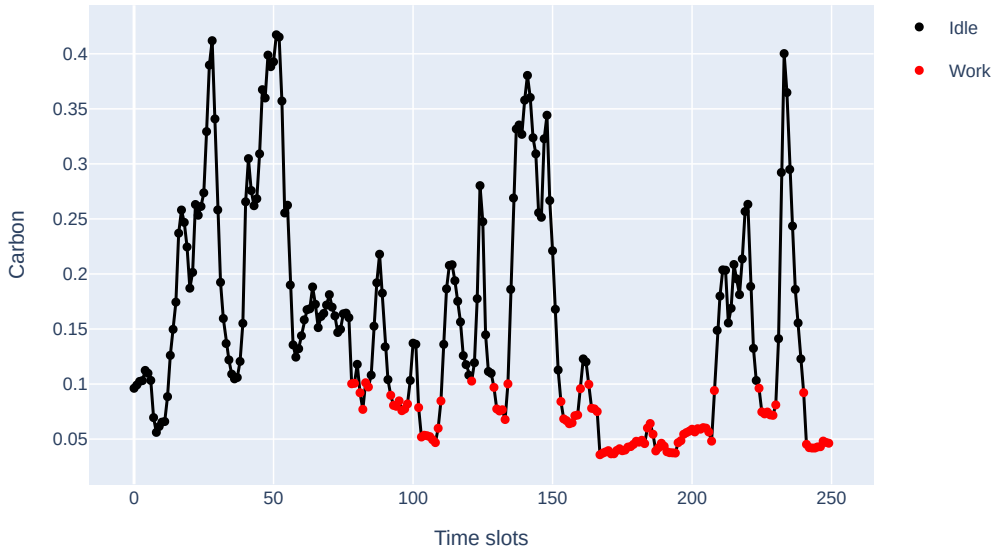

Figure 5.2: Executing the job for length-many time uses the lowest slots.

Listing 5.9: 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 5.1. 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 5.1: 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 solver scheduled jobs are set up at the very first time slots, as no startup could happen before time slot 0, thus minimizing carbon but obviously producing 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 slot, similarly to the previous Listing 5.9. With this step, the result already looks more like a reasonable schedule, as shown in Figure 5.3. This time, the optimal schedule includes executing the job in one go. Doing a visual check, the job is also executed on seemingly the lowest carbon intensities.

Adding a Notion of Progress So far, the assumption of constant power has been continued. Changing this assumption under the previous other scheduler using the greedy algorithm in section 5.2.2, was relatively easy, as changing the constant expression to a python function of the model sufficed. This cannot be reused in LP however, as the problem definition may only include linear equations, which a generic function is not. In order to add dynamic power into the linear program, we decided to *linearize* the function, meaning the model needed to be split up into multiple linear approximations.

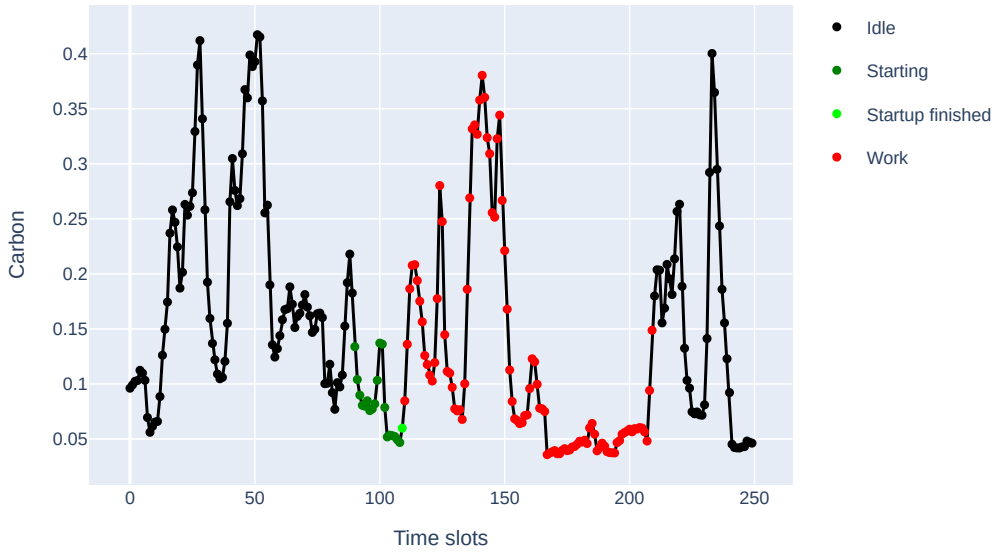

Figure 5.3: The scheduler now adds a startup phase if work is begun. In comparison to the previous figure, there is now a large block of work.

As the model already is essentially a step function of time to power, each phase being one step, the mapping is inherently pretty close. As such, *a lot* of equations were needed that each express "if the progress in the job is t, set power to $model(t)$". For that a notion of progress and time is needed inside the 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 5.10: Progress Variables in LP

```
1  # define "work_time_progressed" and "startup_time_progressed" as
2  # DEADLINE-many integer variables
3
4  M = DEADLINE * 2
5  for t in range(DEADLINE-1):
6      if (t>0):
7          prob += startup_time_progressed[t] >= startup_time_progressed[t-
               1] + 1 - (1 - starting[t]) * M
8          prob += startup_time_progressed[t] <= startup_time_progressed[t-
               1] + 1 + (1 - starting[t]) * M
9      prob += startup_time_progressed[t] <= starting[t] * M
10     prob += work_time_progressed[0] == 0
11     if (t > 0):
12         prob += work_time_progressed[t] == work_time_progressed[t-1] +
               work[t]
```

On a base level, 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).

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", in this case "large" being twice the amount of time slots, although it could 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 that `M` effectively does nothing (is "*relaxed*"), as every value of that variable is less than `M` by definition of `M`. In 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, line 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 5.4's upper graph. Attention should be drawn to the `work_progress` which keeps its value even when time slots are not used for work. The schedule found by the solver is not different to the previous one which added overhead, as the optimization goal was not changed and the progress indicators have no impact on the scheduling (yet).

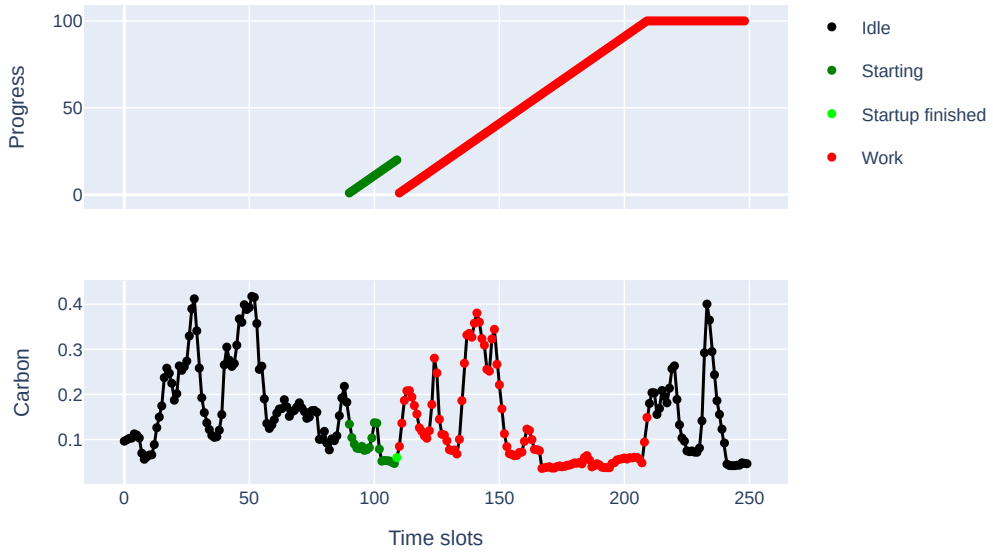

Figure 5.4: The scheduler now includes progress variables that keep track of the time spend in startup or work. The work progress is kept.

Adding Dynamic Power According to the New Model 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 schedule's carbon emissions can be changed to the following Formula (5.1):

$$carbonEmitted(t) = \sum_{p \in Phases} isActive(p,t) * powerOfPhase(p) * carbonEmission(t) \quad (5.1)$$

With that goal in mind, we add the pseudocode in Listing 5.11 to determine when a phase is active.

Listing 5.11: 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_variable[t] >= phase_indicator_lower[t] +
           phase_indicator_upper[t] - 1
15     prob += phase_variable[t] <= phase_indicator_lower[t]
16     prob += phase_variable[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_variabel` is then *active* where these two overlap (line 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 5.5. 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 high-powered phase (the green one) is scheduled on the lowest carbon emissions while the low-powered one is scheduled on the higher emission time slots.

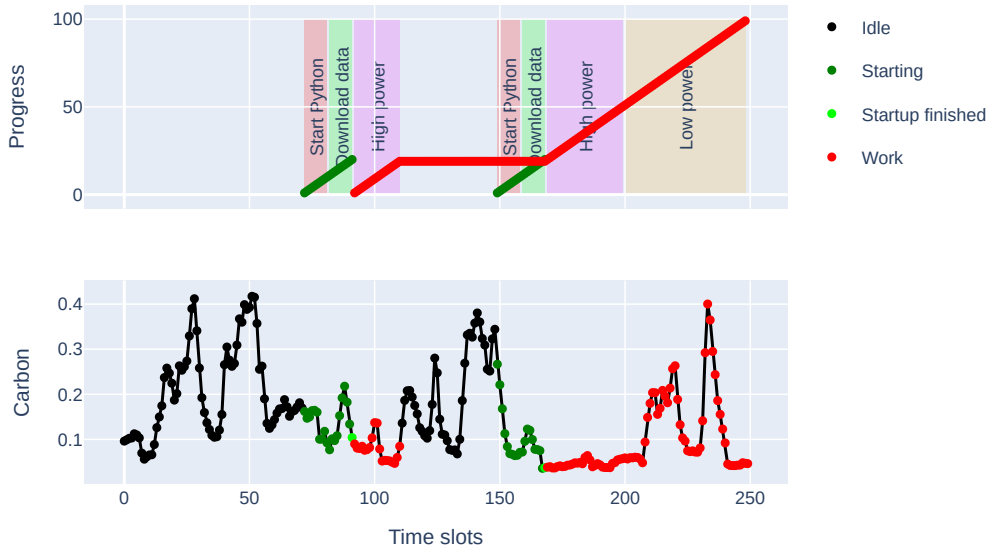

Figure 5.5: The final scheduling including differing power levels according to phases. In this example, there is a high-powered phase at 230 W and a low powered phase at 100 W. Notice how the first one is planned to be executed during the lowest carbon emissions.

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.

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 my laptop (a Lenovo T470 with 8 GB RAM and an i5-7200U CPU @ 2.50GHz), 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 would show that all 8 GB of RAM are in use and that swap space is being utilized a lot.

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 the original `GAIA` implementation, 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 would result 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 would then represent `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.

6 Evaluating Carbon-Aware Scheduling on the New Workload Model

In order to evaluate the carbon savings made possible by the previously established dynamic power model, 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 and used for the models. In a rather explorative process, we use the following parameters and run the simulation on 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 Stratey	suspend & resume, non-interrupted
Phases	high- and low-powered phases of different lengths periodically
Startup length	no startup, 5 minutes, 10 minutes, 30 minutes
Statup power level	100 W, 200 W
Waiting time	4 hours, 12 hours, 1 day, 2 days
Job length	1, 2, and 3 hours

Table 6.1: 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. This would represent the previous `GAIA` implementation, where jobs had a constant amount of power at all points in time.

This needs some better explanation, the phases don't really make too much sense, perhaps just go with the generic ML-job setup I introduced earlier

6.1 Running the Evaluation on a Cluster

As previously discussed in Section 5.2.3, calculating the suspend & resume scheduling tends to have high runtime and memory requirements. For that reason, the simulation was executed on the *SCORE Lab*, standing 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 the used 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 would distribute the jobs on the available nodes. While there is some leeway in the amount of sessions, experiments would crash beyond the 5th instance as the solver would not start.

We work under this restriction by distributing the experiments across two workers (via a script called `evalute_dynamic_power.sh`); as each experiment has an index, one worker executes the even numbered jobs and the other executes the odd ones. Using the 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. The command for launching one example worker, in this case the one for even jobs, is given in Listing 6.1.

Listing 6.1: Executing the Evaluation inside the SCORE Lab's Slurm environment

```
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     --container-mounts=/hpi/fs00/home/vincent.opitz:/home/vincent.opitz \
11     --container-workdir=/home/vincent.opitz/master-thesis/GAIA \
12     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. Line 3, `--container-writable`, means that the container will be reused if called again under the same name, this lets us conduct set up using (`pip install`).

6.2 Results

First, make a cool graph showing each job, comparing between scheduling it with (-out) checkpoint $ resume. HOPEFULLY, this will show that checkpointing & resuming decreases carbon emissions for (some) jobs. Then, compare this to the findings in the GAIA paper, perhaps confirming them.

Secondly, compare each job with phase-information against a job without that information. If things go well, there'd be some gain from having dynamic phases. If not, maybe give some speculation on the reasons.

7 Results

- here we would try to show off the difference between power-and-phase-oblivious scheduling and my new implementation which can make use of that
-

8 Discussion

Discuss some stuff!

9 Future Work

lol

Bibliography

[1] 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 page 8).

[2] 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 page 25).

[3] 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).

[4] 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 1).

[5] 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).

[6] 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 3, 8, 10, 13, 26).

[7] 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 8, 23).

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Arbeit zur Erlangung des Grades “Master of Science” der Digital-Engineering-Fakultät der Universität Potsdam mit dem Titel “A parameterized test bed for carbon aware job scheduling” (“Ein parametrisierbares Testbed für kohlenstoffbewusste Jobplanung”) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 6. Oktober 2024

_____

(Vincent Opitz)