University of Glasgow | School of Computing Science

Carbon-Aware Scheduling and Energy-Efficient Scaling of Batch Cluster Applications

Richard Arthurs (2465714a)

July 15, 2024

ABSTRACT

The escalating carbon footprint of computing, particularly in big data applications, necessitates a shift towards energy-efficient job scheduling and optimal resource utilisation within the Information Technology industry. This paper addresses this pressing issue by proposing a hybrid approach which integrates carbon-aware scheduling and dynamic hardware configuration adjustments. We propose a method which profiles previous executions of batch processed workloads to not only optimise their carbon emissions but also reduce their overall energy consumption. To achieve this, we incorporated these factors into the scheduler, to ensure that jobs are executed at times when they will produce the lower carbon emissions, while also dynamically adjusting hardware configurations to align with the workloads performance needs.
Furthermore, we experimentally evaluated the performance of our implementation with a variety of different runtime constrained workloads, to investigate the influence of runtime and deadline window lengths respectively. Our implementation resulted in significant reduction in i) carbon emissions of submitted jobs with a 24-hour deadline by a median value of 28.9%, ii) 5% greater savings than a Carbon-Aware scheduler without energy consumption profile consideration, and achieved iii) an energy reduction median value of 10% across all jobs, while satisfying job deadlines.

1. INTRODUCTION

Information and communications technologies (ICT) across the world currently produce between 2.1% and 3.9% of global emissions, a figure projected to rise without significant intervention [15]. The growth of hyper-scale cloud providers is fuelling this rapid increase in energy use [19], which require an immense amount of computing power to operate, this ability is provided by cloud data centres across the world. Data centres are constantly increasing global emissions [29], as the complexity and density of data advances with time. This presents an escalating issue which is detrimental to the sustainability of modern computing.
Despite the previous work on making energy-efficiency improvements for cloud data centres, these techniques alone are insufficient with respect to the ambitious sustainability targets of cloud data centres [19]. This is because even an energy-efficient data centre can still produce substantial carbon emissions due to their energy consumption [29, 19]. Thus, the Information Technology industry must prioritise both carbon-aware computing [19] and energy-efficiency of operations [9, 49], to truly move towards carbon optimisation.

Figure 1 depicts how the carbon intensity deviates over the course of a week with a diurnal pattern. This variance also illustrates how much the carbon footprint of an application can be affected by when it is processed, this presents an opportunity for optimisation. Carbon intelligent actions can be achieved by exploiting carbon intensity value fluctuations within the application's submission and completion deadline time [19, 44, 48]. Precisely temporal workload shifting, the aim of this technique in this context is to consume energy when the energy in question is produced by low-carbon sources.

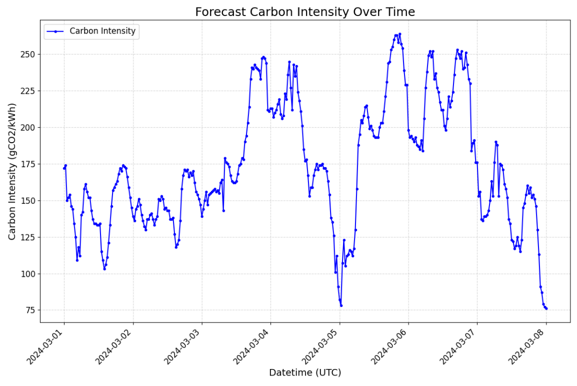


Figure 1: Graph of carbon intensity (gCO_2/kWh) values across a 7-day period sourced from the National Grid ESO, the graph illustrates the variability in carbon emissions of electricity.

There have been investigations into the effectiveness of temporal load-shifting as a means of completing operation using more clean energy [44, 19]. The practicality of such an approach with the aim of consuming cleaner energy from a public grid has been demonstrated by Google's Carbon Intelligent Computing System [32]. This research focuses on the approach of modelling an entire data centre's demands and shifting workloads across configurations with hundreds of nodes, this may be now public information but not particularly applicable to those that don't have resources on the scale of Google [32]. They do not analyse individual jobs, to profile them and have an optimised execution plan for each job on the cluster, this increased precision could lead to more carbon and energy savings.

Traditional data analytics frameworks have their own schedulers, for instance Apache Spark [4] has three default schedulers on YARN. These are First in First Out scheduler (FIFO), Capacity scheduler (Capacity), and Fair scheduler (FAIR). These each respectively have their own rationales, but they present several shortcomings with respect to different scheduling goals. There is no dynamic resource allocation for different types of Spark application (i.e. CPU intensive applications, I/O intensive workloads and hybrid applications), this leads to poor application performance [25] and excess energy usage [1, 49]. Energy consumption is not considered by any of the default scheduling strategies, resulting in these big data Spark applications suffering from excessive energy waste. Maximising resource utilisation via hardware dynamic scaling methods is key to reducing this energy waste [26].

Given these considerations, it becomes imperative to devise a system that considers both the carbon intensity of the energy and the optimal hardware configuration to efficiently utilise this energy. This is especially exploitable for batch workloads, which offer significant temporal, performance, and deadline flexibility [44].

This research aims to develop a hybrid approach which incorporates all the techniques mentioned in this section. Firstly, horizontal scale out should be determined in a manner which consumes the least energy while still satisfying the deadline. Secondly, profile the execution of the batch process such that the performance requirements of each workload for each stage is quantified to calculate an energy-efficient execution plan, and the energy consumption is modelled for consideration. Thirdly, carbon-aware scheduling must consider the application's energy usage profile and align this with the carbon forecast to find the optimal execution time when the job will produce the lowest carbon emissions. Finally, during execution, the progress of the application must be monitored, and the hardware adjusted according to the workloads performance needs at that each stage in the execution. Our aim is to demonstrate the potential of horizontal scale out, execution trace profiling on batch processed iterative jobs, energy consumption informed carbon scheduling, and finally dynamic node scaling during execution. With the goal of reducing the overall environmental impact of the job, by reducing both energy consumption and carbon emissions produced.

- The idea that using a hybrid approach in order to reduce the carbon footprint of an application alongside increasing overall energy efficiency. This approach is demonstrated in Section 3.
- An implementation of a Spark application profiler, which given a central processing unit (CPU) power usage log and a Spark event log, will produce an interpretable profile of the nature of the jobs' execution. This is done based on a novel approach using Spark application stages as identifiers for workload phases.
- An intelligent DVFS plan which is transmitted to each node, this plan will adjust the frequency governors of the processors in synchronisation with the progress of the application through its stages.
- An evaluation of the implementations' performance on a wide variety of applications which have been analysed from previous executions. The aim here is to demonstrate how much carbon and energy savings can be made for applications which have different deadline lengths and runtimes. Then have a discussion on the findings and what can be interpreted from them.

2. BACKGROUND

This section will outline the important properties involved in efficiently scheduling and scaling batch workloads on big data infrastructures. This presents many issues, many of which are complex to understand, generally these can be summarised into 6 main categories: resources management, data management, fairness, workload balancing, fault tolerance, and energy-efficiency [38]. This background will focus on elements which affect the overall environmental impact of a workload execution.

2.1 Carbon Awareness

The carbon intensity (gCO_2/kWh) of an energy source describes the amount of carbon that is emitted per unit (kWh) of electricity produced. This value fluctuates throughout the day based on what source the energy is coming from, whether that be a renewable source or non-renewable source, will dictate the scale of the carbon emissions emitted to produce that energy. It's imperative to not only rely on highly dependable forecasts, but also understand their calculation methods and anticipated accuracy. Carbon intensity, as a parameter, presents unique challenges in prediction when compared to more predictable metrics like electricity demand. The National Grid ESO (Electricity System Operator) [10], provides 48-hour carbon intensity predictions which are updated every hour. They utilise a state-of-the-art supervised Machine Learning regression model to produce these forecasts, which increases the accuracy of the forecasts over time. Figure 2 illustrates this forecast window and shows how closely the actual values for carbon intensity correlate with the predictions.

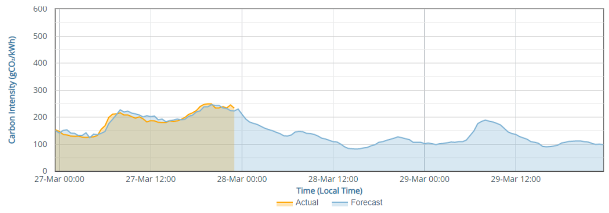


Figure 2: Carbon intensity forecast predictions from the National Grid ESO graphed alongside the actual carbon intensity, the legend shows the distinction colouring between actual and forecast values.

This carbon intensity forecast must be considered in order to determine the optimal time to execute the job. This is a trivial implementation of finding the time with the lowest intensity to process the job. However, when considering the execution profile in cohesion with this mechanic, the operation becomes more complex as there will be prolonged periods in a jobs' execution where it will use considerably more energy than other periods. This presents an opportunity to further optimise the carbon intensity analysis to align the jobs' execution profile high energy periods with the lowest carbon intensity time.

Carbon intensity values vary across the course of a day, and the granularity of the National Grid forecasts allows 30-minute blocks to work with, this gives leverage for optimisation of runtime, this will be bounded by the deadline given to the job and the runtime. Temporal workload shifting can be utilised here to align the job's execution with the carbon intensity range currently available, in a position which minimises the carbon intensity [19, 44, 12, 32].

2.2 Batch Processed Workloads

Batch processing refers to the execution of a series of tasks sequentially without the need for user interaction, besides the job submission. This approach is commonly utilised in various computing environments to handle large volumes of data or meet performance needs, or both. This technique is essential for handling repetitive tasks or data processing operations that do not require real-time interaction.

There are two types of workloads in data centres, those which are flexible and those which are inflexible, sometimes also known as delay-tolerant and delay-sensitive. Delay-tolerant refers to a workload which can be assigned a different execution as long as it is finished before the allocated deadline [43].

These workloads have two main characteristics which make them ideal for energy consumption optimisation, these have a sufficiently predictable execution pattern based on previous executions [36], and they are resource intensive. Due to this high resource intensity and massive volumes of data, these workloads contribute massive to carbon emissions, posing environmental sustainability concerns [32].

Batch processed workloads in real-world scenarios are generally delay-tolerant and have a degree of resource elasticity, this makes them suitable for offsetting workload execution times to periods which have lower carbon intensity, as evidenced in recent research [19, 44, 12, 32].

Throughout execution, they can exhibit a fluctuation of performance needs based on the current operation, this can lead to excess energy consumption due to static resource allocation and inefficient workload management strategies [26]. This runtime constrained nature, based on performance and volume of data, is what enables frameworks like Spark to scale out these operations, breaking them into partitions which can be proceeded rapidly.

In summary, understanding the characteristics and challenges of batch processed workloads is crucial for carbon optimisation of big data operations. This knowledge is vital to designing an effective carbon-aware, energy-efficient scheduling system.

2.3 Job Profiling

Analysing previous application execution profiles is a common method used for predicting runtimes [8], for scheduling big data applications. This strategy can also be used to gain insights about workload phases and how each factor of the system is currently affecting the performance, or progress of the execution. These factors include CPU utilisation, memory access, disk IO bandwidth, and network bandwidth [46].

As previously mentioned, the CPU has the highest energy consumption of these factors and has the most adaptability. We can see in Figure 3 that the CPU power usage varies significantly over an execution trace. Understanding when the application requires high CPU performance and when it does not, provides the basis for the energy savings actions that we aim to make in this research. Most systems in data centres have an inbuilt power plan which already controls their CPU frequency where it believes appropriate, but this is not optimal and often leads to underutilised resources [26, 31], these systems also have no way to predict load and prepare for it with the appropriate hardware configuration.

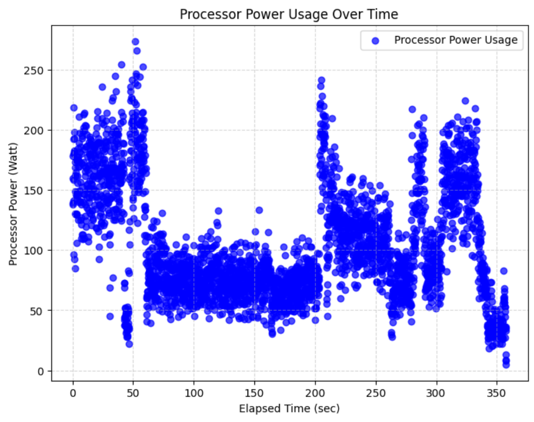


Figure 3: Scatter plot of processor power state (watts) against elapsed time (seconds) of an application. This example is a Grep algorithm unoptimised execution without any interference from the scheduler.

2.4 DVFS

Dynamic Voltage and Frequency scaling (DVFS) is a commonly used approach for optimising power usage of systems, this can be achieved by matching system power usage with the required performance. Specifically, the CPU is responsible for around 42% of the power usage, and in some scenarios can occupy 70% of the server's total energy consumption [25, 7]. When working with big data applications, they commonly present an execution profile which has a significant variance in CPU boundness throughout the execution of the job [36, 41, 28]. Supervised DVFS enables processors to minimise power usage by adjusting the frequency and supplied voltage to an appropriate amount in non CPU-bound periods, without stretching execution times [27]. In order to do this supervised DVFS, the execution trace of the application must be profiled from previous executions [36].

The energy efficiency of a big data processing platform is closely related to the utilisation of its hardware [37], this presents the case that an efficient resource allocation strategy for a system will increase the average resource utilisation, in turn improving the energy efficiency of such a system. To simplify this rationale in terms of the CPU, having a high frequency and low utilisation is a waste of energy as the process does not require this performance from the system, whereas a frequency state where the utilisation is high i.e. as close to 100% as possible is more energy efficient. This presents the basis for dynamic voltage and frequency scaling (DVFS), in the case of processing batch workloads, ensuring that the processor frequency state is apt for the current workload or precisely current stage of the application.

This method has been shown to be a highly effective way of reducing energy consumption of CPU-intensive applications [8] and those with more communication and I/O intensive workloads have also shown to save energy with this method [46], thus showing this technique has an adaptability to different workload profiles.

2.5 Horizontal Scale-out

Horizontal scaling, or sometimes referred to as scaling out, refers to increasing the number of computing nodes in a cluster. These additional nodes increase the overall computing power of the system, they require a framework for communication and data transmission like Hadoop File System (HDFS) [2]. Horizontal scaling increases a system's fault tolerance, and means extra performance can be gained without having to purchase more powerful hardware when extra nodes can be added. This increases the parallelism, meaning an application can be processed faster [34]. Although, there are many challenges present with scaling big data operations with the ever forward moving processor technology [23].

Having the correct number of nodes for a given workload is critical to achieving energy efficiency [47] [46], namely the configuration which uses the lowest amount of energy. Determining the optimal number of worker nodes will involve running the application on different numbers of workers, then extracting the power usage from each of these nodes. This information can then be used to demonstrate which configuration uses the lowest energy. The result of this will depend on the workload, more CPU-intensive workloads may benefit from a larger number of nodes due to the increased parallelism meaning a faster execution time [46]. Conversely, more I/O bound workloads would benefit less from more worker nodes and this would lead to unnecessary energy consumption and higher unutilised resources [14].

2.6 Big Data Applications

Big data processing frameworks, like Spark and Hadoop, are tools which enable processing of large-scale data sets rapidly on distributed systems or local machines. The term application refers to a user program which utilises a framework to perform various data processing operations. Applications can be written in a variety of languages, but are most commonly in Java, Scala or Python. Spark is a popular option due to the remarkable performance it boasts, outpacing MapReduce [3] by a factor of around one hundred. Spark's deployability is rooted from its open-source nature and extreme versatility to environments. Applications can be processed on local machines, cloud provided services and both homogeneous and heterogenous clusters.

This high performance can be attributed to its innovative utilisation of in-memory caching and optimisation of task processing via the use of Resilient Distributed Datasets (RDDs) abstraction. RDD abstraction enables hyper parallel processing across multiple worker nodes, allowing efficient computation on massive datasets with complex operations. When processing large datasets, Spark divides them into partitions and distributes them across the worker nodes in the cluster. This partitioning of data across multiple nodes is what allows Spark to increase its processing speed and inherently allows Spark to scale in order to handle larger volumes of data and or performance needs.

The execution of an application can be broken down into the operations being exercised on the data, these operations are organized into logical units called stages. A stage represents a sequence of tasks that can be executed in parallel, resulting in optimised, fast computation. Understanding how an application is broken down into stages is crucial for analysing its performance and reducing resource underutilisation [42]. Information about these stages being submitted and completed can be gathered from the Event logs produced by the Spark context, these provide insights into the execution flow and performance characteristics and duration of each stage. This understanding forms the basis for developing energy-efficient scheduling systems that optimise resource allocation and minimize energy consumption in Spark clusters.

3. APPROACH

This section will demonstrate how the system architecture has been implemented, then go into the details of how each component functions and the motivations for the strategies. Firstly, we will investigate the entire system architecture, this is so we have a general idea of how the hybrid system works before going into each component.

3.1 Overview

The aim is to design a system which can process job submissions which have unique identifiers that relate them to previous executions of the same workload, these previous executions have been analysed. This profiling is to produce an optimised execution plan for the workload which will determine the optimal node count and make dynamic scaling actions at runtime, with the rationale of reducing energy consumption by better matching system performance to workload demand. This profiling will also be used to aid the carbon-aware scheduler to delay the execution of the job to the optimal time slot, which minimises carbon emissions. Now that we've established a broad understanding of the system's functionality, we will formalise our assumptions, then dissect the specifics of how each component will be designed to face their respective challenges.

3.2 Assumptions

Acknowledging the complexities and challenges inherent in researching carbon awareness and energy efficiency in distributed systems, we will outline the specific assumptions

guiding our research in this section.
Cluster Environment

1. Dedicated homogenous cluster booked for experimentation, meaning jobs are run and scheduled in pure isolation.

2. Also, that workloads are homogenous across different nodes, that is, each node experiences similar computational demands.

Batch processed workloads

1. Batch processed jobs, which are runtime-constrained, will have been run with different scale-outs (1,2,4,6,8,10 worker nodes) to produce a robust profile for analysis.

2. Jobs will be scheduled with real-world applicable deadlines of 4-hours, 12-hours and 24 hours.

Energy Data

1. Carbon intensity forecasts are assumed to be sufficiently accurate.

2. Worker node energy consumption estimations will be made for the horizontal scale out decisions, these will be calculated using the Thoughtworks CloudCarbonFootprint simulator [40]. This is due to the complexity of accurately estimating with middleware the energy consumption of a node [35].

Scaling overheads

1. Scaling operations are assumed to occur without the use of excess energy, i.e. for starting new worker nodes and transferring data.

2. Making the DVFS actions to each node(s) CPU frequency, will be assumed to have negligible overhead.

3.3 Application Profiler

This section will comprehensively dissect each component of the Application Profiler, demonstrating how the model can produce an execution plan which alters resource allocation and initiating reactive DVFS actions to increase energy efficiency of our batch processed workloads. The objective of the application profiler is to process Spark event logs [4], Intel power and hardware usage logs [21], to produce an optimised execution plan which details the number of nodes and how the CPU frequency should be adjusted at each stage of the application.

3.3.1 Horizontal Node Scale-out

To determine the most effective number of worker nodes for achieving our energy efficiency goal, we will conduct the workload execution across a number of different worker nodes. Subsequently, we will analyse the results, selecting the number of worker nodes with the lowest energy consumption which still completes the job before the specified deadline. Once this optimal number of nodes is identified, we will proceed with profiling the execution of the application across the optimal number of nodes. Although we assumed homogenous workload across worker nodes, we will still gather the utilisation and energy usage statistics from all nodes and take the average to further enhance the validity of our implementation. The CPU utilisation and energy usage data will then be passed to the subsequent stage in the profiler to be investigated, with the goal of increasing CPU utilisation and decreasing energy consumption simultaneously.

3.3.2 Periodisation Strategy

DVFS can be used to reduce energy consumption, especially in instances where there are underutilised resources [26]. Figure 3 illustrates the CPU utilisation of a worker node with a balanced power plan (baseline) for the duration of a Spark Application. From this scatter plot, we can observe that there is a clear variance of values across the execution of the application, these values are grouped into time periods where the CPU utilisation remains around the same value. We can deduce from this that throughout the execution of the application, there is a substantial amount of underutilised resources, as the average CPU utilisation in this example is 42%. During these low utilisation periods, it is the case that the CPU frequency is set too high for the performance requirements of the workload at that time.

When stages of the application are processed from the event logs, these can be aligned with the CPU utilisation profile, the result is demonstrated by Figure 4. In this scatter plot, we can see that the CPU utilisation clumped periods aforementioned can be tied to their corresponding stage. This property, that CPU utilisation is sufficiently constant during an application stage, forms the basis for our periodisation of CPU boundness factor across the execution of a Spark application. Information gathered from these periods will be used to optimise hardware configuration via DVFS, with the rationale of maximising CPU utilisation by reducing frequency when utilisation is low and maintaining or boosting to a high frequency when utilisation is high.

The method also has the innate ability to adapt to different data sizes, as each stage used for this profiling will be run no matter what the data size. Meaning that the model can be used for the same application with different data input sizes. For instance, if one stage is completing one operation with more data this stage will just take longer and the CPU boundness of the action will remain the same, for the type of jobs specified in the assumptions. This differs from other DVFS techniques, which try to react to load in real-time [30] and those who rely on machine learning models for changes [49].

3.3.3 Period Workload Nature

These periods must then be parsed to give CPU_{stress} for that period, we can formalise this calculation into Equation 2 below:

$$Average\, CPU_{stress} = \sum_{t=i}^{n} \left(\frac{\text{current frequency}}{\text{max frequency}} \times \text{utilisation } \%\right) \tag{1}$$

The equation calculates the average CPU stress by summing the product of the fraction of the current frequency to the maximum frequency and the CPU utilisation percentage at each time step within the specified period. Where i represents the start time of each stage and n represents the length of the stage in seconds. The resulting values reside in the defined range $x \in (0, 100]$. It is important to note that stages are completed sequentially, one after another.

These values are then used as an indicator for the DVFS program of how CPU bound a stage is and how the frequency should be adjusted to cater for the performance needs of a stage. Once more, these adjustments are made to increase average CPU utilisation and decrease overall energy consumption.

3.4 DVFS Technique

To formalise the relationship between frequency and utilisation, an equation must be made to model the relationship. This equation represents the stress the CPU is under at a moment in time.

$$\text{CPU}\textit{stress} = \frac{\text{current frequency}}{\text{max frequency}} \times \text{utilisation } \% \tag{2}$$

This value can be used to gauge to what degree CPU bound each workload phase in the application is, then this value can be used to make the dynamic adjustments at runtime. CPU boundness indicates the extent to which the CPU serves as the primary bottleneck in the overall progression of the operation. From this, the CPU frequency can be adjusted to a value which effectively satisfies the performance needs of the current workload.

For a given processor, there are defined hardware limits for the lowest CPU frequency and highest CPU frequency. This range is our space for optimisation of CPU frequency, and to do this we will use the CPU*stress* value for the current stage of the application. These stress values are translated into frequencies, this is done by converting them to a percentage then multiplying them by the maximum frequency, this number is then rounded up to 5%, this is a small buffer to ensure runtime is not hindered substantially. The result will be the frequency the CPU will be set to for the execution of this stage. If the frequency is below the lower hardware limit of the system, then the frequency must be set to the minimum frequency.

Algorithm 1 Optimise CPU Frequency

```
1: Input: $stress\_value$, $min\_frequency$, $max\_frequency$
2: Output: $optimised\_frequency$
3: $percentage \leftarrow stress\_value \div 100$
4: $target\_frequency \leftarrow \text{round\_up}(percentage\_frequency)$
5: if $target\_frequency < min\_frequency$ then
6:     $optimised\_frequency \leftarrow min\_frequency$
7: else
8:     $optimised\_frequency \leftarrow target\_frequency$
9: end if
10: return $optimised\_frequency$
```

This algorithm iterates over the set of CPU stress values corresponding to the stages of the application, to produce the set of optimised CPU frequency values. These values are then pushed to the database with the job ID, ready to be read by the scheduler. These values will be used to adjust the CPU frequency in response to the application's progresses through each stage. Specifically, they are used to update the minimum and maximum frequency values corresponding to the current CPU governor to match the optimised CPU frequency.

This plan outlines instructions for adjusting CPU frequency throughout each stage of the application and the optimal number of nodes for execution. The goal of this profile is to process the application with the lower energy consumption than the previous execution, without increasing the runtime substantially. To achieve this, the profiler adjusts the CPU frequency value in a manner which increases average utilisation and reduces average frequency.

3.4.1 Energy Consumption Profile

An important process to understand, is calculating the energy consumed by each stage of the application. The reason this is an effective index to use is because for the duration of a stage, the processor will remain in the same power state, thus consuming around the same power for that period. The calculation is formulated here as total energy consumed, which corresponds to the total energy the CPU used during the execution. Where E_s denotes the total energy consumed by the processor during that stage and P_t represents the current CPU power draw at the time t and $[i..n]$ represents the duration of stages expressed as positive integers:

$$E_s = \sum_{t=i}^{n} P_t \tag{3}$$

This produces the total amount of energy used by the application during execution, by summing all the stages. This data must then be used to align the energy usage with the carbon intensity forecast in order to find the optimal execution time.

3.5 Carbon-aware Scheduler

The role of the scheduler is to receive job submissions from the user, the user must specify the job files, any arguments, job ID, and the deadline window in which they require the results by, i.e. the time at which the job must be completed. The scheduler must then read the relevant information from the database about the job using the unique identifier job ID, this information includes the DVFS profile and the energy usage profile. The scheduler must pass the DVFS profile to the Spark Master when the time comes for submission. The energy profile is used in cohesion with the carbon intensity forecasts supplied by the NationalGridESO [10], to make a carbon intelligent decision on when to execute the workload to achieve the lowest carbon emissions possible in the timeframe constrained by the deadline.

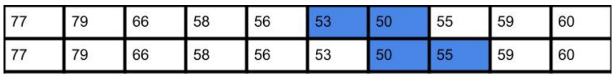
3.5.1 Optimal Execution Time

Finding our optimal execution time involves the carbon forecast, job submission time, deadline, job runtime and energy profile. This optimisation window must be iterated through alongside the energy usage profile. This involves calculating the potential carbon emissions would be produced at each possible execution time, searching for the minimum position. Then the algorithm will return the optimal time at which the job should be submitted. To increase the accuracy and precision of our execution placement, we must ensure that this optimal execution time is routinely calculated to ensure that updated carbon intensity forecasts are checked. This is because forecasts are updated throughout the day and the closer the current time gets to the forecast time, the more accurate the forecast predictions should be.

Algorithm 2 Find Optimal Job Submission Time

```
1: Input: Carbon forecast, job submission time, deadline,
   job runtime, energy profile
2: Output: Optimal job submission time
3: $optimal\_time \leftarrow$ null
4: $min\_carbon\_emission \leftarrow \infty$
5: for $t$ in Optimisation Window do
6:     $carbon\_emission \leftarrow$ calculateCarbonEmission($t$)
7:     if $carbon\_emission < min\_carbon\_emission$ then
8:         $min\_carbon\_emission \leftarrow carbon\_emission$
9:         $optimal\_time \leftarrow t$
10:    end if
11: end for
12: return $optimal\_time$
```

The output of this algorithm is a carbon intelligent execution time, which considers the carbon intensity forecast alongside the energy usage profile to ensure maximum savings in the execution window. The figure below illustrates a carbon intensity forecast across an execution window, here the benefit of analysing the energy profile alongside the forecast will be demonstrated. We will assume that 75% of the job's energy consumption occurs in the first 30 minutes and the remaining 25% occurs in the next 30 minutes. In the first row, we can see when an uninformed carbon aware scheduler would execute the job. In the second row, we can see when our scheduler would execute the job, which would lead to more overall carbon emission savings. This technique has previously been shown to increase carbon aware temporal workload shifting effectiveness[19].



77	79	66	58	56	53	50	55	59	60
77	79	66	58	56	53	50	55	59	60

Figure 4: Carbon intensity forecast for a job execution window. Row 1 represents when an uninformed carbon aware scheduler would allocate the execution of the job. Row 2 represents how an energy informed, carbon aware scheduler would allocate the execution of the job.

3.6 Limitations

Please note that only CPU power usage is considered when creating the energy usage profile for energy aware scheduling. However, when determining optimal node count, the profiler also takes into consideration additional energy demands such as those related to memory, disk, data transmission and networking. This comprehensive approach ensures that the selection of worker nodes for completing a job is optimised to minimise overall energy consumption. These calculations are a course-grained estimate using the Cloud Carbon Footprint methodology [40].

4. IMPLEMENTATION

This section explains the implementation of the system, which increases the energy efficiency and optimises the carbon emissions of batch processed workloads. The source code for our implementation can be found on Github [6].

4.1 Overview

As discussed in the approach, our system has three main components: the application profiler, the DVFS program and the carbon aware scheduler. We can observe the system architecture illustrated in Figure 5. The application profiler uses previous executions of jobs in order to produce an energy consumption profile for the scheduler, the optimal energy-efficient worker count, and a DVFS profile for the worker nodes to utilise. The DVFS is responsible for listening to the progress of the application and altering the CPU frequency in accordance to progress. The carbon aware scheduler is responsible for receiving job submissions from users, then scheduling such jobs in a manner which optimises carbon emissions with consideration of the energy consumption profile, constrained by the user provided deadline and runtime of the job.

4.2 Application Profiler

As assumptions state, it is assumed that the jobs have been run previously with different scale-outs i.e. worker node counts. The profiler automatically determines the number of nodes based on which previous execution had the lowest energy consumption. These previous executions reside in a bin directory, which contains their corresponding Spark Event and Intel Power logs. The Spark event logs are produced by the Apache Spark framework [4], and the Intel Power logs are produced by Running Average Power Limiting (RAPL) [11] interfaces. Also by assumption, as each worker node has homogeneous workload, they will all be assigned an identical DVFS plan.

The application profiler requires the execution stage timeline and the hardware statistics for analysis. Figure 6 shows a diagram of the overall pipeline of data, the program firstly parses the Spark event logs, these are filtered to extract the stage start times identified by the SparkListener **StageSubmitted** object, which contains information about the identifiers for the stage and RDD data. The stage end times are identified by the SparkListener **StageCompleted** object, which contains the same information as the stage begin identifier and the number of tasks completed during this stage. From this information, the profiler learns the runtime, stage count and stage durations for this execution trace.

4.2.1 DVFS Plan

The next process is correlating this information with the Intel Power Data log, for the purpose of creating the DVFS plan and constructing the energy consumption profile. This data is processed into a Pandas data frame containing the entries displayed in the Intel Power Data table in Figure 6. Then we must create a new column in the data frame to calculate the CPU_{stress} for a given time in the log. This calculation uses Equation 2 to generate the stress values based on the ratio of current frequency and maximum frequency multiplied by the utilisation, these values are simply taken from the current entries in that row and the maximum CPU

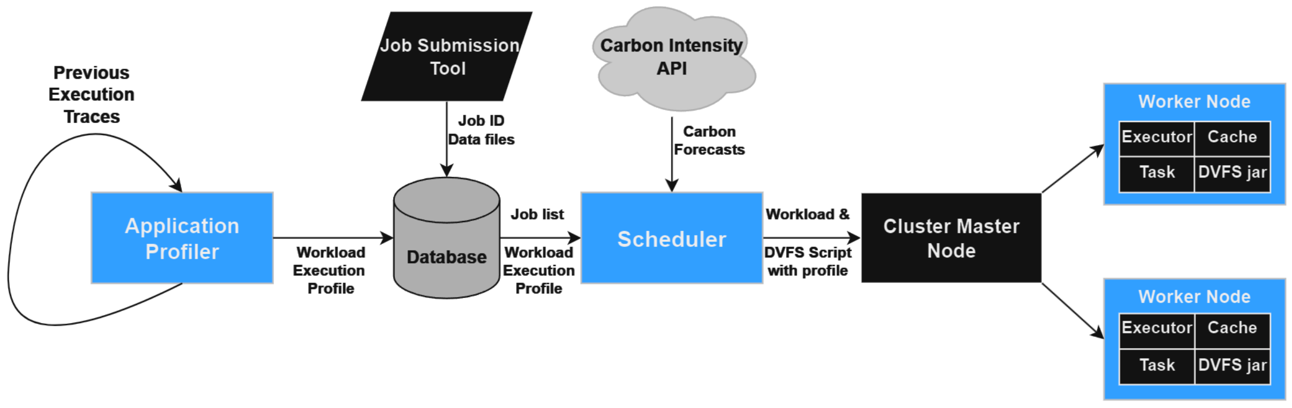


Figure 5: The system architecture illustrates the interaction between components to ensure a stable environment. It features a job queue where execution times are dynamically updated in real-time, responding to updated carbon intensity forecasts.

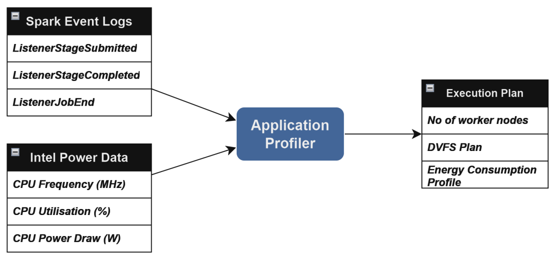


Figure 6: Application profiler data pipeline overview, showing Spark Event logs [4] and Intel Power Data [21] logs being processed into an energy-efficient execution plan.

frequency is known.

We then organise the stages into time windows, these will be our buckets for the log values which are associated with each stage. We then take our updated data frame containing the CPU_{stress} values and place them in their corresponding bucket based on their timestamp. The total in these buckets must then be divided by the duration of the stage to generate the average $Average\ CPU stress$, this calculation can be observed in Equation 1. Consequently, we obtain the $Average\ CPU stress$ for each stage, which forms the basis for the DVFS actions, this information is subsequently aggregated into an intelligent DVFS Plan which is sent to the database along with the scale-out factor (number of worker nodes).

4.2.2 *Energy Consumption Profile*

Finally, the energy consumption profile must be formulated, this involves once more the Spark Event logs and Intel Power Data logs. We must use Equation 3 and calculate the average for this value across each stage. This will produce an array of tuples which each hold a stage ID and the average CPU power draw of the CPU during that stage, respectively. This array can be used by the scheduler, as in the Algorithm 2 in Section 3.5.1, to find the optimal execution time with energy consumption and carbon intensity forecast both considered.

4.3 Job Submission Tool

As the System Architecture diagram in Figure 5 illustrates, the Job Submission Tool is responsible for receiving job submissions from the user and transmitting that input once validated to the database. This has been implemented as a command-line tool which prompts the user for the job **ID**, **runtime**, **filename**, **args** and **deadline** in a list separated by spaces. This input is then validated to ensure each value has been inputted in the correct format and if not the user is prompted by an error message and asked to retry the input. Once the input is validated, it will be parsed into the corresponding variables then passed into the database, followed by a success message with the submission time. After which, the tool will once more prompt the user, informing them that the system is ready for another job submission.

4.4 Carbon Aware Scheduler Algorithm

Figure 7 is a control flow diagram which shows how the carbon aware scheduler maintains a state where submitted jobs are detected then scheduled to be executed at their optimal time. The diagram shows how each of the components interact to maintain this stable state, these all have different responsibilities which will be further explained in this section. Each step involved in making this implementation functional will be explored, elucidating the strategic technical decisions made throughout this process.

4.4.1 *Entry Point*

The entry point is the '**main**' method of the scheduler program, this function is responsible for starting the scheduler and creating the initial environment which is determined by the job presence in the database. The control flow diagram in Figure 7 illustrates how this method maintains this stable state and ensures that jobs are executed at the optimal time to reduce carbon emissions. When jobs are detected their

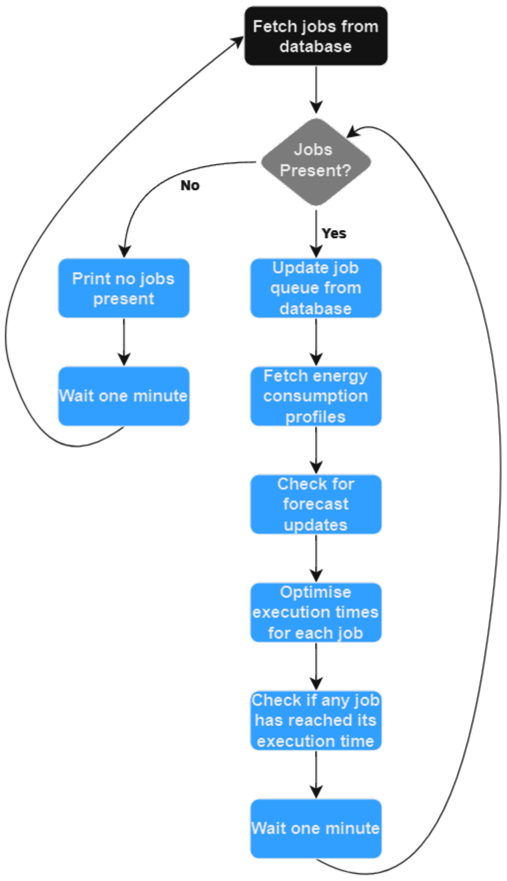

Fetch jobs from database
Jobs Present?
No
Yes
Print no jobs present
Wait one minute
Update job queue from database
Fetch energy consumption profiles
Check for forecast updates
Optimise execution times for each job
Check if any job has reached its execution time
Wait one minute

Figure 7: Control flow chart diagram of the scheduler behaviour based on job presence.

corresponding information from the scheduler, namely the energy consumption model, DVFS plan and node count are fetched and stored for consideration and submission respectively. This method also ensures that any carbon forecast updates are registered, and current execution times are updated if there is a more optimal position as a result of the forecast update.

These actions are performed periodically when there are jobs present in the database. To prioritise energy-efficiency, the loop is set to wait 60 seconds between each iteration. This deliberate waiting time ensures the schedulers presence is not substantial in terms of energy consumption. Importantly, the scheduler operates on the master node, which serves solely as an orchestrator and does not impact the performance of worker nodes responsible for processing applications.

This iterative approach serves the purpose of continuous improvement, aiming to further mitigate carbon emissions beyond the initial optimal execution time guidance. The goal is to execute the job as closely as possible to the absolute optimal execution time for carbon efficiency.

4.4.2 Processing of Carbon Intensity Forecasts

The '**updatePrediction**' method is responsible for fetching carbon intensity forecast data from the National Grid API. It retrieves the latest forecast data and extracts the values for the upcoming 24-hour period. This involves filtering out values outside this time window and validating that each entry has a forecast value.

The retrieved data is initially in a comma-separated file, this is processed into a Pandas data frame for easy manipulation and access in subsequent carbon emission calculations. Specifically, the '**forecast**' column is isolated from the data frame and converted into a list of integer values, each value corresponds to the carbon intensity for a 30-minute block.

This 30-minute block structure is coarse-grained and requires being converted into minutes blocks, this increase in granularity will allow more effective carbon emission reductions for jobs with larger than 30 minute runtimes, this is further explained in the implementation of the optimal execution time method. However, simply multiplying each block by 30 to cover the next 24 hours is inadequate. Consider the scenario where the current time is 9:10; in this case, using the 9:00-9:30 block and multiplying it by 30 would inaccurately represent the subsequent minute blocks due to preceding blocks being asynchronous with their actual values.

To rectify this, a formula is applied that calculates the remainder of the minutes divided by 30, this can be represented by the quotient symbol in python: $10\%30 = 10$. This remainder value is then utilised to discard carbon intensity predictions which correspond to past time intervals. Specifically, the formula removes the first (minutes past the hour % 30) elements from the list of carbon intensity values for the next 24 hours. This adjustment ensures accurate alignment with the current time, facilitating precise carbon emission calculations.

4.4.3 Finding Optimal Execution Time

The '**updateTargets**' method is responsible for assigning execution times to jobs processed by the scheduler. Each job in the queue undergoes a calculation to determine the time slot in its execution window that would result in the lowest carbon emissions, taking into account the job's runtime, deadline and energy consumption profile.

Initially, the method checks whether each job exists in a list called '**runTimes**' which contains optimised runtimes for each job. If not, the job is assigned its deadline minus its runtime and a buffer as its execution time. Additionally, a buffer is included (25% of the job's runtime) to account for potential worst-case scenario extra runtime due to Dynamic Voltage and Frequency Scaling (DVFS) actions.

Subsequently, each job's identifier variable is passed into the '**lowestCarbon**' function, which is responsible for finding the optimal execution time. This identifier gives this method access to the job's runtime, deadline and energy consumption profile for consideration when deciding the optimal execution time. Below is the detailed pseudocode in Algorithm 3 which shows how the '**lowestCarbon**' function determines the optimal execution time based on the execution window and the energy consumption profile. The job is carbon simulated from every possible execution time in minutes, at each step the carbon intensity value is multiplied by the average energy demand for the application stage at that time. The minimum is found from this operation, which represents the execution time which would currently produce the lowest carbon footprint for the job. This optimised time is appended to the '**runTimes**' and the previous corresponding runtime for that job removed.

Algorithm 3 Calculate Optimal Start Time

```
1:  for $i$ in $range(deadlineLength)$ do
2:      $total \leftarrow 0$
3:      $stagePosition \leftarrow 0$
4:      if $i > deadlineLength - jobSize - (20\%$ then
5:          break
6:      end if
7:      for $j$ in $range(jobSize)$ do
8:          if $(j > stageOrder[stagePosition])$ then
9:              $total \leftarrow current\ stage\ portion\ of\ minute$
10:             $total \leftarrow next\ stage\ porition\ of\ minute$
11:         else
12:             $total \leftarrow total + carbonForecast[i+j] \times energyProfile[stagePosition][1]$
13:         end if
14:         if $best > total$ then
15:             $best \leftarrow total$
16:             $mins \leftarrow i$
17:         end if
18:     end for
19: end for
```

As the stage times have a higher granularity than the minutes, their duration is stored in seconds then converted to decimal minute values for this routine. When the carbon simulation is iterating a minute block which contains more than one stage, then both energy demands must be considered proportionally for that minute. This is done by taking the decimal value from the stage duration applying this to the energy demand value for that stage then subtracting this value from one then applying this scalar to the energy demand value for the next stage. This ensures precision and accuracy within the carbon simulation, leading to greater carbon emission savings.

4.4.4 *Submitting Jobs*

Once the scheduler has maintained a stable environment and found the optimal execution for a job, the time comes for the program to submit the job to the cluster. The 'executeDeadlines' method is responsible for this part of the routine, the purpose of this method is to submit pending jobs to the cluster not only at the optimal assigned time but with the correct hardware configuration and while attaching the DVFS program. This method is part of the overall loop of maintaining this stable environment, when the method is called it checks the 'runTimes' list to detect whether any jobs have reached their runtime.

If this condition is true, the job is ready for submission to the cluster, this involves fetching the execution plan supplied by the profiler to the database. From this, the method can learn how many worker nodes to instruct the cluster to use and what DVFS plan should be attached to the DVFS program for the dynamic scaling actions. This method takes the **CustomSparkListener** DVFS Java file and compiles with the corresponding DVFS plan as an argument to produce a JAR file which can be attached to the job submission. This listener dynamically scales the CPU frequency in response to the progress between the stages of the application. Further details on this functionality will be provided in the next subsection.

4.5 DVFS Implementation

The program responsible for the DVFS is called **CustomSparkListener**, the purpose of this is to monitor the execution progress of Spark stages and dynamically adapt the CPU frequency of worker nodes accordingly. By adjusting the CPU frequency in response to workload variations, the aim is to optimise energy consumption without sacrificing performance.

Once the scheduler has compiled the DVFS program with the DVFS plan for the job, it is ready to be submitted as an application jar in the Spark submit command. Once the job is submitted, the jar file is copied into the working directory for each SparkContext on the worker nodes. Once here, the jar will begin to listen for progress in the applications' execution.

Initially, the jar will adjust the CPU frequency to the setting for the first stage of the application. This program is aware of the hardware limits in place for the CPU of the worker node, thus will apply the scaling actions in relation to this. While being aware that if the $AverageCPUstress$ value for a stage results in a frequency adjustment below the hardware limit, the jar will set the frequency to the minimum instead.

An example of a reaction to a stage change would go as follows, the listener detects the application has completed stage 1. The jar will then fetch the $AverageCPUstress$ value for stage 2, multiply this by the maximum frequency, check whether the result is within the hardware limits, then set the CPU frequency accordingly.

5. EXPERIMENTAL EVALUATION

This section evaluates the performance of our hybrid implementation, described in Section 4. We individually analyse the effectiveness of each component, then investigate the overall performance of the system. Examining the effects of different workload deadlines, runtimes and different types of Spark applications such as CPU intensive applications versus I/O intensive applications and hybrids.

5.1 Experimental Setup

5.1.1 *Infrastructure*

We used a local computing cluster consisting of 1 server for the master now and 10 servers for the worker nodes, each are equipped with 2 * Intel Xeon E5-2640 2GHz and 64Gb of RAM, this equates to 16 cores per node each with 4Gb of RAM, connected through a 10Gb Ethernet Interconnect. These distributed systems are running Scientific Linux version 6 and using Spark version 3.5.1 pre-built for Apache Hadoop 3.3 with Scala 2.12.14.

5.1.2 Workload

Below in Table 1 the algorithms and the associated input data types are listed. This list consists of a mix of CPU-intensive, I/O intensive and hybrid applications, this property is situational for some jobs and varies based on factors such as dataset size and hardware configurations. Workload characterization was key here, so these algorithms were set data inputs which constrain the runtimes to be a minimum of 5 minutes and a maximum of 90 minutes, the configuration used for constraining job runtimes can be found at the BenchSpark GitHub repository [18]. There were an equal amount of runtimes ran for each of these runtime ranges, 5–30 minutes, 31–60 minutes and 61–90 minutes.

Data Type	Algorithm	# of Runs
Vector	KMeans	13
Vector	LinearRegression	13
Vector	LogisticRegression	13
Tabular	GroupByCount	13
Tabular	Join	13
Tabular	SelectWhereOrderBy	13
Text	Grep	13
Text	Sort	13
Text	WordCount	17

Table 1: List of algorithms and the associated data type for the data they process, used for the experiments. There is also a column indicating how many times they were run in experiments.

To better simulate real-world scenarios, various deadline windows were employed for the jobs. Job submissions were assigned deadlines of 4-hours, 12-hours and 24-hours, the same amount of jobs were run for each deadline. For instance, a job with a 12-hour deadline emulates a developer submitting a job at the end of the working day, expecting the results by the following morning. Similarly, a 4-hour window represents a developer submitting a job in the morning and requiring the results by afternoon, and so on. The objective here is to demonstrate that by alternating these time frames, reductions in carbon footprint remain achievable, but how does this metric influence the percentage of carbon savings.

5.1.3 Job Submissions

Through the testing process, job submission times were purposely varied to closely mirror real-world scenarios. These times ranged from early mornings to late into the evening, 6am to 10pm was the range used. The testing begun data collection in mid-March, then continued for around a month to gather as much data as possible for analysis. Primed system signifies that the system has previous execution traces ready in the profiler's bin, as in assumptions. In total 121 jobs were submitted to the primed system, which were subsequently scheduled and submitted with optimised execution plans based on previous execution traces.

These previous execution traces were run after the optimal node count was determined, after that each workload was run with a balanced power plan devoid of any DVFS intervention, the output execution traces were the basis for the optimisations made to the 121 jobs submitted to the system. Each one of the jobs submitted has a corresponding previous unoptimised execution trace present ready for profiling, the energy consumption of these executions will be our comparison when evaluating the effectiveness of our dynamic node scaling actions during runtime.

5.1.4 Carbon Forecasts

Carbon forecasts were provided by the National Grid ESO [10], the scheduler considers the forecast values for the scheduling decisions. However, in our evaluation, we opt to utilise the actual values associated to the carbon trace of the jobs' execution time, to accurately demonstrate the tangible carbon savings achieved. The savings will be compared to the job being run at submission and at a random time which still satisfies the job deadline.

5.2 Evaluation of Horizontal Scale Out

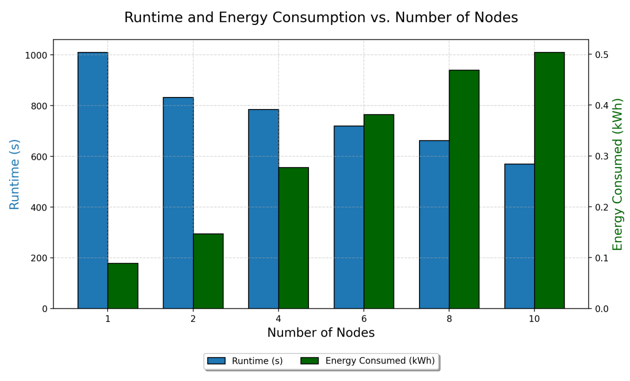


Figure 8: Bar chart showing relationship between the worker node count, versus the runtime and total energy consumption of a CPU-intensive workload. Runtime is expressed in seconds (s) and Energy Consumption in kilowatt-hours (kWh).

When comparing runtimes of applications across the different node counts of 1, 2, 4, 6, 8, and 10 it was observed that for CPU-intensive workloads the runtime would decrease as the node count increased, but for I/O intensive workloads the runtime would increase as the node count increased. This is likely due to the increased amount of data relocation and communication between nodes required to complete the tasks. Figure 8 demonstrates the relationship between node count on application runtime and energy consumption for a CPU-intensive workload. We can see that the runtime is steadily decreasing as more compute power is added, but the energy consumption is also growing at a substantially faster rate than the runtime is decreasing. For this workload, one worker node would be selected, as this would complete the job by the deadline and use less energy than the other node counts.

5.3 Evaluation of Carbon-Aware Scheduler

To evaluate the performance of the implementation statically, we can analyse the carbon emission saving percentages across the 121 jobs to provide insights on what savings can be expected. Firstly, the average carbon footprint reduction percentage stands at 17%, this result aligns with the expectations from temporal workload shifting techniques [44]. Secondly, the median value was 28%, this metric offers a more robust representation of the expected reduction of carbon emissions for a workload which lies within our specification as mentioned in Section 5.1.2. These reductions are around 5% greater than an approach which we implemented previously [5], which only considered the runtime of the application, not the energy consumption profile.

Relying on carbon intensity forecasts for savings has been shown to successfully reduce the average carbon footprint of a submitted job by 19%, the scheduler did not increase the carbon emissions of any workload due to the scheduling placement. These savings are broken down into categories for deadline windows and runtimes in Sections 5.3.1 and 5.3.2.

5.3.1 Effect of Deadline Window Length

Previous work has demonstrated that delay-tolerant workloads can yield substantial savings, and the trend has been presented that these savings will increase with longer deadlines [19]. This is because with longer deadlines, more carbon forecast blocks become available for scheduling. This larger period for optimisation empowers the scheduler to further exploit the energy consumption profile to make more savings, jobs with a 24-hour window had an average saving of 25%.

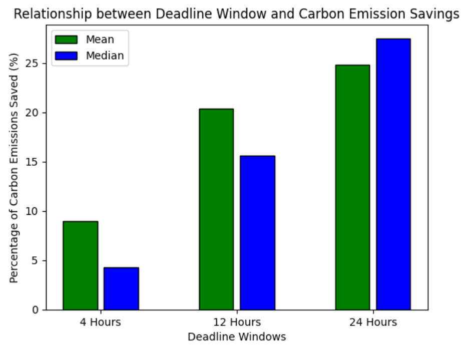


Figure 9: This bar chart illustrates the mean and median carbon emission percentage savings for the different deadline windows. This was produced from the compiled results from all scheduled jobs.

Figure 10 demonstrates how the mean and median savings values increase as the deadline window enlarges. From this we can observe that when a larger window for optimisation is available to the scheduler, larger savings are possible and generally are made. This statement is further consolidated in Figure 9 which shows that the average and median savings increase drastically when a larger deadline window is available for scheduling.

5.3.2 Effect of Job Runtime

The runtime of a job is detrimental to carbon savings percentages, as if there is a low-carbon block available, there are generally clustered with other low-carbon blocks due to the NationalGrid carbon intensity forecasts generally having a diurnal pattern. From this, we can say that larger jobs can access more low-carbon blocks when present, this presenting a larger carbon emission reductions than smaller jobs. Jobs with a runtime equal to or less than 30 had a median saving of 10.4%, while jobs with a runtime greater than 30 had a median saving of 16%.

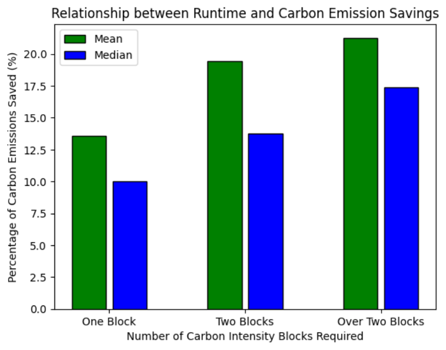


Figure 10: This bar chart illustrates the mean and median carbon emission percentage savings for the different amount of carbon blocks available for scheduling, this is determined by the runtime. This was produced from the compiled results from all scheduled jobs.

Figure 10 demonstrates how the mean and median savings value increases as the number of carbon blocks used in the execution increases. During the analysis of the results, a noticeable trend emerged: as the runtime and deadline window length (i.e. 4, 12, 24 hours) were closer to one another, the savings were greater than where the difference was larger. Suggesting that temporal workload shifting for large jobs can make significant carbon savings, even if the deadline length is only relatively bigger than the runtime.

5.4 Evaluation of Node DVFS scaling

The aim of the DVFS actions was to increase the average CPU utilisation throughout the execution of the batch processed workloads. This approach aimed to align the system's energy consumption with the required performance level. Across the 121 jobs, the system increased the median utilisation figure by 11% compared to the unoptimised executions, which ran with a balanced power plan. This resulted in a median reduction of 10% in the energy required for an execution, these savings are substantial when you are considering the amount of energy consumed by batch processed workloads every day [29]. There was a median runtime increase of 13% when compared to the unoptimised executions, but all jobs submitted were completed before their deadline. There was also a trend that more I/O intensive workloads could achieve higher energy consumption reductions, with some reaching 15-20% energy saved. This is a result of these applications having long periods where there is a low compute performance requirement, but the CPU frequency is not reduced in response to this and left at base frequency, wasting energy.

Figure 11 illustrates an example of how an unoptimised and optimised execution profile compare. We can see that the power usage in the optimised profile is much more sporadic and unconstrained in the unoptimised execution. Conversely, in the optimised solution the power usage exhibits much more stability and a tighter control, which results in lower energy consumption, this is a result of appropriate CPU frequencies being set for the workload.

Figure 11 also demonstrates how the average resource utilisation has been increased across the execution of the application. Particularly for periods where the unoptimised executions exhibit long periods of low utilisation, our approach excels at reducing the energy waste in these periods.

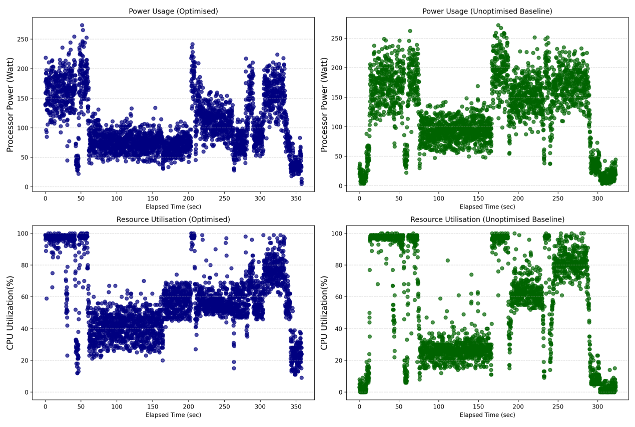


Figure 11: This plot illustrates the characteristics of an unoptimised balanced power plan execution profile versus an DVFS enabled execution, this was a Grep algorithm workload. In respect to CPU power usage and CPU utilisation, measured in watts (W) and percent (%) respectively.

5.5 Periodisation Strategy

Our strategy was breaking down each workload into periods which exhibit the same performance needs, such that we can use these periods to make interpretations for scaling actions. This method was using Spark stages as an indicator for creating these periods, a strategy which has not been heavily investigated previously for energy savings. Below in Figure 12 we can observe how the CPU utilisation correlates closely with the Spark stages which are distinguished by different colours. Overall, this method has been a great indicator of the CPU boundness or compute requirement of each part of the application. Similar methods of breaking down applications in stages or tasks for indication of performance needs, have also shown success on other frameworks, such as in MapReduce [20], implying this technique could be applicable across big data processing frameworks besides Spark.

5.6 Discussion

The evaluation of our systems performance presented its

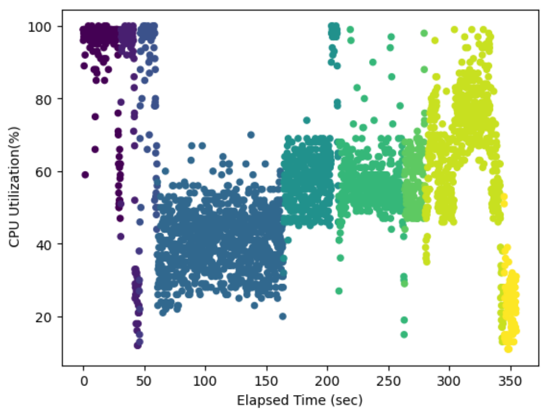


Figure 12: This plot demonstrates the capability of Spark stages as a means of distinguishing workload compute requirements. This was a Grep algorithm unoptimised execution profile.

ability to reduce carbon emissions by delaying the execution of workloads and ability to complete operations with reduced energy consumption requirements, therefore we have demonstrated that our system is capable of substantially reducing the environmental impact of a batch processed workload with a specified deadline on an isolated cluster. These results show that not only carbon savings can be made by profiling a workload [19], but by understanding the nature of the workload and applying periodisation, energy consumption reductions can be made, while still satisfying the job deadline. There were some unexpected findings found during our research, one of the most interesting was using Spark stages as an indicator of workload compute requirement for workloads, this was found to be an effective indicator which had not been previously investigated for the purpose of energy-efficiency improvements. We also found that the ratio between runtime and deadline length correlates with an increase in potential carbon emissions savings as this ratio increases. Another trend in the results was that applications with longer periods which are I/O intensive, achieved much greater energy savings, this is due to an inappropriate CPU frequency employed at low utilisations by an uninformed system power plan during the unoptimised executions.

6. RELATED WORK

This section examines prior research in batch workload profiling, carbon-aware workload scheduling, carbon-aware scheduling, and energy-efficient scaling actions.

6.1 Trace Analysis Profiling

In recent years, there have been numerous proposed solutions for modelling cluster resource utilisation traces, as means to increase performance [36, 25, 33] and those to increase the energy-efficiency [43, 37, 46]. Many approaches use the insights gained from workload profiling to increase the resource utilisation to improve energy-efficiency [37, 46, 25, 27], this rationale is solidly grounded in research. Our method is a precise and meticulous approach where each individual batch processed workload is profiled and execution is optimised on isolated worker nodes.

6.2 Carbon-Aware Scheduling

There are a number of studies which tackle carbon-aware computing in the context of big data and distributed systems [19, 32, 44, 48, 12]. Carbon intensity predictions from a public grid have been shown to be a reliable indicator to offset workloads with flexible deadlines to reduce the overall carbon footprint of an application [19, 44, 32]. The Carbon-Scalar [19] paper, demonstrates that increased completion time deadlines mean more low carbon slots are available, which allows larger reductions to be achieved, this was also the case in our research.

The Let's wait-a-while [44] approach employs temporal shifting to reduce the carbon emissions produced by batch workloads, they achieve this by utilised threshold and deadline constrained methods, they leverage overnight or weekend times to further maximise savings. This is similar to our method of time shifting, but they fail to consider the energy consumption profile of the workloads when making scheduling decisions, which could mean larger savings as demonstrated in Section 5.3.

6.3 DVFS

Dynamic voltage and frequency scaling (DVFS) is one of the most effective techniques for optimising energy consumption of computer systems, specifically in data processing [39, 26, 8, 1, 27, 49]. A common approach is creating a model which reacts to load across a system and adjusts the hardware in real-time [1], these achieve energy savings compared to traditional methods where the hardware follows a balanced power plan. There are a lot of studies on using DVFS as a means to reduce the energy consumption in cloud provided clusters [8, 27, 39], although DVFS enabled cloud environments are uncommon. Improving the energy-efficiency of processing batch workloads using DVFS involves many factors, [49] utilises learning models to predict the workload and thermal dissipation. From their experiment with many different DVFS techniques, the most successful method attempts to use the live CPU utilisation to adjust the CPU frequency to an appropriate value. Conversely, our approach uses periodisation to generate an adaptable DVFS plan, which monitors the progress of the execution.

6.4 Horizontal Node Scaling

Horizontal node scaling, in the context of energy-efficiency and performance, has had a substantial amount of previous work which supports the belief, that this can be utilised to reliably increase the performance of big data operations [34, 16, 20]. What is often overlooked is the implications of using this technique in terms of additional energy consumption. Also, there is significant complexity involved in maintaining an optimal configuration, for instance how should the system react once CPU utilisation decreases (i.e. whether to scale down or not) [19]. The process of scaling up or down entail significant energy overhead in terms of data transmission and required inter-node communications, the impact of these factors are frequently not considered when scheduling [34]. To avoid this complexity issue and unaccounted additional energy waste, our approach pre-determined the optimal node count before execution and does not alter this during runtime.

7. CONCLUSION

The properties of batch processed workloads that have been demonstrated in this paper, resource elasticity and delay-tolerant nature allow optimisations to be made which reduce the overall environmental impact of the computations. Our hybrid system leverages these properties, to efficiently schedule and scale applications, based on profiled execution traces, with the aim of fostering eco-friendly execution practices. We implemented our system on a homogeneous cluster with 10 worker nodes, 121 jobs were optimally scheduled and executed on this configuration to measure the performance of the system. We demonstrated that using batch processed workloads with real-world scenario deadlines, our implementation reduced i) carbon emissions of submitted jobs with a 24-hour deadline by a median value of 28.9%, ii) 5% less emissions than a Carbon-Aware scheduler without energy consumption profile consideration, iii) energy reduction median value of 10%. The evaluation shown that the workloads with the highest potential for carbon savings were those with the 24-hour deadline duration and 90 minute (longest) runtime. Also, that the jobs which made the largest energy savings were those with long periods of I/O intensive workload.

For future work, the profiler currently only considers CPU usage metrics, a more comprehensive approach could also incorporate a memory-aware element [45] to the execution profiling, this would give more insights into optimisation that could be made to the execution of the job to make it more energy-efficient. Another improvement would be long term testing and more intensive testing, as in more jobs ran with longer runtimes. This was not possible with the timeframe of the project and the limited hardware available. Longer term testing could yield more insightful results regarding how the time of the year influences the scheduler's performance, this could also enable stronger visualisations of the aforementioned trends which emerged during this research. Due to the increasing complexity of architecture of modern CPUs, for instance, Intel's performance cores and efficiency cores in their Raptor Lake processors [22], gauging how stressed a CPU is becoming a more complicated calculation. Studies and documentation are presenting Instructions per Cycle or IPC as a better indicator for performance needs [17, 24, 13] than CPU utilisation, as this metric is more bound to software and workload [24, 13], thus research would be intriguing on how this metric can be used to profile big data processing with our method.

Acknowledgments. I would like to thank Dr. Lauritz Thamsen for his invaluable guidance, enthusiasm and profound insights, throughout each stage of this research endeavour. I would also like to show my gratitude to Professor Phil Trindler for the access we were granted to testing on the school's GPG Cluster, this was immensely beneficial to our research.

8. REFERENCES

[1] H. Ahmadvand, F. Foroutan, and M. Fathy. DV-DVFS: merging data variety and DVFS technique to manage the energy consumption of big data processing. *Journal of Big Data*, 8(1):45, Mar. 2021.

[2] Apache. Hadoop distributed file system. Available at `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html`. Accessed: April, 2024.

[3] Apache. Mapreduce. Available at `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`. Accessed: April, 2024.

[4] Apache. Spark monitoring & instrumentation documentation. Available at `https://spark.apache.org/docs/latest/monitoring.html`. Accessed: April, 2024.

[5] R. Arthurs. Carbon-aware big data processing in cloud infrastructures - level 4 dissertation. Available at `https://github.com/RickyArthurs/Carbon-Aware-Big-Data-Job-Scheduler/blob/main/dissertation%20L4.pdf`. Accessed: April, 2024.

[6] R. Arthurs. Research source code github repository. Available at `https://github.com/RickyArthurs/L5-Source-Code`. Accessed: April, 2024.

[7] L. A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Springer International Publishing, Cham, 2019.

[8] R. N. Calheiros and R. Buyya. Energy-Efficient Scheduling of Urgent Bag-of-Tasks Applications in Clouds through DVFS. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 342–349, Singapore, Singapore, Dec. 2014. IEEE.

[9] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 43–56, Bern Switzerland, Apr. 2012. ACM.

[10] Datopian. National carbon intensity forecast. Available at `https://data.nationalgrideso.com`. Accessed: April, 2024.

[11] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 189–194, Austin Texas USA, Aug. 2010. ACM.

[12] J. Dodge, T. Prewitt, R. T. D. Combes, E. Odmark, R. Schwartz, E. Strubell, A. S. Luccioni, N. A. Smith, N. DeCario, and W. Buchanan. Measuring the Carbon Intensity of AI in Cloud Instances. 2022. Publisher: [object Object] Version Number: 1.

[13] S. Eyerman and L. Eeckhout. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE Computer Architecture Letters*, 13(2):93–96, July 2014.

[14] F. Fernandes, D. Beserra, E. D. Moreno, B. Schulze, and R. C. G. Pinto. A virtual machine scheduler based on CPU and I/O-bound features for energy-aware in high performance computing clouds. *Computers & Electrical Engineering*, 56:854–870, Nov. 2016.

[15] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, 2(9):100340, Sept. 2021.

[16] L. Globa and N. Gvozdetska. Energy efficient workload processing in distributed computing environment modeling. In *2019 International Conference on Information and Telecommunication Technologies and Radio Electronics (UkrMiCo)*, pages 1–6, Odessa, Ukraine, Sept. 2019. IEEE.

[17] B. Gregg. *Bpf performance tools: Linux system and application observability*. Pearson Education, Inc, Hoboken, 1 edition, 2019.

[18] D. Group. benchspark github repository - jobs configuration file - an extensible toolset for spark performance benchmarking. Available at `https://github.com/dos-group/benchspark/blob/main/run_scripts/jobs_config.py`. Accessed: April, 2024.

[19] W. A. Hanafy, Q. Liang, N. Bashir, D. Irwin, and P. Shenoy. CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency, Oct. 2023.

[20] T. T. Htay and S. Phyu. Improving the performance of Hadoop MapReduce Applications via Optimization of concurrent containers per Node. In *2020 IEEE Conference on Computer Applications(ICCA)*, pages 1–5, Yangon, Myanmar, Feb. 2020. IEEE.

[21] Intel. Performance counter monitor (intel® pcm). Available at `https://github.com/intel/pcm`. Accessed: April, 2024.

[22] Intel. Raptor lake processor generation. Available at `https://ark.intel.com/content/www/us/en/ark/products/codename/215599/products-formerly-raptor-lake.html`. Accessed: April, 2024.

[23] A. Katal, M. Wazid, and R. H. Goudar. Big data: Issues, challenges, tools and Good practices. In *2013 Sixth International Conference on Contemporary Computing (IC3)*, pages 404–409, Noida, India, Aug. 2013. IEEE.

[24] M. Lan, R. Yan, L. Huang, Y. Cheng, L. Yang, and J. Zhang. IPC-loss: A fine-grained processor performance analyze method. In *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*, pages 11–18, Nanjing, China, Sept. 2022. IEEE.

[25] H. Li, Y. Wei, Y. Xiong, E. Ma, and W. Tian. A frequency-aware and energy-saving strategy based on DVFS for Spark. *The Journal of Supercomputing*, 77(10):11575–11596, Oct. 2021.

[26] H. Liu, B. Liu, L. T. Yang, M. Lin, Y. Deng, K. Bilal, and S. U. Khan. Thermal-Aware and DVFS-Enabled Big Data Task Scheduling for Data Centers. *IEEE Transactions on Big Data*, 4(2):177–190, June 2018.

[27] J. Mao, X. Peng, T. Cao, T. Bhattacharya, and X. Qin. A frequency-aware management strategy for virtual machines in DVFS-enabled clouds. *Sustainable*

Computing: Informatics and Systems, 33:100643, Jan. 2022.

[28] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Perez-Hernandez. Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–442, Taipei, Taiwan, Sept. 2016. IEEE.

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

[30] L. Mo, Q. Zhou, A. Kritikakou, and J. Liu. Energy Efficient, Real-time and Reliable Task Deployment on NoC-based Multicores with DVFS. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1347–1352, Antwerp, Belgium, Mar. 2022. IEEE.

[31] Qinghui Tang, S. Gupta, and G. Varsamopoulos. Energy-Efficient Thermal-Aware Task Scheduling for Homogeneous High-Performance Computing Data Centers: A Cyber-Physical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1458–1472, Nov. 2008.

[32] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne. Carbon-Aware Computing for Datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, Mar. 2023.

[33] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–13, San Jose California, Oct. 2012. ACM.

[34] C. Roy, M. Pandey, and S. SwarupRautaray. A Proposal for Optimization of Data Node by Horizontal Scaling of Name Node Using Big Data Tools. In *2018 3rd International Conference for Convergence in Technology (I2CT)*, pages 1–6, Pune, Apr. 2018. IEEE.

[35] A. M. Sampaio and J. G. Barbosa. Energy-Efficient and SLA-Based Resource Management in Cloud Data Centers. In *Advances in Computers*, volume 100, pages 103–159. Elsevier, 2016.

[36] D. Scheinert, A. Alamgiralem, J. Bader, J. Will, T. Wittkopp, and L. Thamsen. On the Potential of Execution Traces for Batch Processing Workload Optimization in Public Clouds. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 3113–3118, Orlando, FL, USA, Dec. 2021. IEEE.

[37] J. Song, Z. Ma, R. Thomas, and G. Yu. Energy efficiency optimization in big data processing platform by improving resources utilization. *Sustainable Computing: Informatics and Systems*, 21:80–89, Mar. 2019.

[38] M. Soualhia, F. Khomh, and S. Tahar. Task Scheduling in Big Data Platforms: A Systematic Literature Review. *Journal of Systems and Software*, 134:170–189, Dec. 2017.

[39] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li. An Energy-Efficient Task Scheduling Algorithm in DVFS-enabled Cloud Environment. *Journal of Grid Computing*, 14(1):55–74, Mar. 2016.

[40] Thoughtworks. The cloud carbon footprint. Available at `https://www.cloudcarbonfootprint.org/`. Accessed: April, 2024.

[41] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244, Karlsruhe Germany, June 2011. ACM.

[42] K. Wang, M. Maifi Hasan Khan, N. Nguyen, and S. Gokhale. A Model Driven Approach Towards Improving the Performance of Apache Spark Applications. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 233–242, Madison, WI, USA, Mar. 2019. IEEE.

[43] P. Wang, L. Xie, Y. Lu, and Z. Ding. Day-ahead emission-aware resource planning for data center considering energy storage and batch workloads. In *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pages 1–6, Beijing, Nov. 2017. IEEE.

[44] P. Wiesner, I. Behnke, D. Scheinert, K. Gontarska, and L. Thamsen. Let's wait awhile: how temporal workload shifting can reduce carbon emissions in the cloud. In *Proceedings of the 22nd International Middleware Conference*, pages 260–272, Québec city Canada, Dec. 2021. ACM.

[45] J. Will, L. Thamsen, J. Bader, D. Scheinert, and O. Kao. Ruya: Memory-Aware Iterative Optimization of Cluster Configurations for Big Data Processing. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 161–169, Osaka, Japan, Dec. 2022. IEEE.

[46] T. Wirtz and R. Ge. Improving MapReduce energy efficiency for computation intensive workloads. In *2011 International Green Computing Conference and Workshops*, pages 1–8, Orlando, FL, USA, July 2011. IEEE.

[47] Y. Ying, R. Birke, C. Wang, L. Y. Chen, and N. Gautam. Optimizing Energy, Locality and Priority in a MapReduce Cluster. In *2015 IEEE International Conference on Autonomic Computing*, pages 21–30, Grenoble, France, July 2015. IEEE.

[48] Y. Zhang, Y. Wang, and X. Wang. GreenWare: Greening Cloud-Scale Data Centers to Maximize the Use of Renewable Energy. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, F. Kon, and A.-M. Kermarrec, editors, *Middleware 2011*, volume 7049, pages 143–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[49] C. Zhuo, D. Gao, Y. Cao, T. Shen, L. Zhang, J. Zhou, and X. Yin. A DVFS Design and Simulation Framework Using Machine Learning Models. *IEEE Design & Test*, 40(1):52–61, Feb. 2023.