carbond: An Operating-System Daemon for Carbon Awareness

Andreas Schmidt
Gregory Stock
Robin Ohs
Saarland Informatics Campus

Luis Gerhorst*
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Benedict Herzog
Timo Hönig
Ruhr University Bochum (RUB)

ABSTRACT

To reduce the carbon footprint of software, it is imperative that systems first become aware of their footprint. Despite various proposals to make software carbon aware via application-level development kits, we believe awareness of and adjustment to carbon-emission information is an operating-system duty—similar to how it manages time information today. In this paper, we motivate and envision `carbond`, a Linux-based service to mediate carbon information between hardware (including power supply) and application-level software. Following the recently established *Software Carbon Intensity* (SCI) standard, `carbond` deals with operational as well as embodied emissions and has different *units of work* (low level as CPU cycle up to high-level as user request) in mind. In addition to the service itself, we showcase how it can be used by application SDKs (libraries) as well as command-line utilities.

CCS CONCEPTS

• **Hardware → Impact on the environment**; Power estimation and optimization.

KEYWORDS

embodied emissions, Linux, operational emissions, power supply, sustainability

ACM Reference Format:
Andreas Schmidt, Gregory Stock, Robin Ohs, Luis Gerhorst, Benedict Herzog, and Timo Hönig. 2023. `carbond`: An Operating-System Daemon for Carbon Awareness. In *2nd Workshop on Sustainable Computer Systems (HotCarbon '23), July 9, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. `https://doi.org/10.1145/3604930.3605707`

1 INTRODUCTION

Since the early days of computing systems, operating systems (OS) are required to share available system resources among system users and processes. Time sharing [11] is the key concept to account and distribute available clock cycles of CPUs in this case—from the first multi-user systems of the 1960s [10] to the cloud-computing systems [3] of today this concept is still dominant. Today, additional resources besides time are considered in the efficient operation of computer systems and data centres, namely power and energy [24].

Modern operating systems such as Linux track the use of resources (i.e. time, power, energy) at the process-level and provide system-level interfaces (i.e. `perf(1)`) to analyse and debug resource-usage patterns of user-space applications. The availability of such system-level interfaces establishes a *resource awareness* that was previously unavailable to system programmers and users. What is yet missing, however, are methods to attribute and trace the carbon use of system activities (i.e. at process and application level).

Tracking carbon emissions is essential to keep computing accountable for their contribution to global warming—where the current projection is non-ideal [21, 7]. In contrast to other domains (e.g. transport), (a) carbon emissions cannot be measured directly for computing (cf. car emission tests), and (b) workloads can be finely grained (single car trip vs. end-user web request). Hence, we must attribute past *embodied* emissions (to produce the hardware) and current *operational* emissions (to power the hardware) to current workloads (the software providing a service). Both emission types come in (different) forms of demanded energy, which have individual carbon intensities ($\frac{\mathrm{gCO_2}}{\mathrm{kWh}}$) that must be considered.

In this paper, we discuss the constructive way towards the goal of establishing carbon awareness for software and present `carbond`, a Linux system `daemon(7)` to provide information on carbon use of software components. The contributions are three-fold:

- We discuss requirements for an operating-system service that provides carbon awareness—both for operational as well as embodied carbon.
- We propose `carbond`, a prototypical implementation of such a service together with different use cases.
- We discuss further steps towards carbon-aware computing.

Next, Sec. 2 introduces different carbon tracking metrics. Our prototypical OS service is presented in Sec. 3, followed by its use cases in Sec. 4. We close with a conclusion and outlook in Sec. 5.

2 CARBON METRICS

Before it is even possible to reduce the carbon footprint of software systems [19], it is important to accurately measure it. Unlike vehicle driving emissions, for example, the carbon footprint of a workload running on a software system cannot be directly measured and tracked. The reason for that is that there are not only the current *operational* emissions O caused by the energy consumption of the hardware but also a share of the (past) *embodied* emissions M. The latter includes all the carbon emitted during the production and disposal of the hardware device [27]. While operational emissions are highly dynamic and depend on the current carbon intensity of our energy source, embodied emissions can be estimated once and assumed to remain constant for each piece of hardware.

Essentially, carbon measurement is about measuring energy demands with different carbon intensities. This is already difficult

*Also with Ruhr University Bochum (RUB).

HotCarbon '23, July 9, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0242-6/23/07.
`https://doi.org/10.1145/3604930.3605707`

Table 1: Comparing Metrics for Carbon (Efficiency).

↓ Supported Property Metric →	PUE	CUE	GPUE	CCI	SCI
Operational Energy Efficiency	✓	✓	✓	✓	✓
Operational Carbon Efficiency	✗	✓	✓	✓	✓
Manufacturing Efficiency	✗	✗	✗	✓	✓
Reuse of Existing Devices	✗	✗	✗	✓	✓
Fine-grained Units of Work	✗	✗	✗	?	✓

as there is no single solution for tracking energy demand at all required granularities, down from a single function call up to an entire end-to-end request [1]. At the same time, reliable sources of carbon intensity are hard to find. For operational intensity, various sources exist—some of them even include scientific investigations on their adequacy.[1] For embodied intensity, manufacturers execute *Life Cycle Analysis* (LCA) (e.g. [12]) to assign a total embodied carbon footprint to a piece of hardware. Together with evaluations on the typical capacity of a piece of hardware (e.g. supported memory accesses) one is able to formulate emissions per functional unit (i.e. intensity). This relies on both the LCA as well as the expected performance to be adequate—making it an intricate task.

Historically, data-centre sustainability metrics have typically been based solely on energy consumption, ignoring the source of the energy. This changed recently due to global efforts of decarbonisation in response to climate change, highlighting the need for new metrics that focus on carbon awareness [8, 13]. Before presenting the metric we use in more detail, we provide an overview and comparison of carbon metrics that have been proposed in the literature. The ideal carbon metric should reward operational efficiency in terms of carbon and therefore energy. The metric should also reflect the embodied emissions to produce the hardware and favour the reuse of existing equipment over the purchase of new hardware, as the former is associated with lower embodied emissions. Finally, the metric should be able to accurately analyse any software application, e.g. from a large cloud system to a small personal computer. To be useful, the score should only improve as carbon emissions are reduced, thereby encouraging sustainable computing.

Our comparison is summarised in Tab. 1. The Power Usage Effectiveness (PUE) [4] metric is not exactly a carbon-aware metric—but it is widely used to evaluate the energetic efficiency of data centres. PUE is the ratio of the data centre's total energy consumption (including UPS, cooling, lighting) to the energy consumption of the IT equipment alone (i.e. necessary hardware used for computing, storage, and networking). The ideal value is 1.0, i.e. all energy is consumed by IT equipment only. As PUE only considers energy consumption and ignores the carbon intensity of the energy source, the Carbon Usage Effectiveness (CUE) [5] metric was created. Similar to PUE, it is defined as the ratio of the total CO_2 emissions caused by the total energy consumption of the data centre to the energy consumption of the IT equipment alone. Here, the optimal value is 0.0, meaning that the data centre would emit zero carbon. Green PUE (GPUE) [17] is another carbon-aware extension of PUE that considers the weighted sum of the carbon intensities of the energy sources used. Both the CUE and GPUE share that they do not incorporate embodied emissions at all. Two metrics that consider both operational and embodied emissions are the Computational Carbon Intensity (CCI) [26] and the Software Carbon Intensity (SCI) [15]. Both metrics are defined as a rate of carbon emissions per unit of computational work. This means that both metrics can be used to reason about fine-grained units of work, such as individual API requests. However, the CCI still measures the lifetime carbon impact and then provides amortised carbon emissions per operation. In contrast, the SCI is defined so that the operational emissions of arbitrary workloads can be explicitly analysed. For this reason, and because the SCI has been submitted to ISO for standardisation in February 2023, we use the SCI for `carbond`.

[1] `https://www.watttime.org/marginal-emissions-methodology/`

Software Carbon Intensity (SCI)

The SCI [15] can accurately capture the carbon footprint of a software system, including both operational and embodied emissions. SCI is a *rate* of carbon emissions for a software system, i.e. the amount of carbon emissions C (measured in gCO_2) per arbitrary but fixed unit of work R. The SCI score is calculated as $SCI = \frac{C}{R} = \frac{O+M}{R}$.

Formally, operational emissions are given by $O = E \cdot I$, where E is the total energy (in kWh) consumed by the software system and I is the current power supply's carbon intensity (in $\frac{gCO_2}{kWh}$), i.e. how much carbon was emitted to produce one kWh of energy. Embodied emissions are given by $M = TE \cdot TS \cdot RS$, where TE are the total embodied emissions, TS is the time-share (fraction of the total lifespan of the hardware used by the software), and RS is the resource-share (fraction of the total available hardware resources reserved by the software).

This last part makes measuring the carbon footprint of software systems intricate, as the correctness of the metric depends on an accurate estimate of the expected hardware lifetime. By underapproximating, we attribute a too high footprint to each workload during the lifespan and a zero footprint to workloads used beyond the hardware's lifespan. On the contrary, if the hardware fails before the expected lifespan, some embodied emissions remain that can no longer be attributed to any workload.

Therefore, `carbond` must be aware of (i) the current power supply carbon intensity I, (ii) the total energy E required to perform a task R and (iii) for any involved piece of hardware its total embodied emissions TE and anticipated lifespan as well as the respective time-share TS and resource-share RS for a given task R. In addition, `carbond` will need to translate single app-level tasks (e.g. user request or API call) into different hardware-level workloads (e.g. 10 CPU cycles or allocation of 4kB).

3 CARBOND

`carbond` forms an abstraction layer in between hardware and application software (cf. Fig. 1). With this additional layer, energy-aware software is able to access *operational* and *embodied* carbon intensity data for supported hardware components via an application-facing API provided by `carbond`. For that, `carbond` needs to collect these intensities and periodically update them if appropriate.

3.1 Interfaces

An application-facing API, accessible to all processes with read permissions, is realised by storing collected intensity data in the file system. Applications can read the files directly or use an existing

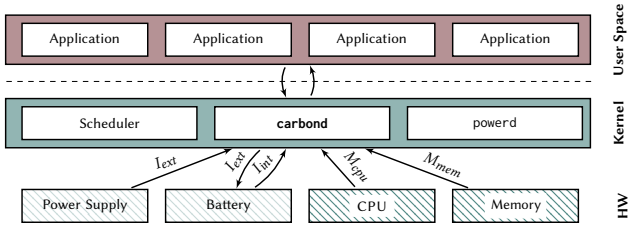


Figure 1: System architecture of carbond (embedded emissions dark hatched, operational ones light hatched).

library that implements this functionality. As some intensities are subject to change (e.g. for the power supply), applications must listen to file system updates by carbond via a subscription mechanism (e.g. by monitoring file system events with inotify).[2] The *SCI* can then be recalculated using the updated intensities.

To store *operational* and *embodied* carbon intensities in a logical structure, we create a file tree in /var/carbond containing storage files for each supported hardware component organised into categories (subdirectories) such as power, storage, network or processing. Before exposing a value for the *operational* intensity, it is important to distinguish between battery-powered systems and those without a battery. In the case where the system has no battery or has a battery but is plugged in, the *operational* intensity at a time t is equal to the intensity of the external power source $I_{ext}(t)$, while for unplugged battery-powered systems, the intensity is equal to the internal carbon intensity of the battery $I_{int}(t)$. In order to always write the currently valid *operational* intensity to the file system, carbond must know about the presence of a battery and monitor any changes in the state of charging. For *embodied* intensities, on the other hand, it is sufficient to compute and store M_c once for each component c. This value remains valid unless the component is replaced or another resource-share RS_c is applied. A CPU, for example, could have the supported unit of work *cycle* with $M_{cpu} = 10\frac{\text{ngCO}_2}{cycle}$.

3.2 Embodied Emissions

Systems are composed of several different components, each of which embodies a specific amount of carbon that was emitted during production. Moreover, each component has an[3] associated unit of work (*cycle*, 8kB *allocated memory*). Workloads running on a system consist of multiple smaller hardware tasks that can be translated into a collection of units of work. Since workloads do not always include all components, the attributed *embodied* emission M varies between workloads. For example, a workload consisting of $25\times$ *cycle* (CPU) and $5\times$8kB *allocated memory*, will involve at least the CPU and memory but not necessarily a GPU or data drive. The set of components running the smaller hardware tasks of workload R is defined as $hw_R = \{c \mid c \text{ is component and used in } R\}$ and can be used to compute M associated with R (M_R), as

$$M_R = \sum_{c \in hw_R} M_c \cdot R_c$$

where M_c is the *embodied* emission of component c, and R_c is the number of performed units of work of c. Referring back to *SCI*, we have $M_c = TE_c \cdot TS_c \cdot RS_c$.

Unfortunately, TE_c and TS_c are not known and must be determined. For TE_c, possible solutions include having each device supply such a value itself, up to defining the values in a configuration file, which are then picked up by carbond. Estimating realistic amounts of *embodied* carbon is an open research question and is the focus of papers such as [20]. Furthermore, to determine the component's time-share TS_c the projected lifetime T_{BC} of the component needs to be approximated. Having an estimation of T_{BC} and the duration T—equal to the time one unit of work takes to execute—it is possible to calculate $TS_c = \frac{T}{T_{BC}}$. Given a resource-share RS_c (e.g. $\frac{1}{4}$ if using 1 of 4 cores of a CPU), the component's *embodied* emission is calculated as $M_c = TE_c \cdot \frac{T}{T_{BC}} \cdot RS_c$, given in gCO_2 divided by the components unit of work. Substituting the formula for M_c into the formula for M_R, the total *embodied* emission of the workload R is equal to

$$M_R = \sum_{c \in hw_R} TE_c \cdot \frac{T}{T_{BC}} \cdot RS_c \cdot R_c$$

3.3 Operational Emissions

In operation, the only way for a computing system to cause emissions is by operational energy. This energy can either be supplied through an external source (i.e. system is plugged in) or a battery storage (i.e. system is unplugged). carbond should handle these two cases transparently, hence the current carbon intensity is either the intensity of the external source or the battery. For the external source, we use WattTime or ElectricityMap to retrieve localised data—other options are discussed in Sec. 5.

As of today, there is no notion of carbon intensity of battery-driven power supply. Our proposal is to leverage the intensity of the external source and track the charging of the battery. While the battery is charged with energy that can later be used, it also charges emissions (proportional to the current intensity and the amount of charged energy). When we later discharge the battery, the intensity becomes this accumulated intensity of the battery.

Formally, the battery must keep track of its current energy level E_{bat} (measured in J or kWh) and the absolute value of its accumulated carbon emissions C_{bat} (measured in gCO_2). The internal carbon intensity of the battery at a given time t can then be calculated as $I_{int}(t) = C_{bat}(t)/E_{bat}(t)$.

When a battery is charged with a given power $P(t)$ (measured in W), its energy level at time t depends only on the power and can be calculated as

$$E_{bat}(t) = E_0 + \int_0^t P(t)\,\mathrm{d}t$$

where E_0 is the initial energy level in the battery at time $t = 0$. This formula also holds in the case of discharge, i.e. when $P < 0$.

Similarly, the carbon emissions inside the battery increase during charging. However, the slope of the increase depends on the current carbon intensity of the external energy source. It is given by

$$C_{bat}(t) = C_0 + \int_0^t P(t) \cdot I_{ext}(t)\,\mathrm{d}t$$

[2] https://manpages.ubuntu.com/manpages/bionic/man7/inotify
[3] One for now, later we could imagine providing a catalogue of work units.

where C_0 is the initial amount of carbon (in gCO_2) in the battery at time $t = 0$.

Discharging the battery, i.e. using the energy to power some task, will cause the internal amount of carbon to decrease. However, the internal carbon intensity remains constant during the discharge period, because no external energy is added. Formally, $C_{bat}(t)$ is given by

$$C_{bat}(t) = C_0 + \int_0^t P(t) \cdot I_{int}(t)\,\mathrm{d}t = C_0 + \frac{C_0}{E_0}\int_0^t P(t)\,\mathrm{d}t$$

Of course, charging and discharging can be done simultaneously. However, there are several ways to handle this. In this paper, we have decided to always consider the battery as a buffer between the external power source and the hardware to be powered. This simplifies the calculations and allows us to use the canonical combination of the two formulae for charge and discharge. In some situations, however, this behaviour may be undesirable. For example, consider a battery with a currently high I_{bat} that is now being charged with carbon-free energy and simultaneously being used to power some task. In our scenario, the green energy would be used to improve I_{bat} over time, and the task would be attributed with the dirty energy from the battery. Which modelling strategy to use is a point for future discussion.

Fig. 2 illustrates this. Between time t_0 to t_1, the battery is charged with a constant rate $P(t)$. Then, the external power source is disconnected, and the energy stored in the battery is used to power a task (again at constant power) until it is completely depleted at t_2. The top graph shows the carbon intensity I_{ext} of the external power source over time for the relevant charging period. Assuming a piecewise constant step function is realistic, e.g. due to rate limits of the external data source. The same graph also shows the internal carbon intensity I_{int} of the battery. Starting from an initial intensity, given by the ratio of the initial amount of carbon C_0 to the stored energy E_0, I_{int} follows the movement of I_{ext} and remains constant once the external power source is disconnected at t_1. The lower graph shows the state of charge E_{bat} over time as well as the absolute value of the accumulated carbon emissions C_{bat}. Since the (dis)charging power is constant, the stored energy evolves linearly. However, the stored carbon emissions increase irregularly, depending on the current I_{ext}. During discharging, C_{bat} and E_{bat} decrease linearly in such a way that their ratio remains constant and that they reach zero at exactly the same time t_2.

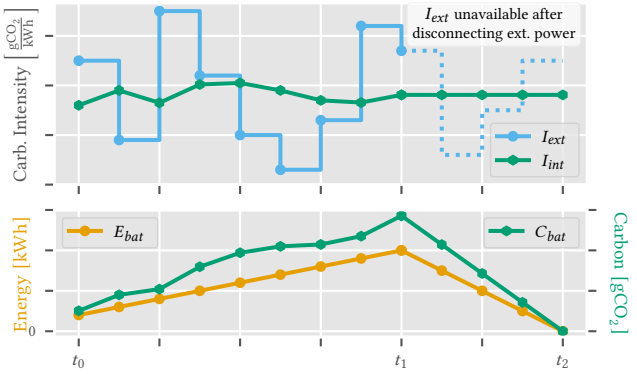


Figure 2: Example of the operational emissions of a battery.

4 CARBOND IN USE

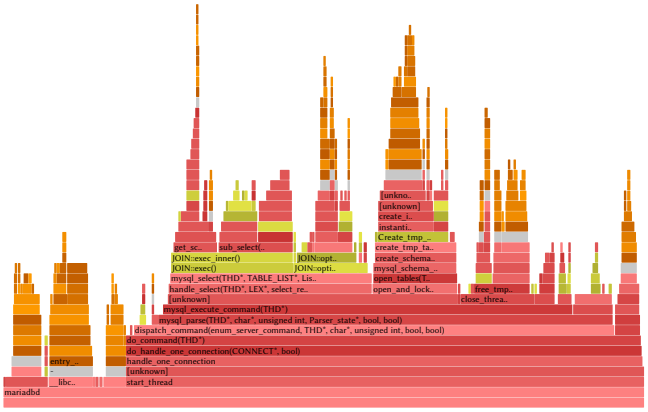
We envision use cases at different levels of abstraction for `carbond`:

Sub Process Level. As mentioned in [1], there is interest in tracking carbon down to the level of *requests* (e.g. web requests). Hence, there must be libraries for common programming languages that (a) track resource consumption on a logical level (e.g. used memory) and (b) query `carbond` to compute how much carbon was emitted by a certain request. Our Rust library `resource-gauge` [23] could leverage this, as it already allows to annotate functions but also data with resource requirements—generating runtime monitoring code where necessary. Similar monitoring approaches could be created for carbon; including the enforcement of certain emission limits.

Process Level. One possible use case for `carbond` at process level is the generation of *flame graphs* [16]. Flame graphs are a well-known tool to visualise the resource demand of applications, for example, CPU time or memory demand on a per-function level. Subfig. 3a shows an example of a flame graph analysing a *mariadb* database serving a calendar server. From the flame graph follows that in this instance most connections handle `SELECT` queries. Unsurprisingly, the `JOIN` operations are main contributors to the CPU time for handling a `SELECT`.

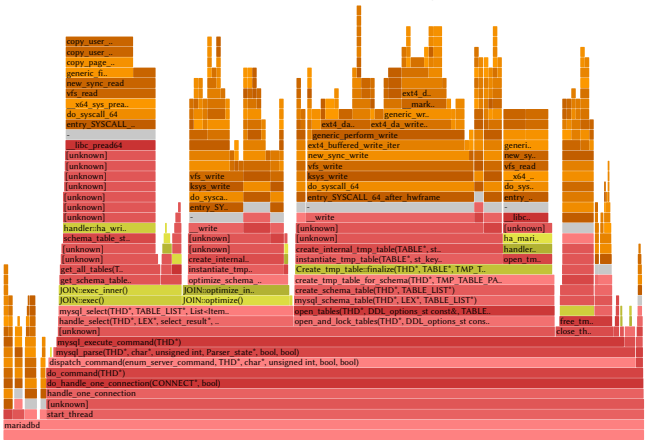
Although already distinctly helpful for developers to detect hot spots (i.e. excessive CPU time spending), flame graphs in the current form provide no insight into the carbon emissions of application parts. Subfigs. 3b and 3c exemplary show flame graphs visualising the carbon emissions of the same database as shown in Subfig. 3a. Both carbon-emission flame graphs are not based on actual values but act as an example of how we envision a flame graph visualisation enriched with carbon-emission data as provided by `carbond`.

Subfigs. 3a and 3b show the same database scenario visualising CPU time and carbon emissions, respectively. As block IO operations require a dedicated block device (e.g. HDD, SSD), the respective IO operations must also account for the block device's carbon emissions. Hence, diagrams of CPU time and carbon emissions vary as significantly as shown in Fig. 3. In particular, functions handling filesystem access (e.g. `vfs_write()` (orange)) show significantly wider frames. In the flame-graph notation, a wider frame denotes more usage of the visualised resource (i.e. carbon emissions in this case). This can be seen, for example, by looking at the `mysql_execute_command()` (red) function. The biggest CPU-time contributor for this function is the `JOIN::exec()` (green) function. For the carbon emissions, however, the contribution of `JOIN::exec()` is minor compared to `JOIN::optimize()` (green) and `Create_tmp_table::finalize()` (green) so that the function `JOIN::exec()` is not even shown anymore in the flame graph. The reason for this is that `JOIN::exec()` involves no filesystem access, whereas the other two functions write temporary results to the filesystem. Hence, the hot spots for spent CPU time and for carbon emissions may vary significantly, and flame graphs enriched with `carbond` data can help developers to identify the carbon-emission hot spots in their applications. Additionally, it is possible to account



(a) CPU Time

(b) Carbon Emissions (High)



(c) Carbon Emissions (Low)

Figure 3: Flame graphs showing a *mariadb* database serving a calendar server. Subfig. 3a shows the CPU time distribution and Subfigs. 3b and 3c show how we envision flame graphs visualising the carbon emissions of the database at different points in time (not based on actual data).

for different carbon-emission levels depending on the current footprint of operational emissions. To this end, Subfig. 3c shows a flame graph visualising the carbon emissions at a point in time with a more favourable footprint for the operational emissions.

Besides flame graphs, a natural use case for `carbond` is the performance analysis tool `perf`. `perf` is a CLI tool capable of deep analyses of an application's performance. For example, by the use of *Performance Monitor Counters* (PMCs), an inspection of the cache hit rate, branch-prediction performance, and even the energy consumption (using the *Intel Running Average Power Limit* interface) is possible. `carbond` provides the building blocks to expand `perf`'s analysis capabilities with a convenient and well-known way to determine the carbon emissions of running an application.

Existing Tools. We believe that the plethora of carbon awareness and carbon-aware tools that are developed right now can benefit from a common platform such as `carbond`. The Carbon Aware SDK [14] currently has no support for embodied carbon and could hence use our approach. A programming model such as Eco [28] as well as a programming framework such as Green [6] could draw carbon data from `carbond`. In a similar fashion, programming language extensions such as EnerJ [22] and Energy Types [9] could use cleanness states (Dirty/Clean) instead of power states (High/Low) to adapt to at runtime (i.e. by using approximate computing in unsustainable scenarios). Recently, there is an increased interest in *Performance Interfaces* (e.g. [18]), which could not only predict cycles / energy of certain blocks of code, but a carbon footprint directly—based on `carbond` information. We further believe that the virtual energy system Ecoviser [25] can work together with `carbond` to provide its service to containerised applications—or integrate with Treehouse [2]. Finally, integration into carbon-aware networking approaches [29] is of great interest.

5 CONCLUSION & OUTLOOK

In this paper, we presented our design for *carbon awareness as an operating-system service*. Our prototype `carbond` leverages the mathematical framework of the Software Carbon Intensity to retrieve, measure, manage, and expose intensities of both the current power supply as well as individual hardware components. We motivated multiple usage cases for `carbond` to support both carbon-aware applications as well as other operating systems services. We envision future work to be done in the following areas:

Data Sources for Intensity and Embodied Carbon. While `carbond` is agnostic to the quality of its data sources, only realistic data fosters carbon awareness. In an ideal future, both the current carbon intensity of grid energy as well as the embodied carbon footprint of a piece of hardware are available openly and free of charge.

Network Carbon Protocol. Given the increasing importance of carbon intensity data, we suggest the development of the `ncp` (Network Carbon Protocol). Being similar to `ntp` for time but leveraging the smart grid to exchange information. For example, a smart meter could broadcast the current power intensity into the home network, making every device aware. In houses with access to other sources than residential supply, the fact that currently power is coming from own solar energy can be broadcast to devices.

Carbon-Aware Battery. We sketched out an OS-level solution to track the carbon intensity of a battery across charging and discharging. However, this ignores the situation where no OS is running to detect charging or passive discharging (due to storing the battery unused). Ideally, future batteries come with a similar solution to ours that enables them to track their carbon intensity.

ACKNOWLEDGMENTS

This work was funded by the German Research Foundation (DFG) project number 502228341 ("Memento"), 465958100 ("NEON"), and 389792660 as part of TRR 248 – CPEC[4] and from the Bundesministerium für Bildung and Forschung (BMBF, Federal Ministry of Education and Research) in Germany for the project AI-NET-ANTILLAS 16KIS1315. Luis Gerhorst's contribution to this work was made prior to his employment at Amazon.

[4] https://perspicuous-computing.science

AVAILABILITY

The source code of `carbond` is published under the MIT license and freely available at https://doi.org/10.5281/zenodo.8063846.

REFERENCES

[1] Vaastav Anand, Zhiqiang Xie, Matheus Stolet, Roberta De Viti, Thomas Davidson, Reyhaneh Karimipour, Safya Alzayat, and Jonathan Mace. 2022. The Odd One Out: Energy is not like Other Metrics. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[2] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. 2022. Treehouse: a case for carbon-aware datacenter software. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[3] Michael Armbrust et al. 2009. Above the Clouds: A Berkeley View of Cloud Computing. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, (Feb. 2009).

[4] Victor Avelar, Dan Azevedo, and Alan French. 2012. PUE™: A Comprehensive Examination of the Metric. White paper #49. The Green Grid.

[5] Dan Azevedo, Michael Patterson, Jack Pouchet, and Roger Tipley. 2010. Carbon Usage Effectiveness (CUE): A Green Grid Data Center Sustainability Metric. White paper #32. The Green Grid.

[6] Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*, 198–209. DOI: 10.1145/1806596.1806620.

[7] Noman Bashir, David Irwin, Prashant Shenoy, and Abel Souza. 2022. Sustainable computing–without the hot air. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[8] Andrew A. Chien, Chaojie Zhang, Liuzixuan Lin, and Varsha Rao. 2022. Beyond PUE: Flexible datacenters empowering the cloud to decarbonize. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[9] Michael Cohen, Haitao Steve Zhu, Senem Ezgi Emgin, and Yu David Liu. 2012. Energy types. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012)*, 831–850. DOI: 10.1145/2384616.2384676.

[10] Fernando J Corbató and Victor A Vyssotsky. 1965. Introduction and overview of the multics system. In *Proceedings of the AFIPS Fall Joint Computer Conferences Joint Computer Conference, Part I*. DOI: 10.1145/1463891.1463912.

[11] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. 1962. An experimental time-sharing system. In *Proceedings of the AIEE/IRE Spring Joint Computer Conference*, 335–344. DOI: 10.1145/1460833.1460871.

[12] Fujitsu. 2021. Product life cycle assessment of fujitsu esprimo p9010 desktop pc. (2021). https://www.fujitsu.com/global/documents/about/environment/Life%20cycle%20analyses%20of%20Fujitsu%20Desktop%20ESPRIMO%20P9010%20June%202021.pdf.

[13] Anshul Gandhi et al. 2022. Metrics for sustainability in data centers. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[14] Green Software Foundation. 2023. Carbon Aware SDK. (2023). https://github.com/Green-Software-Foundation/carbon-aware-sdk.

[15] Green Software Foundation. 2021. Software Carbon Intensity Standard. Version 1.0.0. (Nov. 1, 2021). https://github.com/Green-Software-Foundation/sci/blob/main/Software_Carbon_Intensity/Software_Carbon_Intensity_Specification.md.

[16] Brendan Gregg. 2016. The flame graph. *Communications of the ACM*, 59, 6, 48–57. DOI: 10.1145/2909476.

[17] Fawaz Al-Hazemi, Alaelddin Fuad Yousif Mohammed, Lemi Isaac Yoseke Laku, and Rayan Alanazi. 2019. PUE or GPUE: A carbon-aware metric for data centers. In *Proceedings of the 21st International Conference on Advanced Communication Technology, ICACT 2019*. IEEE, 38–41. DOI: 10.23919/ICACT.2019.8701895.

[18] Rishabh R. Iyer, Katerina J. Argyraki, and George Candea. 2022. Performance interfaces for network functions. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*.

[19] Colleen Josephson, Nicola Peill-Moelter, Zhelong Pan, Ben Pfaff, and Victor Firoiu. 2022. The sky is not the limit: untapped opportunities for green computing. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[20] Donald Kline Jr., Nikolas Parshook, Xiaoyu Ge, Erik Brunvand, Rami G. Melhem, Panos K. Chrysanthis, and Alex K. Jones. 2019. Greenchip: A tool for evaluating holistic sustainability of modern computing systems. *Sustainable Computing: Informatics and Systems*, 22, 322–332. DOI: 10.1016/j.suscom.2017.10.001.

[21] Mark Pesce. 2021. Cloud computing's coming energy crisis – the cloud's electricity needs are growing unsustainably. *IEEE Spectrum*. https://spectrum.ieee.org/cloud-computings-coming-energy-crisis.

[22] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, 164–174. DOI: 10.1145/1993498.1993518.

[23] Andreas Schmidt, Luis Gerhorst, Kai Vogelgesang, and Timo Hönig. 2023. ResourceGauge: enabling resource-aware software components. In *Proceedings of the 17th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (ECRTS-OSPERT 2023)*.

[24] Amit Sinha. 2001. *Energy efficient operating systems and software*. Ph.D. Dissertation. Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/86773.

[25] Abel Souza, Noman Bashir, Jorge Murillo, Walid A. Hanafy, Qianlin Liang, David E. Irwin, and Prashant J. Shenoy. 2023. Ecovisor: A virtual energy system for carbon-efficient applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*, 252–265. DOI: 10.1145/3575693.3575709.

[26] Jennifer Switzer, Gabriel Marcano, Ryan Kastner, and Pat Pannuto. 2023. Junkyard computing: repurposing discarded smartphones to minimize carbon. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*, 400–412. DOI: 10.1145/3575693.3575710.

[27] Swamit Tannu and Prashant J. Nair. 2022. The dirty secret of SSDs: embodied carbon. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.

[28] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. 2015. A programming model for sustainable software. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 767–777. DOI: 10.1109/ICSE.2015.89.

[29] Noa Zilberman, Eve M Schooler, Uri Cummings, Rajit Manohar, Dawn Nafus, Robert Soulé, and Rick Taylor. 2022. Toward Carbon-Aware Networking. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon 2022)*.