Energy-Aware Process Scheduling in Linux

ABSTRACT

Datacenters are becoming one of the most significant worldwide consumers of electricity and sources of carbon emissions. With the end of Dennard's scaling, and as cloud datacenter power usage effectiveness becomes close to optimal, we can no longer rely on hardware advancements alone to sustainably meet growing computational needs. System software must play a bigger role in optimizing energy consumption of applications. We argue the operating system (OS), at the heart of datacenter resource allocation and scheduling decisions, must be made energy-aware.

This paper has two goals. First, we show how Linux can be made energy-aware without making any kernel changes, by introducing a new energy accounting framework, Wattmeter. Wattmeter uses eBPF functions to efficiently measure per-process energy consumption at millisecond-scale granularity with low overhead. Second, we show how this information can be used to make energy-informed scheduling decisions, with two proof-of-concept scheduling policies: a policy that equalizes energy across processes, and one that caps the amount of energy that can be consumed by a process. This paper represents a first step in making the operating system energy-aware, and demonstrating how that capability can be used to control applications' energy consumption.

CCS CONCEPTS

• **Software and its engineering** → **Scheduling**; • **Hardware** → **Power estimation and optimization**.

KEYWORDS

energy-aware scheduling, sustainable computing, eBPF

ACM Reference Format:

. 2024. Energy-Aware Process Scheduling in Linux. In *Proceedings of HotCarbon'24.* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Datacenters are already estimated to account for 1-2% of global electricity consumption and carbon emissions [16]. At the same time, application compute demands are growing exponentially, especially with the rise of AI. With the end of Dennard's scaling, we can no longer rely solely on hardware-level energy efficiency to curb the rise of datacenter energy usage [2].

In this work, we focus on making the Linux operating system (OS), the fundamental software abstraction underlying datacenter applications, energy-aware. In particular, the OS scheduler plays a pivotal role in deciding which processes and threads get access to hardware resources at millisecond-scale timescales.

We target two primary challenges. First, how can the scheduler be made aware of the energy consumption of specific processes at a sufficiently-fine granularity to make online scheduling decisions, without incurring a high overhead. Second, can we design new scheduling policies that take the energy consumption of processes into account when making scheduling decisions, without making kernel changes.

To tackle the first challenge, we observe that modern Intel and AMD servers provide system-level millisecond-scale energy readings at the socket level, via running average power limits (RAPL) [8]. To estimate per-process energy consumption, systems such as Scaphandre [14], Kepler [1], and EnergAt [12] attribute RAPL readings to each process based on their respective CPU time and resource utilization. However, these estimates are not granular enough for the process scheduler because they monitor energy by reading RAPL counters from userspace at much coarser time scales (see §2.1). If we configure these tools to monitor power at millisecond-scale frequencies they would incur very frequent context switches, consuming excessive CPU and hurting application runtimes.

In contrast to these systems, our approach is to measure RAPL counters and store their state within the kernel, without requiring a context switch to fetch each measurements. We implement a novel energy accounting framework, Wattmeter, which utilizes the eBPF framework [10] to read energy measurements from RAPL upon every context switch. This allows for real-time, precise energy measurements at the scheduling-event level.

To tackle the second challenge, we leverage recent frameworks that allow Linux's scheduling policies to be customized from userspace [6, 15, 19]. As a proof of concept of how process scheduling policies can be made energy-aware, we developed two simple scheduling policies on top of Google's extensible userspace scheduling framework, ghOSt [15]. Our policies include energy-fair scheduler (EFS), which equalizes energy share between processes, and energy-capped scheduler (ECS), which allows users to set limits on how much energy each process can use. Implementing these policies requires no further kernel changes beyond the initial installation of the custom ghOSt kernel. With the potential future integration of `sched_ext` [6] into the official Linux kernel, similar `sched_ext`-based policies could be implemented without any kernel modifications at all. We evaluate EFS and ECS, and show they can equalize and limit the amount of energy each process consumes, respectively, compared to the default Linux scheduling policy, CFS, which cannot control process energy consumption.

2 BACKGROUND

This section provides background on the tools that allow us to make the Linux scheduler energy-aware without making kernel changes: RAPL (§2.1), eBPF (§2.2) and scheduling frameworks like ghOSt (§2.3). Afterwards, we demonstrate experimentally that even

HotCarbon'24, July 9, 2024, Santa Cruz, CA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

processes that are allocated the same CPU time may exhibit widely-varying energy consumption, due to their different CPU, memory and I/O access patterns (§2.4).

2.1 Energy Measurement Tools

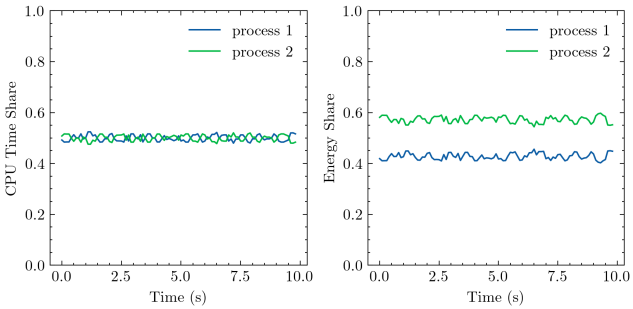
RAPL is a power management interface first implemented by Intel in the Sandy Bridge architecture in 2011, for monitoring and controlling energy use across computer components without extra hardware [8]. AMD also implements this measurement interface, starting with its first Zen architecture in 2017. RAPL defines several power domains that corresponds to specific hardware components, such as *core* (CPU cores), *uncore* (integrated GPUs), *pkg* (whole CPU Package), *dram* (DRAM), and *psys* (the whole platform). Each domain corresponds to specific hardware components, allowing detailed monitoring and management of energy usage across different parts of a computing device. The energy measurements are usually updated at an interval of approximately $976\mu s$. RAPL exposes the measurement through model-specific registers (MSR), which can be accessed through higher-level OS interfaces such as Linux's `powercap` and `perf_event` [25].

Higher-level power measurement tools like Scaphandre [14], Kepler [1], and EnergAt [12] utilize RAPL to estimate per-process energy consumption via a userspace polling mechanism, attributing energy measurements based on CPU runtime statistics found in Linux (`procfs`). However, since the polling process runs in userspace, millisecond-scale measurements are only possible if the polling process is scheduled to run precisely every millisecond (immediately after RAPL updates). There is no guarantee that their polling process would get scheduled so frequently, and even if it did, the constant context switching would incur a very significant overhead. As a result, these tools cannot offer accurate millisecond-scale energy measurements.

Wattmeter avoids this pitfall by using eBPF functions that run within the kernel, avoiding the need to switch back to userspace and avoiding the need to dedicate a thread to busy-polling.

2.2 eBPF

eBPF [10] is a Linux- and Windows-supported framework that enables userspace applications to offload simple functions into the kernel space. At installation time, these functions are verified to ensure their safety. For example, the eBPF verifier checks that the functions do not exceed a specified instruction limit, contain unbounded loops, or perform out-of-bounds memory accesses. Tracepoints are static hooks in the Linux kernel that eBPF programs can attach to. When a traced event happens, eBPF programs attached to the tracepoint are activated to execute and gather data associated with the event. eBPF also facilitates safe interaction between userspace applications and kernel eBPF programs via shared data structures known as *eBPF maps*. In the context of Wattmeter, eBPF programs are deployed at various scheduler tracepoints to keep account of each process's precise execution timeslice and its corresponding energy consumption (§3).



(a) Time share **(b) Energy share**

Figure 1: Two processes that get scheduled the same CPU time by CFS but significantly differ in their energy consumption.

2.3 Linux Scheduling Customization Frameworks

ghOSt is a scheduling framework developed by Google that delegates kernel-level scheduling decisions to userspace policies in Linux systems, thereby enabling flexible and rapid adaptation to the dynamic needs of datacenter workloads [15, 19]. It modifies the Linux kernel to introduce a ghOSt scheduler class that can be instantiated from userspace with any custom scheduling policy. The framework supports communication, abstraction, and action protocols that allow user programs to express complex policies with minimal overhead. By installing the ghOSt kernel, we implement our energy-aware scheduling policies directly in userspace (§4.1), without having to modify Linux kernel code.

Another framework for implementing custom scheduling policies without direct kernel modification is Meta's `sched_ext`. `sched_ext` is a Linux kernel patch that proposes a new abstract scheduler class, which can be instantiated with scheduling programs in eBPF at runtime. There are ongoing discussion of upstreaming `sched_ext` into Linux.

2.4 Motivation: Task Energy Variability

Schedulers commonly try to ensure fairness by simply equally splitting the amount of time the tasks get allocated on a particular computing resource. This approach is exemplified by CFS, which tries to equalize the amount of CPU time allocated to each process. However, the amount of time given to a process is not always a good predictor of the amount of energy that process will consume.

We demonstrate this discrepancy between CPU and energy consumption with a simple experiment. Figure 1 shows the energy consumption of two processes that are run on Linux with CFS, which allocates each of the processes roughly the same CPU time. The two processes that execute two Python programs. Process 1 is memory-intensive: it repeatedly reads and writes large byte-arrays into heap memory, while process 2 is CPU-intensive: it repeatedly computes SHA256 on long strings (see §5 for the full evaluation setup). We measure each process's energy consumption using Wattmeter.

The experiment shows that despite the fact that both processes are scheduled almost identical CPU time, process 2 consumes about 50% more energy than process 1. In general, variability in energy

consumption can stem from factors such as the usage of different CPU instructions, memory consumption, and the cache hit rate [17]. Linux today has no awareness of this variability in energy consumption across processes.

One can imagine energy could be taken into account in different scheduling policies. For example, one may want to lower the total energy consumption across the entire system by intelligently scheduling processes on each node. In §4.1, as a proof of concept, we use a case study of a simple policy that simply tries to equalize the energy consumption across processes and prioritize more energy-efficient processes, while keeping CPU utilization high. One can also imagine that in multi-tenant environments, administrators would like to cap the amount of energy each process consumes (e.g., imposing `cgroup` energy limit). In §4.2, we implement another policy that tries to limit the power usage of individual processes.

3 WATTMETER: MILLISECOND-SCALE PER-PROCESS POWER ACCOUNTING

Unlike existing approaches [1, 12, 14], which measure the coarse-grained system-wide energy and attribute the energy consumed by each process according to their CPU time, we want to directly measure the energy usage for each execution interval, from the moment a process is scheduled until it is de-scheduled. More precisely, for any process p, its energy consumption is given by

$$E(p) = \int_{t_0}^{t_n} \text{on\_cpu}(p, t) \cdot (W_t - W_{\text{static}}) \; dt$$

where $[t_0, t_n]$ is the measurement period, $\text{on\_cpu}(p, t) \in \{0, 1\}$ denotes whether p is running on the CPU at moment t, and W_t, W_{static} are the current power and the system's static power, respectively. In practice, dt is RAPL's measurement granularity, which is approximately $1ms$.

To measure power at a millisecond-scale granularity, Wattmeter utilizes eBPF's tracepoint mechanism, which triggers eBPF programs instantaneously, without having to wait for the measurement program itself to be scheduled. The advantage of this approach is that it incurs much lower overhead than reading the data from userspace, because it eliminates the need for context switching, and it can measure power at a much finer time granularity. In addition, eBPF functions are defined by userspace applications and do not require kernel changes.

Challenges. There are several technical hurdles in implementing Wattmeter's functionality in eBPF. First, eBPF access restrictions prevent direct access to the RAPL interface. Therefore, Wattmeter obtains energy measurements through a `perf_event` that exposes RAPL data. However, eBPF programs cannot directly create `perf_event` objects. To overcome this second challenge, the userspace *agent*, which runs the scheduler (see more details below), opens a `perf_event` and passes it to Wattmeter through an eBPF `BPF_MAP_TYPE_PERF_EVENT_ARRAY` map, which then reads from the `perf_event` on the user's behalf. This mechanism ensures that Wattmeter can effectively measure energy consumption without violating eBPF's access constraints.

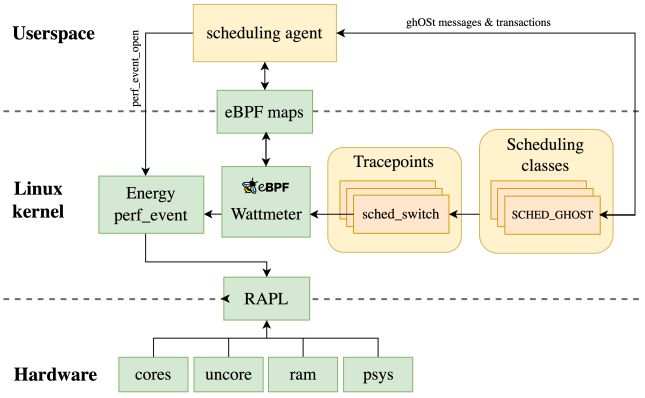


Figure 2: System diagram. Energy measurement components are green (§3); Scheduling components are yellow (§4).

Architecture. Wattmeter is an eBPF program that attaches to the `sched_switch` tracepoint (Figure 2). It maintains the following eBPF maps, accessible to the userspace program:

(1) `total_consumption`: keeps track of the system-wide energy consumption and CPU time.
(2) `pid_to_consumption`: associates each process's PID with its accumulated energy consumption and CPU time.

Upon each context switch, `sched_switch` activates Wattmeter to update its states. The program first updates `total_consumption` with the current energy and time. It then attributes the energy and CPU time since the last context switch to the process that is being switched off of the CPU in `pid_to_consumption`.

Hardware support. At the moment, Intel only supports CPU-socket-level measurement, and not per-core measurements. With this hardware limit, Wattmeter is unable to obtain accurate per-process energy consumption for parallel processes running on CPUs within the same socket. Fortunately, AMD's Zen architecture does offer per-physical-core energy counters, and there has been ongoing discussions and development for Linux support for exposing these data through `perf_event` [20]. We hope that this paper further motivates adding support for per-core RAPL measurements.

4 CASE STUDIES

This section describes the design of two proof-of-concept energy-aware scheduling policies that utilize energy data measured using Wattmeter. Both polices are implemented as a userspace ghOSt agent that communicates with the kernel-based ghOSt scheduling class to execute scheduling decisions, and interacts with Wattmeter and the rest of the system as shown in Figure 2.

4.1 Energy-Fair Scheduling

Linux's default scheduler, CFS, tries to provide each process a fair share of CPU time, while maximizing CPU utilization, but it does not take into account that even if the time share of different processes is relatively similar, their energy consumption may vary considerably (see §2.4). Equipped with Wattmeter, we present a modification of CFS that instead equalizes energy share across processes.

EFS has a simple goal of balancing energy across processes, while maintaining high CPU utilization. The latter is important, because it ensures tasks make forward progress (e.g., a trivial solution to equalizing energy would be to simply shut the entire server down). To do so, EFS builds on top of CFS to take power into account. The CFS algorithm keeps track of the virtual runtime (*vruntime*) of each process, which is the real runtime weighted by each process's *nice value*. At each scheduling decision, CFS picks the process with the lowest *vruntime*. EFS works similarly: it first computes *vruntime*, and the difference is that it further scales *vruntime* with the an additional coefficient *vpower* to compute *venergy*, which is used to make scheduling decisions in place of *vruntime*. In precise terms, the update rule for *venergy* is

$$\Delta venergy_t(p) = \Delta vruntime_t(p) \cdot vpower_t(p),$$

for each timestep t and each process $p \in \mathscr{P}$ that is managed by EFS. Then, EFS picks process p where

$$p = \arg\min_{p \in \mathscr{P}} venergy(p).$$

To compute $vpower_t(p)$, the EFS agent keeps an exponential moving average (EMA) of p's power in Watts, given by

$$\mathcal{P}_t^{\text{EMA}}(p) = (1-\gamma) \cdot \mathcal{P}_{t-1}^{\text{EMA}}(p) + \gamma \cdot (W_t - W_{\text{static}}),$$

where W_t is computed from Wattmeter's `pid_to_consumption` eBPF map and γ the smoothing factor of EMA. The usage of EMA allows us to compute a process's overall power by taking into account both historical and new energy readings, and γ is a tunable parameter that controls how much emphasis is put on the new data points.

At each timeslice, the statistics of the currently-running process is sufficient to update the these quantities. Then, $vpower_t(p)$ is given by

$$vpower_t(p) = \frac{\mathcal{P}_t^{\text{EMA}}(p)}{\sum_{p \in \mathscr{P}} \mathcal{P}_t^{\text{EMA}}(p)}.$$

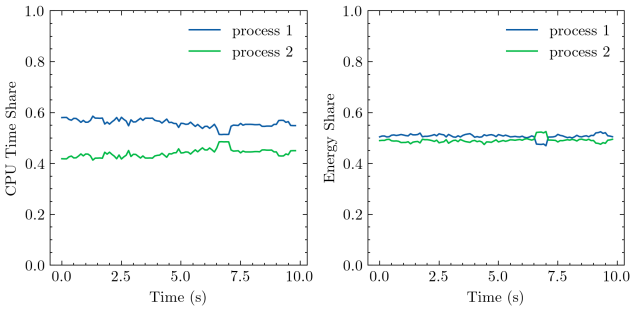
In other words, $vpower_t(p)$ captures the proportion of system-wide power consumed by process p.

4.2 Energy-Capped Scheduling

Control groups (`cgroup`) play a pivotal role in resource management and isolation. They allow system administrators to allocate, prioritize, deny, and manage system resources—such as CPU time, memory, network bandwidth, and disk I/O, among user-defined groups of processes. This mechanism is commonly used in multi-tenant environments like Docker and Kubernetes, as it provides the underlying functionality necessary for limiting and isolating resource usage. However, `cgroup` does not offer energy-related allocations or management. Using precise energy measurements from Wattmeter, we design a process-level energy-capping mechanism.

Similar to `cgroup`, users can set an energy limit on individual processes by providing a time window T and a limit value E_{limit}. The energy-limited process should be allowed to consume a maximum of E_{limit} Joules within any time window of T seconds.

We implement energy capping by performing a "limit check" at each scheduling decision. By keeping track of each process's energy consumption in a moving window of T seconds, we can determine whether a process has consumed more than E_{limit} Joules in the current time window. When picking the next task to run, ECS only picks from the subset of processes that have not reached their limits.



(a) Time share **(b) Energy share**

Figure 3: EFS equalizes the energy consumption of two processes with different power consumption profiles.

5 EVALUATION

For our experiments, we use a machine equipped with RAPL-enabled processor: 11th Gen Intel Core i5-1135G7 running at 2.40GHz. In Wattmeter, we measure RAPL's *pkg* domain. Due to Intel's lack of per-core energy domain, we ran our experiments on a single core. We plan in the future to run multicore experiments on an AMD machine that supports RAPL per-core energy domains.

In experiments for Figures 1, 3 and 5, we spawned the same two processes described in §2.4. In the experiment for Figure 4, we ran dynamic workloads that alternate between process 1 and process 2. All of our experiments execute for a period of 10 seconds.

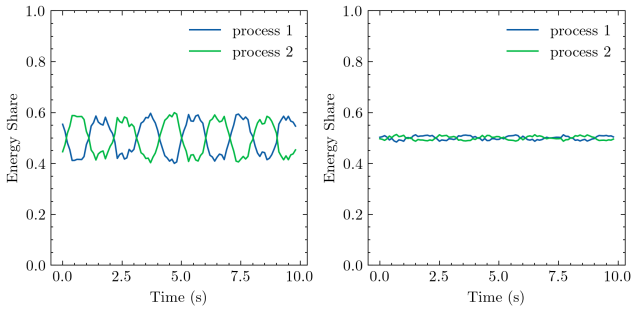
5.1 Energy Fairness

Figure 3 shows the effect of EFS on the CPU and energy shares between the two processes. With an interval of 0.1 seconds, we measured how much time and energy has been used by each process. Then, the time and energy share of the process was computed using a sliding window of 0.5 seconds. Since power consumption differs across the processes, EFS balances energy share by prioritizing the more energy-efficient process (process 1). Compared with CFS (see Figure 1), EFS allocates CPU time differently to achieve energy fairness. In contrast to CFS, which keeps the processes consuming different power throughout the experiment, EFS equalizes the energy share. Note that across the entire experiment, both EFS and CFS fully utilize the CPU core.

Figure 4 demonstrates EFS's ability to balance energy between processes with dynamically changing behaviors. The graph for CFS shows a drastically alternating energy consumption pattern when their CPU time is equalized. EFS is able to handle such dynamic workloads well and keep the energy share close to fair at all times.

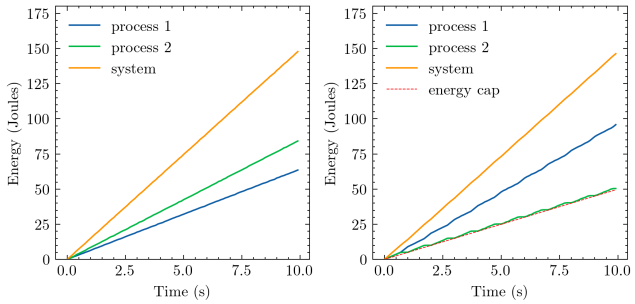
5.2 Energy Capping

Figure 5 presents the results for an experiment with the same two processes, where we capped process 2's energy usage to 5 J/s. The red line in Figure 5b represents the energy cap, and the experiment shows that ECS is effective in limiting process energy consumption



(a) CFS energy share **(b) EFS energy share**

Figure 4: EFS can respond to dynamic power consumption of workloads.



(a) CFS **(b) ECS, process 2 capped at 5 J/s**

Figure 5: ECS with energy capping for process 2.

according to user specification. We show the consumption patterns under CFS in Figure 5a, without any energy limits. Since process 2 is more power-intensive, it consumes more energy within the sampling period.

5.3 Overhead of Wattmeter

To measure how much overhead is introduced by Wattmeter and our scheduling policies, we compare the throughput of a system equipped with EFS with one equipped with CFS to run the same tasks. Specifically, we ran two identical CPU-bound tasks, each repeatedly computing a SHA256 of long strings in a bounded loop. We then measured the time taken for both processes to finish, and repeated this experiment 10 times.

The result shows that processes scheduled with EFS only take 1.25% more time to finish: on average, EFS tasks take 10.83 seconds, and CFS tasks take 10.70 seconds. This small overhead also includes the overhead from the ghOSt framework, which means that the overhead from our energy measurements and scheduling policy is even smaller.

6 LIMITATIONS AND FUTURE WORK

Peripheral devices. Besides CPU and RAM, other devices like disks, NICs and GPUs also consume considerable amounts of energy. However, this energy usage is currently not captured by Wattmeter. For example, when a disk performs an operation, the consumed energy should be attributed to the caller process (currently blocked by the disk operation) instead of the process that is currently running on the CPU. A possible implementation is to instrument the I/O operations initiated by each process with eBPF, and assign energy to each operation. The exact amount of energy to assign can be measured empirically or found in the manufacturer's data sheet for each device, or using specialized interfaces like the SMART API for storage devices.

Distributed applications. We currently focus on scheduling processes on a single node. However, most modern datacenter applications span many nodes across a local (or wide area) network. Attributing energy on a single node is just the first step in attributing energy for large-scale distributed applications. To do the latter, the energy consumption of the network itself, as well as the energy consumption of tasks across many nodes needs to be taken into account, and integrated into a distributed task scheduler such as Kubernetes. The Kepler project [1] has taken some steps in this direction, although it operates at much coarser granularity than Wattmeter.

Minimizing total energy consumption. We implemented two simple policies that equalize energy across processes and limit the energy of any given process. However, one can imagine operators would like to reduce the total energy consumption of tasks (while meeting a certain runtime or performance goal). Such a policy would be more complicated to implement, since it requires the scheduler to also understand the performance constraints and power profile of each task over time, and schedule tasks appropriately. We leave the exploration of such a policy for future work.

7 RELATED WORK

There are existing tools for measuring energy and resource consumption for various settings and applications. On Linux and cloud platforms, there are tools for estimating and controlling process- or container-level energy consumption based on RAPL, hardware monitoring, and other instrumentation mechanisms [1, 7, 12, 14, 25, 28]. There are also specialized tools for applications with substantial energy consumption, in particular, for machine learning workloads [3, 13, 21].

Energy efficiency has been extensively studied in the context of real-time systems. In these systems, process schedulers are tasked with balancing energy efficiency against deadline compliance and system throughput. Existing research indicates that real-time systems equipped with different schedulers can exhibit vastly different energy consumption profiles [4, 23, 24, 26, 27, 29, 30].

Research in energy-aware cloud computing focuses on developing schedulers capable of intelligent consolidation of workloads, cross-machine load balancing, and virtual machine migration. These schedulers enable the dynamic activation and deactivation of machines based on demand, aiming to reduce energy footprint without hurting quality of service [5, 9, 11, 18].

The Linux Energy-Aware Scheduler (EAS) performs energy-aware process placement for CPUs with asymmetric topologies (such as Arm's big.LITTLE). When instantiated with an Energy Model (EM), during a process's wakeup routine, EAS chooses the CPU that is predicted to yield minimal energy consumption without hurting the system's throughput [22].

REFERENCES

[1] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. 2023. Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. 69–71. https://doi.org/10.1109/CLOUD60044.2023.00017

[2] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. 2023. Treehouse: A case for carbon-aware datacenter software. *ACM SIGENERGY Energy Informatics Review* 3, 3 (2023), 64–70.

[3] Lasse F Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv preprint arXiv:2007.03051* (2020).

[4] K. Baynes, C. Collins, E. Fiterman, Brinda Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. 2003. The performance and energy consumption of embedded real-time operating systems. *IEEE Trans. Comput.* 52, 11 (2003), 1454–1469. https://doi.org/10.1109/TC.2003.1244943

[5] Josep Ll Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. 2010. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on energy-Efficient Computing and Networking*. 215–224.

[6] Corbet, Jonathan. 2023. The extensible scheduler class. https://lwn.net/Articles/922405/.

[7] Stefano Corda, Bram Veenboer, and Emma Tolley. 2022. PMT: Power Measurement Toolkit. In *2022 IEEE/ACM International Workshop on HPC User Support Tools (HUST)*. IEEE, 44–47.

[8] Intel Corporation. 2024. Intel 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[9] Hancong Duan, Chao Chen, Geyong Min, and Yu Wu. 2017. Energy-aware scheduling of virtual machines in heterogeneous cloud computing systems. *Future Generation Computer Systems* 74 (2017), 142–150.

[10] eBPF. 2024. extended Berkeley Packet Filter. https://ebpf.io/.

[11] Inigo Goiri, Ferran Julia, Ramon Nou, Josep Ll Berral, Jordi Guitart, and Jordi Torres. 2010. Energy-aware scheduling in virtualized datacenters. In *2010 IEEE International Conference on Cluster Computing*. IEEE, 58–67.

[12] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. 2023. EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*. 1–8.

[13] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research* 21, 248 (2020), 1–43.

[14] hubblo-org. 2023. Scaphandre. https://hubblo-org.github.io/scaphandre-documentation/.

[15] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[16] IEA. 2023. Data Centres and Data Transmission Networks. https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks.

[17] Russ Joseph and Margaret Martonosi. 2001. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design* (Huntington Beach, California, USA) *(ISLPED '01)*. Association for Computing Machinery, New York, NY, USA, 135–140. https://doi.org/10.1145/383082.383119

[18] Fredy Juarez, Jorge Ejarque, and Rosa M Badia. 2018. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems* 78 (2018), 257–271.

[19] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 605–620. https://doi.org/10.1145/3477132.3483548

[20] Wyes Karny. 2023. Enable Core RAPL for AMD. https://lore.kernel.org/lkml/20230217161354.129442-1-wyes.karny@amd.com/T/.

[21] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700* (2019).

[22] Linux. 2024. Energy Aware Scheduling. https://docs.kernel.org/scheduler/sched-energy.html.

[23] Ramesh Mishra, Namrata Rastogi, Dakai Zhu, Daniel Mossé, and Rami Melhem. 2003. Energy aware scheduling for distributed real-time systems. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 9–pp.

[24] Sanjay Moulik, Rishabh Chaudhary, and Zinea Das. 2020. HEARS: A heterogeneous energy-aware real-time scheduler. *Microprocessors and Microsystems* 72 (2020), 102939.

[25] Guillaume Raffin and Denis Trystram. 2024. Dissecting the software-based measurement of CPU energy consumption: a comparative analysis. *arXiv preprint arXiv:2401.15985* (2024).

[26] Abusayeed Saifullah, Sezana Fahmida, Venkata P Modekurthy, Nathan Fisher, and Zhishan Guo. 2020. CPU energy-aware parallel real-time scheduling. *Leibniz international proceedings in informatics* 165 (2020).

[27] Claudio Scordino, Luca Abeni, and Juri Lelli. 2018. Energy-aware real-time scheduling in the linux kernel. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 601–608.

[28] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2013. Power containers: An OS facility for fine-grained power and energy management on multicore servers. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 65–76.

[29] Wei Xiong, Bing Guo, and Shen Yan. 2022. Energy consumption optimization of processor scheduling for real-time embedded systems under the constraints of sequential relationship and reliability. *Alexandria Engineering Journal* 61, 1 (2022), 73–80.

[30] Wanghong Yuan and Klara Nahrstedt. 2003. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 149–163.