Multi-Core Energy-Aware Process Scheduling

Riju Dey[1] rd3054@columbia.edu
Tyler Chang[1] tc3407@columbia.edu

[1]Columbia University, New York, NY, USA

Abstract

Energy consumption increasingly constrains the performance and scalability of modern multicore systems, motivating operating system schedulers that account for energy directly rather than treating it as a secondary concern. Prior work, notably Wattmeter's Energy-Fair Scheduling (EFS) [11], demonstrated that incorporating hardware energy measurements into scheduling decisions can reduce energy usage while preserving fairness, but was primarily evaluated on single-core systems and implemented outside the mainline Linux scheduler.

In this work, we re-implement a variant of EFS using Linux `sched_ext`[6] and eBPF[2], leveraging per-core energy measurements exposed by AMD's RAPL interface. We focus on understanding how energy consumption manifests across different workload classes on modern multicore hardware, rather than proposing a new scheduling policy. Using synthetic CPU-bound and memory-bound workloads, we evaluate how energy usage scales with parallelism, workload intensity, and memory behavior. Our results show that memory-intensive workloads can consume more total energy than CPU-bound workloads despite lower computational intensity, and that high memory pressure may reduce total energy by inducing extended stall periods. These findings highlight the importance of direct energy measurement and suggest that energy-aware scheduling on multicore systems requires careful treatment of both computation and memory behavior.

1 Introduction

Energy efficiency has become a major concern in modern computing systems. As processors continue to scale in core count and heterogeneity, energy consumption increasingly constrains performance, cost, and sustainability across domains ranging from mobile and embedded platforms to large-scale data centers. Traditional operating system schedulers, however, have largely been designed around performance-centric metrics such as fairness, latency, and throughput, treating energy as a secondary or indirect consideration. This motivates a growing body of work on energy-aware scheduling, which seeks to incorporate power and energy signals directly into scheduling decisions in order to better align system behavior with energy efficiency goals.

Prior work has shown that incorporating even limited energy awareness into the operating system scheduler can yield substantial reductions in energy consumption without degrading performance. The Wattmeter paper proposed Energy-Fair Scheduling (EFS), a modification of Linux's Completely Fair Scheduler (CFS) that replaces time-based fairness with energy-based fairness. Rather than accounting for each task's progress using virtual runtime (*vruntime*), as in CFS, EFS scales *vruntime* by the energy consumed while a task is executing, causing energy-hungry tasks to advance more quickly in virtual time and therefore receive proportionally less CPU service. In effect, EFS preserves the fairness structure of CFS while redefining the resource being fairly shared from CPU time to energy. To support this design, Wattmeter relies on hardware energy counters exposed with Intel's Running Average Power Limit (RAPL)[4] interface, which provides fine-grained energy measurements that can be sampled at runtime and attributed to individual tasks. The EFS policy was implemented using ghOSt [8], a user-space kernel bypass framework that allows scheduling decisions to be made outside the kernel while still exerting control over task execution. ghOSt enabled Wattmeter to intercept scheduling events, track per-task energy consumption, and update vruntime in accordance with EFS without modifying core kernel scheduler code.

While Wattmeter provided an important proof of concept, several limitations motivate further exploration. First, modern processors are overwhelmingly multicore, and energy behavior on such systems is significantly more complex. Shared resources, core-level power management, and inter-core interference all complicate the relationship between scheduling decisions and energy consumption. Results derived from single-core systems may not generalize directly to multicore environments, particularly when workloads with diverse resource demands execute concurrently. Second, the ghOSt framework, while powerful, operates outside the mainline Linux scheduler and requires a specialized execution environment. Recent advances such as `sched_ext`[6] offer an alternative path of allowing custom scheduling policies to be implemented using eBPF while integrating directly with the Linux scheduler. This lowers the barrier to experimentation and enables energy-aware scheduling policies to coexist with production kernel infrastructure.

At the same time, hardware support for energy measurement has continued to evolve. Recent generations of AMD processors along with patches to Linux have exposed and enabled per-core energy readings through RAPL[10], enabling more precise attribution of energy consumption at the core level. Unlike earlier platforms where energy measurements were coarse-grained or package-wide, these interfaces make it possible to observe how energy usage varies across cores and over time in response to scheduling decisions. This capability opens the door to studying energy-aware scheduling in realistic multicore settings, where tasks compete for CPU and memory resources and where workload characteristics play a critical role in determining energy efficiency.

Motivated by these developments, this work explores energy behavior in a multicore system using a continuation of the Wattmeter approach, adapted to modern Linux scheduling infrastructure. Rather than proposing a fundamentally new scheduling policy, we re-implement a variant of the EFS policy using `sched_ext` and focus on understanding how energy consumption manifests across different classes of workloads. In particular, we investigate the energy

impact of CPU-intensive versus memory-intensive workloads, leveraging per-core energy measurements to analyze how scheduling and workload behavior interact on multicore hardware.

2 Related Work

Energy-aware scheduling has been studied extensively across operating systems and architecture research. Early work explored dynamic voltage and frequency scaling (DVFS) as a mechanism for reducing energy consumption by adjusting CPU frequency in response to load [13]. Linux integrates several DVFS governors that trade off performance and energy, but these mechanisms operate largely independently of task-level scheduling policy.

Subsequent work has examined energy attribution and accounting in multicore systems, highlighting the challenges posed by shared caches, memory controllers, and uncore components [3, 9]. These studies show that attributing energy to individual tasks is inherently approximate, as energy consumption reflects both local execution and shared resource activity.

More recent work has focused on exposing energy consumption to userspace and analysis tools to better understand application-level energy behavior. Tools such as Scaphandre provide continuous, per-process energy monitoring by leveraging RAPL counters and exporting energy metrics to userspace for profiling and visualization [7]. These systems emphasize observability and accounting rather than direct control over scheduling decisions, but highlight the growing demand for fine-grained, task-level energy insight.

A common characteristic of these energy monitoring and analysis tools is that they operate primarily in userspace, relying on periodic sampling and post hoc analysis of hardware counters. While this design simplifies deployment and instrumentation, it limits the ability to react to energy signals at fine granularity or to influence scheduling decisions in real time. By contrast, `sched_ext` policies implemented in eBPF execute in the scheduler's fast path and can incorporate energy measurements directly into task selection, enabling low-latency, policy-driven responses to energy behavior that are difficult to achieve from userspace alone.

Mainline Linux has also introduced energy-aware scheduler hooks such as the sched_energy interface, which exposes energy accounting and capping support in the kernel scheduler [12]. These interfaces provide a foundation for energy-aware policies in the kernel, but are oriented toward general platform power management and capping rather than the per-task fairness mechanisms explored in this work. While sched_energy provides general energy hooks, it does not directly expose or utilize fine-grained per-task energy readings in the scheduler's fast path in the way `sched_ext` with RAPL does, motivating our combined eBPF and MSR approach.

3 System Design

This project's implementation has two main pieces that work together: (1) a small kernel module that exposes per-core energy readings to BPF as a kfunc[1], and (2) a `sched_ext` scheduler written as an eBPF program that uses those readings to estimate per-task power and enforce energy fairness through an energy-weighted notion of virtual runtime. The scheduler continuously (a) samples the current core's energy counter at context-switch boundaries, (b) turns the energy delta over the elapsed on-core time into a power estimate for the running task, (c) smooths that estimate over time, and (d) uses it at enqueue time to bias ordering so that higher-power tasks effectively advance faster in virtual time than lower-power tasks.

3.1 Per-Core Energy Readings

Because `sched_ext` policies execute in BPF, they cannot directly issue privileged instructions such as reading a model-specific register (MSR). To make per-core energy available inside the scheduling policy, we implemented a kfunc[1] called *read_core_energy* as a kernel module which is then loaded into the kernel and is callable by BPF. The kfunc reads AMD RAPL model-specific registers (MSRs) to obtain a per-core monotonically increasing energy counter for the current core. To interpret the raw counter, the module also reads a power/energy unit MSR during module initialization to extract the energy scaling exponent which, when combined with the energy counter, allows us to obtain metrics in Joules.

3.2 Scheduler Implementation

We ported and adapted the Energy-Fair Scheduling (EFS) policy from the Wattmeter paper into the `sched_ext` framework, keeping the policy's core intuition intact: if two tasks have received comparable service, the one that burns energy faster should be treated as having progressed more and therefore should not continually outrun energy-efficient work. In Wattmeter's formulation[11], that intuition is captured by extending fairness from time to energy using a virtualized metric that resembles CFS's virtual runtime, but scaled by an estimate of task power as shown through the formula:

$$v_{\text{energy}} = v_{\text{runtime}} \cdot v_{\text{power}}$$

EFS would then always pick the task with the lowest v_{energy} usage. Our `sched_ext` implementation follows that same structure: we maintain a global ordering based on a per-task virtual energy metric, and we update each task's estimated power using RAPL energy readings gathered while the task is on-CPU. Using `sched_ext`'s queueing primitives, we maintain a single global ordering over runnable tasks by v_{energy}, closely matching Wattmeter's EFS enqueue/ordering behavior.

When a task begins running, we record the start timestamp and a per-core energy baseline for the CPU it is executing on; when the task stops, we sample the per-core energy counter again and compute $\Delta time$ and $\Delta energy$ for that running interval. Dividing $\Delta energy$ by $\Delta time$ yields an interval power estimate, which we stabilize via a simple smoothing update that blends the new estimate with the prior stored value to reduce sensitivity to short, noisy intervals while still tracking phase changes. In parallel, we update cumulative accounting: we add $\Delta energy$ to the task's attributed consumption and increment global totals for time and energy, which feeds back into the enqueue path as the evolving system-average power baseline used for normalization. On the dispatch side, CPUs draw from the shared globally ordered queue, preserving the v_{energy} rank order as the primary policy mechanism.

3.3 eBPF Maps

To support EFS, we keep a small set of persistent data structures as eBPF maps. Conceptually, these maps serve three purposes: (i)

tracking when a task ran, (ii) tracking what the CPU's energy counter looked like at the start of that run, and (iii) maintaining task-level and system-level aggregates that allow us to normalize and stabilize power estimates.

(1) `pid_to_run_start`: records the timestamp (in nanoseconds) when a task most recently started running. This allows us to compute the duration of each contiguous on-CPU interval.
(2) `cpu_to_prev_energy`: records the last energy reading observed on each CPU at the moment a task began running. Because our energy interface is per-core, this baseline snapshot is naturally indexed by CPU rather than PID.
(3) `pid_to_power`: stores the scheduler's current estimate of each task's power. This value is updated at every stop event using the energy delta accumulated while the task ran.
(4) `pid_to_consumption`: stores each task's cumulative energy consumption over time. While not strictly required to order tasks by *venergy*, it is useful for reporting and analysis at the task granularity.
(5) `total`: a compact global accumulator (a single-entry array) that tracks overall CPU running time and total energy attributed to running tasks. We use it to compute a coarse system average power, which then becomes the normalization baseline for per-task power.

There are several other less relevant eBPF maps that are used to record interesting information such as CPU migration, last enqueue time, etc. We also include some pragmatic handling for special task classes such as having kernel threads being treated separately and routed through a global queue without EFS accounting.

4 Experiments

We performed our experiment on the r6615 machines on Cloudlab [5], which features the AMD EPYC 9004 Series cores, which support reading per-core energy readings via the RAPL MSR registers. [We also experimented with the c6620 machines on Cloudlab, to experiment with Intel's versions of the RAPL interface]. Each processor possesses a different model of how energy is counted among the CPUs

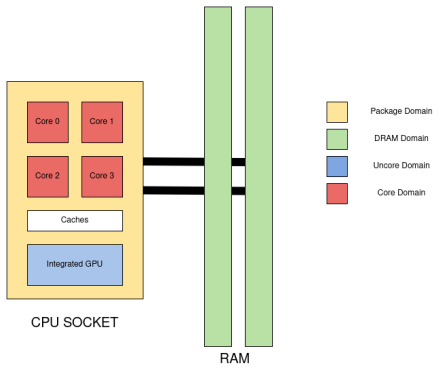

Figure 1: Block diagram representation of the different RAPL domains. AMD supports reading energy from each individual core, while Intel does not.

4.1 CPU vs. Non-CPU Readings

One aspect of Wattmeter that we sought to improve upon was to profile energy usage more granularly across the CPU package. That is, would we be able to make precise measurements of energy across cache usage, the DRAM or the integrated GPU? To test this, we equipped our kfunc to read from the appropriate MSRs necessary to read from the PP0, PP1 and other domains specified in the RAPL interface and measured their values over various workloads.

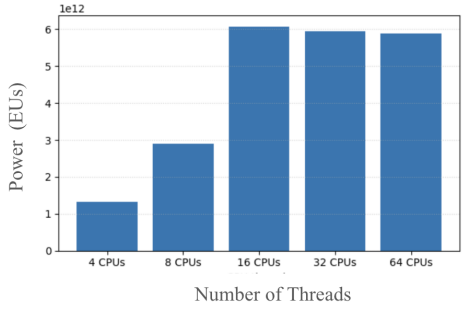

Figure 2: Overall energy consumption of `stress_core` over 20 seconds as a function of number of spawned threads.

4.2 Workloads

We primarily tested our `sched_ext` scheduler using two programs that targeted different components of the CPU. The two programs are defined as follows:

`stress_core` is a CPU-bound benchmark that keeps a single core fully occupied with arithmetic operations, producing sustained high core activity with minimal memory traffic, making it suitable for evaluating core-level energy behavior under `sched_ext`.

`mem_miss` is a memory-bound benchmark that repeatedly accesses a large memory region with a cache-unfriendly access pattern, producing a high rate of cache misses and sustained DRAM traffic. Its parameters can be varied to control the working set size and access behavior, making it suitable for evaluating how `sched_ext` responds to workloads whose energy consumption is dominated by the memory subsystem rather than core execution.

Figure 3 presents the cumulative energy consumption of four workloads measured over a 60-second execution window using our Energy-Fair Scheduler. Energy usage was sampled every 100 ms by polling the `pid_to_consumption` eBPF map, which accumulates per-task energy attributed from per-core RAPL readings. The experiment compares `stress_core` against three memory-intensive configurations of `mem_miss`, each operating on a fixed 512 MB allocation but with increasing working-set sizes and write intensities. The low-intensity configuration accesses a small working set with infrequent writes, the medium configuration increases both the working set and write activity, and the high-intensity configuration maximizes memory pressure by operating on the full allocation

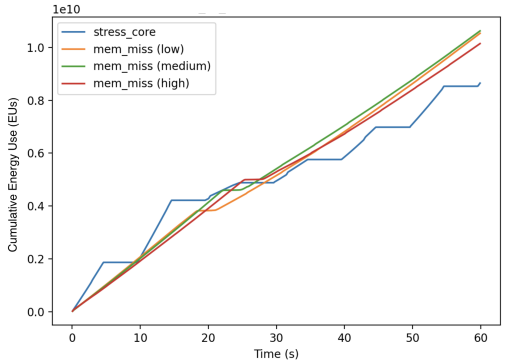

Figure 3: Energy usage of CPU and memory intensive workloads over 60 seconds.

with a high write percentage. These configurations allow us to isolate how progressively stronger memory behavior impacts energy consumption under the same scheduling policy. Across the full duration, all memory-intensive workloads exhibit steeper cumulative energy growth than the CPU-intensive workload, indicating that memory-dominated execution leads to higher overall energy usage despite lower computational intensity.

Interestingly, the most aggressive memory-intensive configuration consumes less total energy than the low- and medium-intensity variants, despite generating more memory pressure. We believe this result can be explained by the interaction between memory stalls and core power draw. At high memory intensity for the given time frame, the core may have spent a significant fraction of time stalled waiting on memory, reducing effective instruction throughput and lowering dynamic core power consumption. In contrast, the low- and medium-intensity configurations maintain enough locality to keep the core more actively engaged while still incurring frequent memory accesses, leading to higher sustained power draw over time. This is not to say that higher memory intensity will always consume less energy in the long run compared to the low- and medium-intensity variants, as shown in Figure 4, but rather, it is a notable observation to be aware of from this experiment.

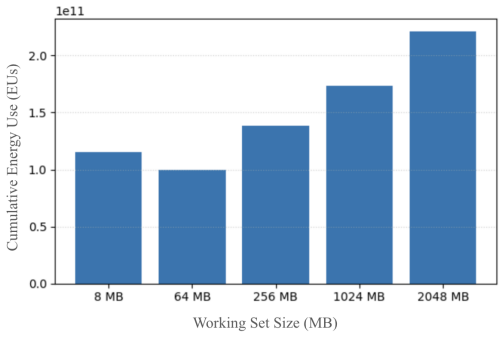

Figure 4: Cumulative energy Usage of `mem_miss` over different working set sizes.

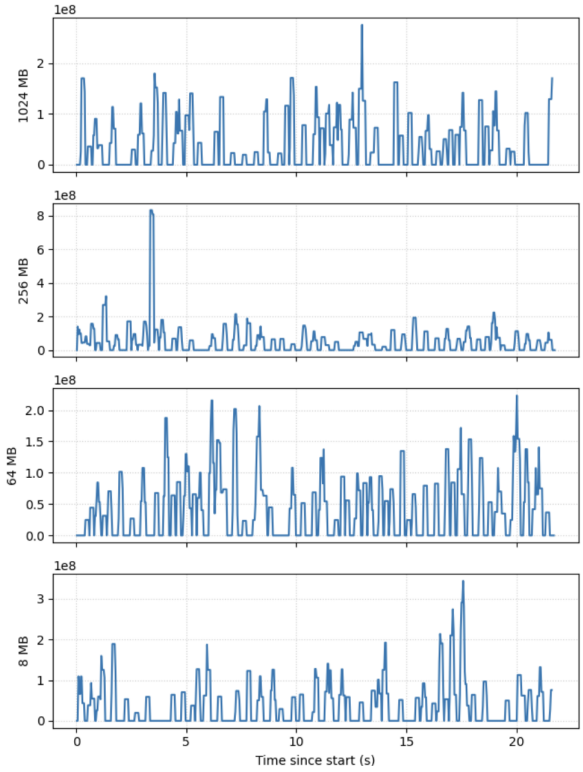

Figure 5: Power behavior over time of `mem_miss` over different working set sizes.

4.3 Concurrency & Parallelism

One of the features of Wattmeter that we wished to improve on was measuring how system energy usage was affected when running multiple programs on multiple cores. To simulate this, we launched varying numbers of instances of both `stress_core` and `mem_miss`.

5 Results and Evaluation

We were able to successfully run our workloads and observe some effect on the overall energy profile of the system. As shown in Figures 2 increasing the number of spawned threads of a CPU-intensive workload almost linearly increased the total energy spent over time. Figure 2 also displays the power-capping feature of the EFS, which caps the energy usage to the performance of an 8-core machine, even though the hardware features 64 cores. Figure 4 displays a very similar result, with `mem_miss` consuming more energy with higher working sets. With regards to actual power behavior, we found that `stress_core` was exactly as expected, consuming energy at extremely steady and constant rates, while `mem_miss` exhibited large fluctuations corresponding to bursts of memory activity and stall-heavy execution. These bursts align with periods of elevated energy consumption followed by extended low-power intervals, reflecting the dominance of memory stalls over sustained core execution. This contrast further illustrates that steady, compute-heavy workloads lead to predictable and linearly scaling energy usage,

whereas memory-bound workloads produce more complex energy profiles that depend strongly on access patterns and locality.

5.1 Core vs Uncore Energy

As mentioned earlier, we attempted to attribute certain amounts of energy usage to different parts of the processor using the different RAPL domains. Unfortunately, however, we found that for most of the Intel processors that we tested on Cloudlab, the RAPL MSRs were not very well-implemented - that is, we would retrieve reasonable values for the package energy domain and the core energy domain, but essentially useless values for the DRAM or "uncore" domain, which contains the integrated GPU or other components. This is most likely due to the fact that the RAPL interface is supported to varying degrees across processors, and we found that using the per-core energy readings of AMDs RAPL were our best bet. We also attempted a naive calculation to separate out the energy used by cache or memory controllers by subtracting the full package power from the sum of the CPU cores, implemented as the following equation:

$$\text{uncore\_energy} = \text{package\_energy} - \sum_{i=0}^{\text{num\_cpus}}$$

Unfortunately, however, we observed that this value rose at a constant rate, even when the power of the CPUs was varying over time. In other words, no matter the variance in program execution, the amount of energy consumed by the "uncore" was constant and hence yielded no useful information for our scheduler or in determining where energy was being spent.

To try and cause some on the uncore energy, we used varying intensities of the `mem_miss` workload, but found negligible variations in energy.

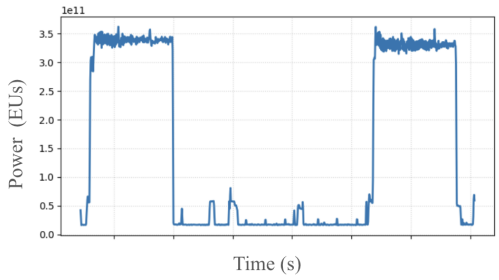

Figure 6: Power behavior of a simple sequential workload, running `stress_core`, `mem_miss` and `stress_core` in order.

To compare the energy usage of `mem_miss` and `stress_core` side by side, we created a sequential "mixed" workload which ran several instances of `stress_core`, followed by several instances of `mem_miss` and so on. One such run is shown in Figure 6. Consistent with our previous isolated runs, we observe that `stress_core` consumes significantly more energy than `mem_miss`, which shows up infrequently in small bursts.

6 Limitations and Future Work

While this project demonstrates the feasibility of implementing an energy-aware scheduler on a multi-core system using `sched_ext` and per-core energy measurements, there are several important limitations that suggest directions for future work.

First, although we leverage per-core RAPL energy counters, the attribution of energy to individual tasks remains approximate. Energy readings are sampled at context-switch boundaries and attributed to the task that was running during the interval, which implicitly assumes that energy consumption during that interval is dominated by the executing task. In practice, energy usage is also influenced by shared microarchitectural components such as caches, memory controllers, and interconnects, as well as by background kernel activity. Future work could incorporate additional hardware counters or statistical attribution techniques to better separate task-level energy consumption from shared system effects.

An additional practical limitation arises from the nature of the RAPL energy interface itself. RAPL exposes energy consumption as a monotonically increasing hardware counter whose units are defined by a model-specific energy scaling factor read from a separate MSR. Converting raw counter values into Joules therefore relies on a fixed conversion factor that is assumed to be accurate and stable across operating conditions. In reality, this factor is derived from hardware characterization and may not perfectly reflect instantaneous energy usage across voltage, frequency, temperature, or workload variations. As a result, while RAPL provides a useful and widely adopted approximation of energy consumption, the measurements it exposes should be interpreted as estimates rather than ground truth. Future work could explore calibration techniques, cross-validation with external power measurements, or hybrid approaches that combine RAPL with other performance counters to improve robustness.

Second, the power estimation model used by our scheduler is intentionally simple. We compute instantaneous power from energy deltas over execution intervals and apply lightweight smoothing to reduce noise. While sufficient for capturing coarse trends, this model may lag rapid phase changes or misestimate power for bursty workloads. More sophisticated filtering, phase detection, or predictive models could improve responsiveness while preserving stability.

Third, our evaluation focuses primarily on synthetic workloads that isolate CPU-bound and memory-bound behavior. While this helps clarify how different resource profiles affect energy consumption, real-world applications often exhibit mixed and time-varying behavior. Evaluating the scheduler using more complex workloads, such as databases, web servers, or multi-stage pipelines, would provide deeper insight into its effectiveness under realistic conditions.

Finally, `sched_ext` provides significant flexibility but also imposes practical constraints. BPF programs must obey verifier restrictions, operate within tight instruction limits, and rely on a limited set of kernel-visible primitives. Exploring which aspects of energy-aware scheduling are best implemented inside `sched_ext` versus in supporting kernel modules remains an open design question.

Overall, this work serves as a step toward practical energy-aware scheduling on modern multicore systems. By combining `sched_ext` with increasingly fine-grained hardware energy measurements, future systems can explore richer scheduling policies that balance performance, fairness, and energy efficiency in a more holistic manner.

7 Code

The source code for this project can be found at https://github.com/tylerchang/energy-scheduler/.

Acknowledgments

To Professor Cidon and Tal Zussman for advising us on this project.

References

[1] 2025. BPF Kernel Functions (kfuncs). The Linux Kernel Documentation. https://www.kernel.org/doc/html/next/bpf/kfuncs.html.

[2] 2025. eBPF.io: Introduction, Tutorials & Community Resources. eBPF.io Website.

[3] Frank Bellosa. 2000. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. In *Proceedings of the ACM SIGOPS European Workshop*.

[4] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*.

[5] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[6] Daniel Hodges. 2024. Scheduling at Scale: eBPF Schedulers with Sched$_{ext}$. *USENIX Association, Dublin*.

[7] Hubblo. 2020. Scaphandre: Energy consumption monitoring. https://github.com/hubblo-org/scaphandre.

[8] 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. https://cs.stanford.edu/~jhumphri/documents/ghost.pdf

[9] David Lo, Liqun Cheng, Madanlal Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2014. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[10] LWN.net. 2024. Add per-core RAPL energy counter support for AMD CPUs. LWN.net Article 981655. https://lwn.net/Articles/981655/.

[11] Feitong Qiao, Yiming Fang, and Asaf Cidon. 2024. Wattmeter: Energy-Aware Process Scheduling On Linux. https://hotcarbon.org/assets/2024/pdf/hotcarbon24-final29.pdf

[12] The Linux Kernel Documentation. 2025. Energy Accounting and Capping (sched_energy). https://docs.kernel.org/scheduler/sched-energy.html.

[13] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. 1994. Scheduling for Reduced CPU Energy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.