Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

SPECIALIZATION PROJECT 2

Wholistic container-level energy consumption estimation

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
July 31, 2025

Study program:
Computer Science, M.Sc.

Imprint

Project: Specialization Project 2
Title: Wholistic container-level energy consumption estimation
Author: Caspar Wackerle
Date: July 31, 2025
Keywords: energy efficiency, cloud, kubernetes

Study program:
Computer Science, M.Sc.
Zurich University of Applied Sciences

Supervisor 1:
Prof. Dr. Thomas Bohnert
Zurich University of Applied Sciences
Email: thomas.michael.bohnert@zhaw.ch
Web: Link

Supervisor 2:
Christof Marti
Zurich University of Applied Sciences
Email: christof.marti@zhaw.ch
Web: Link

Abstract

Abstract

The accompanying source code for this thesis, including all deployment and automation scripts, is available in the **PowerStack**[1] repository on GitHub.

TODO: write abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Context

1.1 Introduction and Context

1.1.1 Cloud Computing and its impact on the global energy challenge

Global energy consumption is rising at an alarming pace, driven in part by the accelerating digital transformation of society. A significant share of this growth comes from data centers, which form the physical backbone of cloud computing. While the cloud offers substantial efficiency gains through resource sharing and dynamic scaling, its aggregate energy footprint is growing rapidly. While data center accounted for around 1.5% (around 415 TWh) of the worlds electricity consumption in 2024, they are set to more than double by 2030[2]. That is slightly more than Japans's current electricity consumption today.

This increase is fueled by the rising demand for compute-heavy workloads such as artificial intelligence, large-scale data processing, and real-time services. Meanwhile, traditional drivers of efficiency—such as Moore's law and Dennard scaling—are slowing down[3, 4]. Improvements in data center infrastructure, like cooling and power delivery, have helped reduce energy intensity per operation[5], but these gains are approaching diminishing returns. As a result, total data center energy use is expected to grow faster than before, even as efficiency per unit of compute continues to improve more slowly[6].

1.1.2 Rise of the Container

Containers have become a core abstraction in modern computing, enabling lightweight, fast, and scalable deployment of applications. Compared to virtual machines, containers impose less overhead, start faster, and support finer-grained resource control. As such, they are widely used in microservice architectures and cloud-native environments[7].

This trend is amplified by the growing popularity of Container-as-a-Service (CaaS) platforms, where containerized workloads are scheduled and managed at high density on shared infrastructure. Kubernetes has become the de facto orchestration tool for managing such workloads at scale. While containers are inherently more energy-efficient than virtual machines in many scenarios[8], their widespread use presents a new challenge: understanding and attributing their energy consumption accurately.

1.1.3 Container Energy Consumption Measurement Challenges

Knowing the energy consumed by a container on a server is the essential elemenent to a container-level energy efficiency assessment of both the container itself, as well as the environment surrounding it. An accurate energy consumption estimation is therefore required to validate and improve any potential energy efficiency improvements of a container enviromnent, from kubernetes system components (e.g. Kubernetes Schedulers) to the containers themselves.

Energy consumption in containerized systems is inherently hard to measure due to the abstraction layers involved. Tools like RAPL (Running Average Power Limit) expose component-level energy metrics on modern Intel and AMD CPUs, but this information is not accessible from within containers or virtual machines. In public cloud environments, such telemetry is either not exposed or aggregated at coarse granularity, making direct measurement infeasible.

Containers further complicate attribution: because they share the kernel and hardware resources, it is difficult to isolate the energy impact of one container from another. Only indirect metrics—such as CPU time, memory usage, or performance counters—are available, and even these may be incomplete or noisy depending on system configuration and workload behavior. Various tools exist that attempt to model container power usage based on these inputs, but rarely are their produced metrics transistent and verified.

1.1.4 Problem Definition

The growing importance of containers in cloud environments, combined with the difficulty of directly measuring their energy usage, motivates this work. In particular, this thesis investigates the questions:

Question 1: Which measuremnt methods, metrics or models allow for reliable container-level power estimation?

Question 2: How should a software-based container energy consumption estimation tool be implemented?

Question 3: How can existing container energy consumption estimation tools be validated?

To answer these questions, this study explores methods of measuring server energy consumption, analyzes container workload metrics, and evaluates modeling techniques that aim to bridge the gap between raw energy data and container-level attribution. **The focus is on bare-metal Kubernetes environments, where full system observability allows for deeper analysis and model validation, serving as a foundation for future energy-aware cloud architectures.**

1.1.5 Context of this thesis

This thesis is part of the Master's program in Computer Science at the Zurich University of Applied Sciences (ZHAW) and represents the second of two specialization projects ("VTs"). The preceding project (VT1) focused on the practical implementation of a test environment for energy efficiency research in Kubernetes clusters. This

thesis (VT2) is meant to explore theoretical and methodolocical aspects of container energy consumption measurements in detail.

Furhtermore, this thesis builds upon prior works focused on performance optimization and energy measurement. EVA1 covered topics such as operating system tools, statistics, and eBPF, while EVA2 explored energy measurement in computer systems, covering hardware, firmware, and software aspects. These foundational topics provide the basis for the current thesis but will not be revisited in detail.

1.1.6 Use of AI Tools

During the writing of this thesis, *ChatGPT*[9] (Version 4o, OpenAI, 2025) was used as an auxiliary tool to enhance efficiency in documentation and technical writing. Specifically, it assisted in:

- Structuring and improving documentation clarity.
- Beautifying and formatting smaller code snippets.
- Assisting in LaTeX syntax corrections and debugging.

All AI-generated content was critically reviewed, edited, and adapted to fit the specific context of this thesis. **ChatGPT was not used for literature research, conceptual development, methodology design, or analytical reasoning.** The core ideas, analysis, and implementation details were developed independently.

1.1.7 Project Repository

All code, configurations, and automation scripts developed for this thesis are publicly available in the PowerStack[1] repository on GitHub. The repository contains Ansible playbooks for automated deployment, Kubernetes configurations, monitoring stack setups, and benchmarking scripts. This allows for full reproducibility of the test environment and facilitates further research or adaptation for similar projects.

Chapter 2

State of the Art and Related Research

2.1 Energy consumption measurement and efficiency on data center level

Energy consumption and efficiency on a data center level has been well-studied to the point where various Literature reviews were published[10, 11]. The bigger part of this research is focused on the data center infrasturture (cooling and power), and with good reason, as the data center infrastructure is responsible for a large part of the energy consumption. While a large number of coarse-, medium- and fine-grained metrics for data center energy consumption exist, most data center operators have focused on improving coarse-grained mertrics (especially the *Power Utilization Effectiveness, PUE*) with improvements to infrastructure. This has resulted in a PUE of 1.1 or lower in some cases[5]. Meanwhile, server energy efficiency has substantially improved, especially for partial load and idle power[12]. This has allowed data center operators to improve energy efficiency by simply installing more efficient cooling and power systems and servers. Fine-grained metrics such as server component utilization rates or speed were generally not used in the context of energy efficiency, but rather as performance metrics to ensure customer satisfaction.

2.2 Energy consumption measurement on a server level

As a result of the energy efficiency improvements of both data center infrastructure and server hardware mentioned in the previous section, a shift has started towards evaluating the actual server load energy efficiency. Efficiency gains on this level compound into further gains at the data center level. The method of resource-sharing of modern cloud computing (and especially the use of containers) have created great opportunities for server workload optimitation for energy efficiency, which in turn require power consumption measurements for evaluation. In the context of containers on multi-core processors, measuring the energy consumption of the entire server is insufficient, since it does not allow the attribution of consumped energy to specific containers or processes. While component-level power measurements provide finer measurements that could theoretically be modelled to display container energy consumption, they drastically raise the complexity for a number of reasons:

- Component-level energy consumption measurement without external tools is far from easy. While some components provide estimation models (e.g. Intel RAPL or *Nvidia Management Library* (NVML)), others can only be estimated using their performance metrics. This will invariably lead to large measurement uncertainties, especially with the component hardware differences between generations and manufacturers.

- The problem of attributing measured or estimated energy consumption to individual containers is in itself non-trivial: It not only requires a fine-grained time synchronization of energy consumption and used container resources due to the fast-switching nature most server components during any sort of multitasking.

- A deep understanding of dynamic or static energy consumption is required: Depending on the energy consumption attribution model, a container might not only account the energy it actively used, but potentially also account for a fraction of the energy consumed for any shared overhead such as shared hardware components, or system resources (such as the Kubernetes system architecure). This idea can be further extended: containers could potentially be penalized for any unused server resources, as these unused capacity still consume energy. These different attribution models lead to a larger debate about the goals of the measurements.

- Any server-level power models used to estimate the relation of individual component energy consumption suffers from the varity of different server configurations due to server specialization, such as Storage-, GPU-, or Memory-optimized servers.

In a systematic review cloud servers power models, Lin et al[13] categorize power collection methods into 4 categories:

Key	**Value**	**Description**	**Deployment Difficulty**	**Data Granularity**	**Data Credibility**
Based on instruments	Installation of extra devices	Bare-metal machines	Easy	Machine Level	Very high
Based on dedicated aquisition system	Specialized systems	Specified models of machines	Difficult	Machine or component-level	High
Based on software monitoring	Build-in power models	Bare-metal and virtual servers	Moderate	Machine, component, or VM level	Fair
Based on simulation	System simulation	Machine, component or VM level	Easy	Machine, component, or VM level	Low

TABLE 2.1: Comparison of power collection methods for cloud servers

The following sections of this chapter aim to present the current state-of-the-art in the various fields of research of the problem domains listed above, focussing on

different measurement approches: Direct hardware measurements, In-band measurement techniques and model-based estimation. The following sections are organized by measurement approach, foregoing organization by server component. For this reason, section 2.7 provides a brief summary of component-specific energy consumption measurement techniques.

2.3 Direct Hardware Measurement

2.3.1 Instrument-based power data aquisition

Instrument-based Data collection aquisition produces the hightest data credibility at a low granularity: These devices, installed externally (measuring the power supplied to the PDU) or internally (measuring the power flow between the PDU and motherboard) have been the source of information for a number of studies. The approach to simply measure electric power at convenient hardware locations using dedicated equipment can of course be extended to provide additional granularity: For example, Desrocher et al[14] custom-created a DIMM extender custom-fitted with Hall-sensor resistors and a linux measurement utility to measure power consumed by a DIMM memory module at 1kHz sampling rate using a *WattsAppPro?* power meter and a *Measurment Computing USB.1208FS-Plus* data aquisition board.

This of course highlights a fundamental truth of instrument-based data collection: While it is possible to implement a measuring solution that provides high-granular and high-sampling rate power data, it is paired with an immense effort since solutions like this are not provided off-the-shelf. Unsuprisingly, this is most valuable for benchmarking or validation (Desrochers et al used their setup to validate Intel RAPL DRAM power estimations on three different systems). However, this methodology is (currently) unsuitable for deployment to data center servers due to its bad scalability and prohibitive costs. Hence, the primary role of instrument-based power data aquisiton is as a benchmarking and validation tool for research and development.

2.3.2 Dedicated Aquisition systems

2.3.2.1 BMC Devices, IPMI and Redfish

Some manufacturers have developed specialized power data aquisition systems for their own server products. The baseboard management controller (BMC) is a typical dedicated aquisition system usually integrated with the motherboard, usually as part of the intelligent platform management interface (IPMI)[13]. It can be connected to the system bus, sensors and a number of components to provide power and temparature information about the CPU, memory, LAN port, fan, and the BMC itself. Some comprehensive management systems such as Dell iDRAC or Lenovo xClarity have been further developed to provide high-quality, fine-grained power data due to their close interoperation between system software and underlying hardware. BMC devices on modern servers often offer IPMI- or Redfish interfaces. While these interfaces use the same physical servers, their implementation differ significantly, where Redfish generally offers higher accuracy (e.g through the use of higher-bit formats, whereas IPMI often uses 8-bit raw numbers).

In the context of container power consumption estimation, IPMI-implementations occupy an interesting role. In 2016, Kavanagh et al[15] found the accuracy of IMPI power data to be relatively inaccurate when compared with an external power meter,

mainly due to the large measurement window size of 120 to 180 seconds and the inaccurate assessment of the idle power. They concluded that IMPI power data was still useful when a longer averaging window was used, and the initial datapoints discounted. In a later study, they suggest combining the measurements of IPMI and Intel RAPL (which they find to underestimate the power consumption) for a reasonable approximation of true measurement[16]. Kavanagh's findings have been cited in various studies, often to negate the use of IPMI for power measurement. When used, it sometimes is chosed because it was the "simplest power metric to read"[17] in the context of entire data centers.

Redfish is a modern Out-of-band Management System, first released in 2015 explicitely to replace IPMI [18]. It uses a RESTful API and JSON data format, making queries with code easier. In 2019, Wang et al[19] directly compared IPMI and redfish power data to a reading of a high accuracy power analyzer, and found Redfish to be more accurate than IPMI, with a MAPE of 2.9%, while also finding a measurement latency of about 200ms. They also found measurements to be more accurate in higher power ranges, which they attribute to the improved latency.

In conclusion, BMC power data aquired over Redfish provides a simple simple and comparatively easy way to measure system power based on various physical system sensors. Its biggest strenght lies in easy implementation and general availability. In the context of container energy consumption, BMC power data lacks the short sampling rates necessary to measure a a highly dynamcic container setup, but can prove useful as a validation or cross-reference dataset for longer intervals exceeding 120 seconds. Unfortunately, the data quality of BMC power data depends on the actual system, and power models can be significantly improved by initial calibration with an external power measurement device[15].

2.4 In-Band Measurement Techniques

In-band measurement techniques refer to methods of power consumption monitoring that utilize built-in telemetry capabilities of system components to collect energy usage data directly from within the host system. Unlike external power meters or BMCs like IPMI, which operate independently of the main system, in-band techniques leverage on-die sensors and software interfaces to gather power metrics in real-time. These techniques provide fine-grained data with minimal additional hardware, making them well-suited for scalable environments like Kubernetes clusters. However, their accuracy and granularity are often dependent on the hardware's internal estimation algorithms, which may introduce uncertainties compared to direct measurement methods.

2.4.1 ACPI

The *Advanced Configuration and Power Interface (ACPI)* is a standardized interface that facilitates power management and hardware configuration by allowing the operating system to control hardware states such as processor sleep, throttling, and performance modes [20]. It plays a significant role in processor performance tuning by exposing C-states (idle), P-states (performance), and T-states (throttling) which the OS can leverage to adjust the processor's activity, frequency, and voltage.

Although ACPI defines these power states, their actual implementation is processor-specific, and the interface does not provide real-time telemetry. As such, ACPI does not expose instantaneous power consumption values. Any attempt to estimate power based on ACPI would require detailed knowledge of processor-specific behavior, including the mapping between frequency, voltage, and power—information that is not exposed through ACPI. As a result, limited research was conducted on this topic.

In theory, one could attempt to use ACPI's `_PSS` (Performance Supported States) table, which lists available P-states along with nominal voltage, frequency, and optionally estimated maximum power dissipation, to perform rough CPU power estimation. This method would involve tracking CPU residency in each performance state and applying simple integration models to estimate total energy. However, due to the static nature of `_PSS` entries and the lack of temporal precision, such estimates would be inherently coarse-grained and typically inaccurate for modern processors with dynamic voltage and frequency scaling or turbo modes.

Consequently, ACPI is rarely used in contemporary power estimation contexts. Its primary role remains in system configuration and power state control rather than accurate energy quantification. In modern Intel processors, the Running Average Power Limit (RAPL) interface provides a more appropriate solution for in-band power measurement. This makes RAPL the preferred tool for energy-aware computing research and production environments alike.

2.4.2 Intel RAPL

Intel Running Average Power Level (RAPL) is a Power Monitoring Counter (PMC)-based feature introduced by Intel and provides a way to monitor and control the energy consumption of various components within their processor package[21]. An adaptation of RAPL for AMD processors uses largely the same mechansms and the same interface[22], although it provides less information than Intel's RAPL, providing no DRAM energy consumption[23]. Unfortunately RAPL does not have a detailed low-level implementation documentation, and the exact methodology of the RAPL calculations remain unknown[24].

Intel RAPL has been used extensively in research to measure energy consumption[25] despite some objections about its accuracy, which will be discussed sections 2.4.2.2 and 2.4.2.3. The general concencus is that RAPL is *good enough* for most scientific work in the field of server energy consumption and efficiency. As Raffin et at[26] point out, it is mostly used *like a black box without deep knowledge of its behavior*, resulting in implementation mistakes. For this reason, the next section 2.4.2.1 presents an overview of the RAPL fundamentals.Finally, section **??** discusses the currently available RAPL-based tools.

2.4.2.1 RAPL measurement methods

This subsection provides an overview of how RAPL works and is used. It is based on the Intel rchitectures Software Developer's Manual[27, Section 16.10] and the works of Raffin et al [26] (2024) and Schöne et al [28] (2024).

Running Average Power Limit (RAPL) is a power management interface in Intel CPUs. Apart from power limiting and thermal management, it also allows to measure the energy consumed by various components (or *domains*). These domains individual CPU cores, integrated graphics (in non-server CPUs) and DRAM, as well as *package*, refering to the whole CPU die. While it initially used models to estimate energy use[29], it now uses physical measurements. The processor is divided into different power domains or "planes", representing specific components, seen in figure 2.1. Notably, not all domains are present in all systems: Both client-grade systems feature the *Package* and *PP0 core* domains, server grade processors typically don't show the *PP1 uncore*-domain typically used for integrated GPUs, and client-grade processors don't show the *DRAM* domain. The *PSYS* domain for the "whole machine" is ill defined and only exists on client-grade systems. In an experiment with recent Lenovo and Alienware laptops, Raffin et al found that the *PSYS* domain reported the total consumption of the laptop, including display, dedicated GPU and other domains. Regardless, this thesis will focus on the RAPL power domains available to server-grade processors.

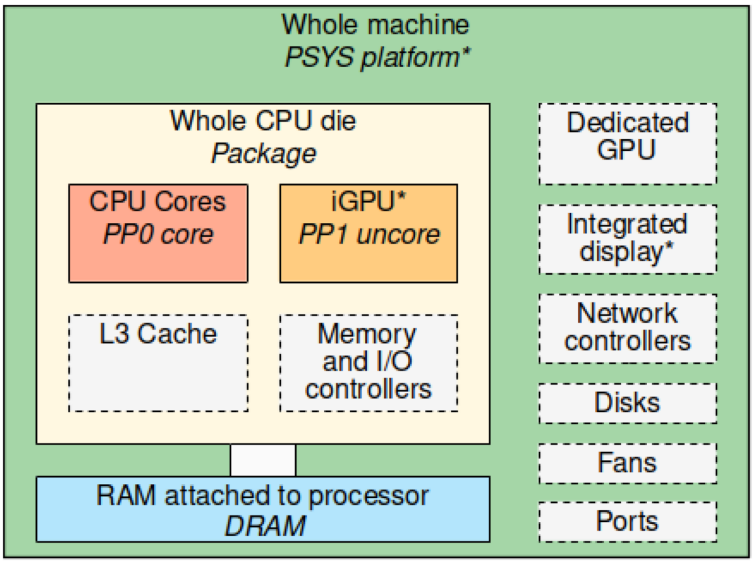


FIGURE 2.1: Hierarchy of possible RAPL domains and their corresponding hardware components. Domain names are in italic, and grayed items do not form a domain on their own, items with an asterisk are not present on servers[26].

RAPL provides hardware counters to read the energy consumption (and set power limits) for each domain. The energy consumption is measured in terms of processor-specific "energy units" (e.g. 61μJ for Haswell and Skylake processors). The counters are exposed to the operating system through model-specific registers (MSRs) and are updated approximately every millisecond. The main advantages of RAPL are that no external powermeters are required, nor a privileged access to the BMC (which could be used to power off the server). RAPL is more accurate than any untuned statistical estimation model.

Various measurement methods can be used to extract RAPL measurements. In a detailed comparison, Raffin et al[26] outline their individual features and tradeoffs, which are summarize in figure 2.2b:

- **Lacking documentation:** Since there is no publicly available documentation

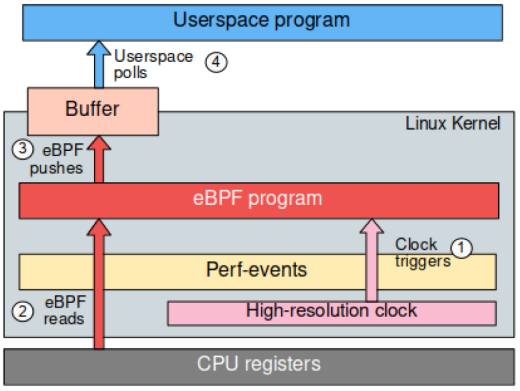
of the low-level RAPL implementation, implementations are bound to suffer inaccuracies and inconsistencies due to a lack of understanding.

- The **Model-Specific Register (MSR)** interface provides low-level access to RAPL energy counters but is complex and hardware-dependent. Developers must manually determine register offsets and unit conversions based on processor model and vendor documentation. This method lacks safeguards, requires deep processor knowledge, and is error-prone, with incorrect readings difficult to detect. Although read-only access poses no risk to system stability, MSRs expose sensitive data and are thus restricted to privileged users (e.g., `root` or `CAP_SYS_RAWIO`). Fine-grained access control is not supported natively, though the `msr-safe` module offers limited mitigation.

- The **Power Capping (powercap)** framework is a high-level Linux kernel interface that exposes RAPL energy data through the sysfs filesystem, making it accessible from userspace. It simplifies energy measurements by automatically handling unit conversions and domain discovery, requiring minimal hardware knowledge. Though domain hierarchy can be confusing (especially with DRAM domains appearing nested under the package domain) powercap remains user-friendly and scriptable. It supports fine-grained access control via file permissions and offers good adaptability to hardware changes, provided the measurement tool doesn't rely on hard-coded domain structures.

- The **perf-events** subsystem provides a higher-level Linux interface for accessing RAPL energy counters as counting events. It supports overflow correction and requires less hardware-specific knowledge than MSR. Each RAPL domain must be opened per CPU socket using `perf_event_open`, and values are polled from userspace. While it lacks a hierarchical structure like powercap and may be harder to use in certain languages or scripts, it remains adaptable and robust across different architectures. Fine-grained access control is possible via kernel capabilities or `perf_event_paranoid` settings.

- **eBPF** enables running custom programs in the Linux kernel, and in this context, it is used to directly read RAPL energy counters from within kernel space, potentially reducing measurement overhead by avoiding user-kernel context switches. The implementation attaches an eBPF program to a CPU clock event, using `perf_event_open` to access energy counters and buffering results for userspace polling (is visualized in figure 2.2a).While offering the same overflow protection as regular `perf-events`, this approach is significantly more complex, prone to low-level errors (especially in C), and requires elevated privileges (`CAP_BPF` or `root`). It also lacks portability, as it demands manual adaptation to kernel features and domain counts, limiting its maintainability across systems.



(A) RAPL perf-event eBPF mechanism

mechanism	technical difficulty	required knowledge	safeguards	privileges	resiliency
MSR	medium	CPU knowledge	none	SYS_RAWIO cap. + msr module	poor
perf-events + eBPF	high (long, complicated code)	limited	overflows unlikely, many other possible mistakes	PERFMON and BPF capabilities	manual tweaks necessary for adaptation
perf-events	low	limited	good, overflows unlikely	PERFMON capability	good
powercap	low	limited	beware of overflows	read access to one dir	good, very flexible

(B) RAPL measurement mechanisms comparison

FIGURE 2.2: RAPL measurements: eBPF and comparison[26]

In their research, Raffin et al conclude that all four mechanisms have small or negligible impact on the running time of their benchmarks. They formulate the following recommendations for future energy monitoring implementations:

- Measuring frequencies should be adapted to the state of the node, preventing high measurement overhead, due to a reduction in time spent in low-power states. Under heavy load, a highfrequencay can be used in order to capture more information.
- `perf-events` is the overall recommeded measurement method with good efficiency, latency and overflow protection. Powercap is less efficient, but provides a simpler sysfs API.
- Even though `perf-events` and eBPF-measurement method seems to be the most energy-efficient, it is not recommended in light of its complexity. For the same reason, the MSR method is not recommeded, as it raises complexity while counter-intuitively being slower than `perf-events`

RAPL MSRs can be read on some cloud computing resources (e.g. some Amazon EC2-instances), although the hypervisor traps the MSR reads, which can add to the polling delay. in EC2, the performance overhead also significantly increases to <2.5% (as compared to <1% on standalone systems)[24].

2.4.2.2 RAPL Validation

Since its inception, RAPL has been subject of various validation studies, with the general concensus that it's accuracy could be considered "good enough"[26]. Notable works are Hackenberg et al, that in 2013 found RAPL accurate but missing timestamps[30], and in 2015 noticed a major improvement to RAPL accuracy, after Intel switched from a modeling approach to actual measurements for their Haswell architecture[29]. Desrochers et al concluded in a 2016 RAPL DRAM validation study[14] that DRAM power measurement was reasonably accurate, especially on server-grade CPUs. They also found measurement quality to drop when measuring and idling system. Later, Alt et al[31] tested DRAM accuracy of heterogeneous memory systems of the more recent Ice Lake-SP architecture and concluded that DRAM estimates behaved differently than on older architectures. They noted that the RAPL

overestimates DRAM energy consumption by a constant offset, which they attribute to the off-DIMM voltage regulators of the memory system.

A critical point in the RAPL validation was the introduction of the Alder Lake architecture, marking Intel's first heterogeneous processor, combining two different core architectures from the Core and Atom families (commonly referred to as P-Cores and E-cores) to improve performance and energy efficiency. While this heterogenity can improve performance and energy efficiency, it also increases complexity of scheduling decisions and power saving mechanisms, adding to the already complex architecture, featuring per-core Dynamic Voltage and frequency Scaling (DVFS), Idle states and Power Limiting / Thermal Protection.

Schöne et al[28] found RAPL in the Alder Lake architecture to be generally consistent with external measurements, but exhibiting lower accuracy in low power scenarios. The following figure 2.3 shows these inaccuracies, albeit tested on a conusmer-grade Intel Core i9-12900K processor measured at the base frequency of 0.8GHz.

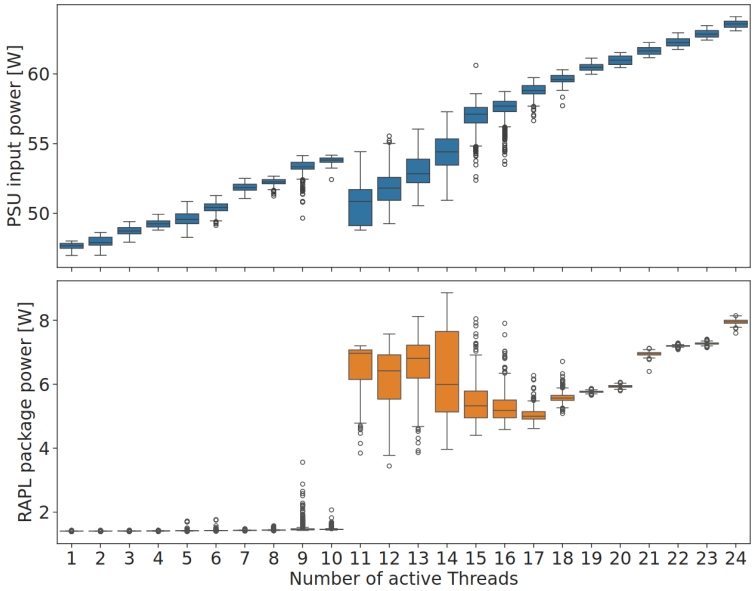


FIGURE 2.3: RAPL and reference power consumption sampled at 100 ms / 50 ms intervals respectively. Double precision matrix multiplication kernel at 0.8GHz running for 60s each at increasing number of active threads[28].

2.4.2.3 RAPL Limitations and issues

Several limitations of RAPL were noticed in various research works. Since RAPL is continually improved by Intel as new Processors are released, some of these issues have since been improved or entirely solved.

- **Register overflow:** The 32-bit register can experience an overflow error[26, 32]. This can be mitigated by sampling more frequently than the register takes to overflow. This interval can be calculated using the following equation:

$$t_{\text{overflow}} = \frac{2^{32} \cdot E_u}{P} \tag{2.1}$$

Here, E_u is the energy unit used ($61\mu J$ for haswell), and P is the power consumption. On a Haswell processor consuming 84W, an overflow would occur every 52 minutes. Intel acknowledges this in the official documentation, stating that the register has a *wraparound time of around 60 seconds when power consumption is high*[27] This is solvable with a simple correction, provided that the measurement interfals are small enough: For two successive measurements m_{prev} and m_{current}, the actual measured difference is given by

$$\Delta m = \begin{cases} m_{\text{current}} - m_{\text{prev}} + C & \text{if } m_{\text{current}} < m_{\text{prev}} \\ m_{\text{current}} - m_{\text{prev}} & \text{otherwise} \end{cases} \tag{2.2}$$

where C is a correction constant that depends on the chosen mechanism:

Mechanism	Constant C
MSR	`u32::MAX`, i.e., $2^{32} - 1$
perf-events	`u64::MAX`, i.e., $2^{64} - 1$
perf-events with eBPF	`u64::MAX`, i.e., $2^{64} - 1$
powercap	Value given by the file `max_energy_uj` in the sysfs folder for the RAPL domain

TABLE 2.2: RAPL overflow correction constant

- **DRAM Accuracy:** DRAM Accuracy can only reliably be used for the Haswell architecture[14, 31, 32], and may still exibit a constant power offset (like attributed to the voltage regulator power loss of the memory system).

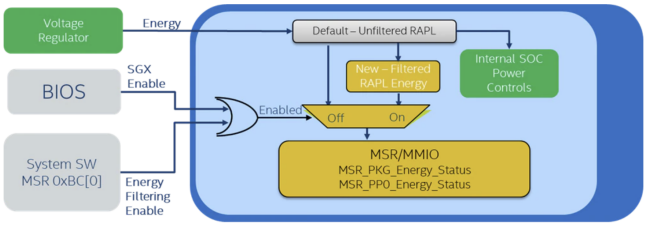
- **Unpredictable Timings:** While the Intel documentation states that the RAPL time unit is 0.976ms, the actual intervals may vary. This is an issue since the measurements do not come with timestamps, making precise measurements difficult[32]. Several coping mechanisms have been used to mitigate this, notably *busypolling* (busypolling the counter for updates, significantly compromizing overhead in terms of time and energy[33]), *supersampling* (lowering the sampling interval, increasing overhead and occasionaly creating duplicates that need to be filtered[32]), or *high frequency sampling* (*lowering* the sampling rate when the resulting data is still sufficient[34]). Another solution is to use a *low sampling frequency* to smoothe out the relative error due to spikes, with the only drawback of loss of temporal precision. At sampling rates slower than 50Hz, the relative error is less than 0.5% [24].

- **Non-atomic register updates:** RAPL register updates are nont atomic[32], meaning that the different RAPL values show a delay between individual updates. This may introduce errors when sampling multiple counters at a high sampling rate, making it possible to read both fresh and stale values of different counters.

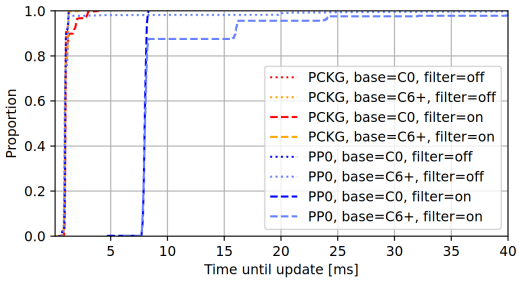
- **Lower idle power accuracy:** When measuring an idling server, RAPL tends to be less accurate[14, 28].

- **Side-channel attacks:** While the update rate of RAPL is usually 1ms, it can get as low as 50 μs for the PP0 domain (processor cores) on desktop processors[28]. This can be used to retrieve processed data in a side channel attack (coined "Platypus")[28, 35].

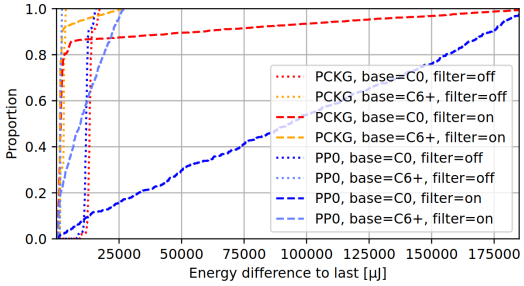
To mitigate this issue while retaining RAPL functionality, Intel implements a filtering technique via the `ENERGY_FILTERING_ENABLE`[36, Table 2-2] entry, or when *Software Guard Extension (SGX)* is activated in the BIOS. This filter adds random noise to the reported values (vizualized in Figure 2.4a). This can be seen For the PP0 domain, this raises the temporal granularity to about 8ms. While this does not affect the average power consumption, point measurement power consumption can be affected. Figure 2.4 shows the effect of the filter, clearly indicating the loss granularity resulting from the activation of the filter. In a 2022 article, Tamara[37] found a surprising higher mean with the filter activated and deemed filtered RAPL energy data unusable. In a more elaborate experiment in 2024, Schöne et al did not encounter these inaccuracies anymore.



(A) RAPL energy filtering[38]



(B) Distribution of time between updates: With an enabled filter, the PP0 domain only provides updates every 8 ms, otherwise RAPL values are updated every 1 ms. If the load is too low, some updates might be skipped, e.g., the next update for PP0 and an enabled filter is at 16 ms.



(C) The minimal increase of a measurement is 1 energy unit of 61.035 μJ. Enabling the filter leads to a significant influence on measurements for the PP0 domain and a measurable influence on PCKG measurements.

FIGURE 2.4: Observable loss of granularity caused by the activation of `ENERGY_FILTERING_ENABLE`[28]

2.4.2.4 RAPL conclusions

The energy measurement accuracy of RAPl has significantly improved since its inception and provides a generally accepted way to measure system energy consumption. It is well-validated and accepted as the most accurate fine-granular energy measurement tool. Some known limitations have historically created inaccuracies in developed measurement tools, but corrections of these limitations exist.

2.4.3 Graphical Processing Units (GPU)

In recent years, the utilization of GPUs in cloud computing environments has grown significantly, driven primarily by the increasing demand for high-performance computations in machine learning, artificial intelligence, and large-scale data processing [39]. Kubernetes now includes mechanisms for GPU provisioning, enabling containerized workloads to leverage GPU acceleration [40].

Although GPUs remain less common than traditional CPU-based workloads in typical Kubernetes clusters, their adoption is rapidly accelerating. Industry reports indicate that GPU usage in Kubernetes has seen a growth rate of nearly 58% year-over-year, outpacing general cloud computing growth rates [41]. This increase is largely attributed to ML workloads and real-time processing tasks that benefit from the parallel processing capabilities of GPUs [42]. Furthermore, hyperscalers have integrated GPU support directly into their managed Kubernetes services, reflecting the growing demand for GPU-powered workloads in containerized environments.

Despite this growth, GPU deployments are still not as pervasive as CPU-based workloads in Kubernetes-managed clusters. The primary focus of this thesis is on the measurement and analysis of energy consumption in more common, CPU- and memory-centric Kubernetes workloads. Nevertheless, due to the rising significance of GPUs, their energy measurement techniques and potential integration within Kubernetes environments are briefly examined.

Ultimately, the inclusion of GPU energy measurements remains outside the primary scope of this thesis but is acknowledged as an important area for future research. This structured exploration serves to highlight current limitations and opportunities for enhancing energy efficiency in Kubernetes-managed GPU workloads.

2.4.3.1 GPU virtualization technologies

Full GPU Virtualization Full GPU virtualization provides isolated instances of a single physical GPU to multiple virtual machines. This is achieved using technologies such as NVIDIA's *vGPU* or AMD's *MxGPU (Multiuser GPU)*. These technologies allow a VM to see a complete GPU, while the underlying hypervisor manages resource partitioning and scheduling [43, 44] eigher though the use of partitioning or time-slicing. In a Kubernetes environment, full GPU virtualization is commonly utilized through:

- **vGPU on VMware or OpenStack:** Kubernetes clusters running on VMware vSphere or OpenStack can request vGPU instances as if they were physical GPUs. These instances are shared among containers while maintaining memory and compute isolation.
- **Device Plugin Integration:** NVIDIA, AMD and Intel provide a Device Plugin for Kubernetes, enabling seamless GPU discovery and allocation across pods [40].

Multi-Instance GPU (MIG) Introduced with the NVIDIA A100 architecture, Multi-Instance GPU (MIG) allows a single GPU to be partitioned into up to seven independent instances, each with its own dedicated compute, memory, and cache resources [45]. Unlike traditional vGPU, MIG provides true hardware-level isolation, preventing noisy-neighbor effects and enabling finer resource allocation. MIG instances are exposed to Kubernetes as individual GPUs. For example, a single A100 GPU partitioned into seven MIG instances appears as seven separate GPU resources, each assignable to different containers. MIG-aware device plugins ensure proper scheduling and isolation. Hence, MIG technology is particularly useful for multi-tenant environments and supports finer granularity in resource allocation compared to traditional vGPU models.

GPU Passthrough GPU passthrough allows a physical GPU to be exclusively assigned to a single VM or container. Unlike virtualization, where resources are shared, passthrough dedicates the full GPU to one environment, offering near-native performance [46]. GPU passthrough is configured at the hypervisor level (e.g., KVM or VMware ESXi) and can be exposed to Kubernetes nodes. Pods scheduled on nodes with GPU passthrough access gain complete control of the GPU, enabling direct memory access and high-performance computation.

GPU virtualization technologies enable efficient multi-tenant use of GPU resources, enhancing performance and cost-effectiveness in cloud-native environments. For the purposes of energy measurement, understanding these virtualization layers is essential for accurate per-container energy attribution.

2.4.3.2 GPU nvidia-NVML energy measurements and validation

Modern GPUs are equipped with **built-in power sensors** that enable real-time energy measurement. For instance, Nvidia GPUs expose power metrics through the *Nvidia System Management Interface (nvidia-smi)*, which reports instantaneous power draw, temperature, and memory usage [47]. This interface allows for programmatic access to GPU power consumption, making it a common choice for monitoring and energy profiling in both standalone and containerized environments [45].

In 2024, Yang et al. conducted a comprehensive study on the accuracy and reliability of NVIDIA's built-in power sensors, examining over 70 different models[48]. He concludes that previous research placed excessive trust in nvidia-NVML, overlooking the importance of measurement methodology. The study revealed several critical findings:

- **Sampling Limitations:** Nvidia NVML gives the option to specify a sampling frequency in units of milliseconds. However, on certain models, such as the A100 and H100, power is sampled only around 25% of the time, introducing potential inaccuracies in total energy consumption estimations.
- **Transient Response Issues:** While measured power reacted instantly to a suddenly applied workload, nvidia-NVML would report values with a delay of several hunderd milliseconds on some devices. Also, a slower rise (with linear growth) was discovered, taking over a second to catch up to correct power figures in some instances. Generally, server-grade GPUs were shown to provide more instantaneous power measurements.
- **Measurement Inaccuracies:** The average error rate in reported power draw was found to be approximately 5%, deviating from NVIDIA's claimed fixed error margin of 5W. This error would remain consistent when the GPU reached a constant power draw.
- **Averaging Effects:** Reported power consumption values are averaged over time, masking short-term fluctuations and potentially underreporting peak consumption.

To address these limitations, the study proposed best practices such as running multiple or longer iterations of workloads to average out sampling errors, introducing

controlled phase shifts to capture different execution states, and applying data corrections to account for transient lags [48]. These adjustments reduced measurement errors by up to 65%, demonstrating the importance of refining raw sensor data for more accurate energy profiling.

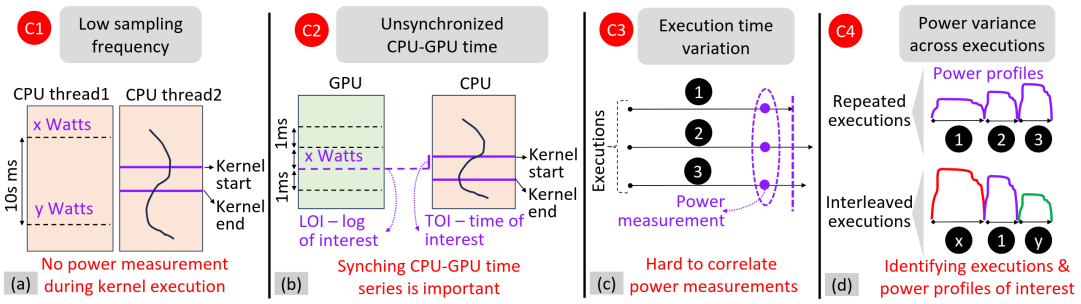
2.4.3.3 Related Research

While most reaseach has used nvidia-NVML to measure GPU power consumption, some research was done on alternative measurement tools, ususally to address similar issues as were stated by Yang et al in the previous section. Specifically, the following three tools focussed were proposed to provide higher sampling rates to enable finer-grained power analysis.

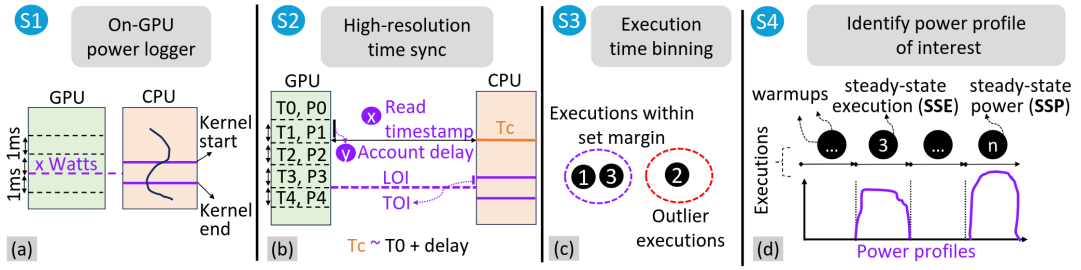
AccellWattch In 2021, Pan et al proposed *AccelWattch*[49], a configurable GPU power model that provides both a higher accuracy cycle-level power model, and a way to measure constant and static power, utilizing any pure-software software performance mode,nvidia-NVML, or a combination of the two. Notably, their model is DVFS-, power-gating- and divergence-aware. The resulting power model was validated against measured ground truth using an Nvidia Volta GV100, yielding a MAPE error between $7.5 - 9.2 \pm 2.1\% - 3.1\%$, depending on the AccelWattch variant. The Volta model was later validated against Pascal TITAN X and Turing RTX 2060-architectures without retraining, achieving $11 \pm 3.8\% and 13 \pm 4.7\%$ MAPE, respectively. The authors conclude that AccelWattch can reliably predict power consumption of these specific GPU architectures. In the context of Kubernetes energy consumption, AccelWattch contributes a fine-grained temporal granularity

FinGraV In 2024, Singhania et al propose *FinGraV*[50] (abbreviated from **Fin**e-**Gra**in **V**isibility), a fine-grained power measurements tool capable of sub-millisecond power profiling for GPU executions on an AMD MI300X GPU. They identify these main challenges of high-resolution GPU power analysis (see figure 2.5a):

- **Low sampling frequency:** Standard GPU power loggers operate at intervals too coarse (tens of milliseconds) to capture the sub-millisecond executions of modern kernels.
- **CPU-GPU time Synchronization:** Synchronizing power measurements with kernel start and end times is problematic due to the asynchronous nature of CPU-GPU communication.
- **Execution time variation:** Minor variations in memory allocation or access patterns lead to inconsistent kernel execution times, complicating time-based power profiling.
- **Power variance across executions:** Repeated executions of the same kernel, or interleaved executions with other kernels, manifest in fluctuating power consumption, challenging consistent profiling.



(A) Challenges in fine-grain GPU power analysis



(B) FinGraV strategies to adress challenges

FIGURE 2.5: FinGraV GPU power measurement challenges and strategies[50]

To overcome these challenges, FinGraV introduces several strategies (see figure 2.5b).

- **On-GPU Power Logger:** FinGraV leverages a high-resolution (1 ms) power logger, capturing the average of multiple instantaneous power readings.
- **High-Resolution Time Synchronization:** GPU timestamps are read from the CPU side before kernel execution, and synchronization is maintained throughout execution to correlate power samples with kernel events.
- **Execution Time Binning:** Kernel executions are grouped into "bins" based on empirical runtime ranges, enabling tighter power profiling while discarding outlier runs.
- **Power Profile Differentiation:** FinGraV distinguishes between Steady-State Execution (SSE) and Steady-State Power (SSP) profiles. SSP represents the stabilized power consumption after initial transients, providing the most accurate depiction of kernel power consumption.

The application of FinGraV to bencharmks reveals several critical obervations: Kernel executions differ significantly between initial runs and steady-state, with deviations up to 80%. Memory-bound kernels and compute-light kernels are found to be highly sensitive to the preceding kernel, impacting their power profile. Furthermore, the authors expose discrepancies in GPU power scaling relative to computational load, particularly for compute-light kernels.

FinGraV introduces promising concepts that could, in theory, enable more granular and accurate GPU power analysis in container-based GPU workloads. Its methodological approach addresses key challenges in sub-millisecond power measurement. However, its current implementation is tightly coupled with the AMD MI300X GPU,

relying on hardware-specific logging capabilities that are not universally available. While the underlying concepts may be extendable to other GPUs, achieving this is far from trivial, requiring significant adaptation and low-level access to power metrics that are often proprietary or limited by driver capabilities.

Consequently, FinGraV highlights both the challenges and potential solutions for fine-grained GPU power analysis but falls short of providing a general-purpose framework that could be easily integrated into Kubernetes energy measurement tools. It also underscores the broader issue that GPU energy consumption analysis remains relatively immature, with only vendor-specific tools like nvidia-NVML offering practical (but coarse) power metrics. This illustrates that while the methodology is theoretically sound, practical implementation across diverse GPU architectures remains a significant challenge.

PowerSensor3 *PowerSensor3*[51] is an open-source hardware tool introduced in 2025, designed to provide high-resolution power measurements for GPUs, SoC boards, PCIe devices, SSDs, and FPGAs. Unlike software-based power models or vendor-specific tools such as NVIDIA's NVML, PowerSensor3 achieves significantly higher accuracy and granularity through direct voltage and current measurements at a sampling rate of up to 20 kHz. This fine temporal resolution allows it to capture transient power behaviors that are typically missed by software-based methods, which are constrained by lower sampling frequencies and indirect estimations. As expected for a purpose-built hardware solution, PowerSensor3 outperforms NVML in both precision and the ability to detect rapid changes in power consumption.

A particularly valuable feature of PowerSensor3 is its capability to monitor not only GPUs but also other critical components such as SoC boards, PCIe-connected accelerators, and storage devices like SSDs. For Kubernetes-based energy efficiency analysis, this would provide unprecedented visibility into the power usage of individual containers, extending monitoring beyond the CPU and GPU to the broader spectrum of peripherals that contribute to overall energy consumption. Such granularity could enhance resource scheduling and energy optimization in containerized environments.

However, while its technical benefits are evident, the practical deployment of dedicated hardware sensors like PowerSensor3 at scale remains both complex and expensive. Integrating such devices across large Kubernetes clusters would require substantial investment in hardware and reconfiguration of infrastructure, making wide adoption unlikely outside of specialized research environments. Consequently, PowerSensor3 and other hardware-dependent methods are not considered in the scope of this thesis. Furthermore, the very recent introduction of PowerSensor3 in 2025 highlights the ongoing challenges of accurate energy monitoring through software alone, reflecting the current gap in reliable, scalable, software-based power measurement solutions.

2.4.3.4 GPU Limitations in Kubernetes Context

The analysis of GPU power consumption has revealed promising research efforts aimed at achieving fine-grained power visibility and energy optimization. Tools such as FinGraV and PowerSensor3 demonstrate that significant strides are being

made in capturing detailed power metrics with high temporal resolution and sub-component granularity. FinGraV addresses the complexities of short-lived GPU kernel executions through innovative profiling methodologies, while PowerSensor3 delivers hardware-level accuracy for GPUs, SoC boards, and various PCIe-connected peripherals. These solutions underscore the potential for more refined power monitoring in high-performance GPU workloads.

However, the current state of GPU energy consumption measurement presents significant challenges for scalable, container-based energy tracking in Kubernetes environments. Research tools like FinGraV and PowerSensor3, while technically robust, are either hardware-dependent or too tightly coupled to specific architectures (such as AMD's MI300X in the case of FinGraV). Hardware-based solutions like PowerSensor3, though highly accurate, are impractical for widespread deployment due to cost and scalability concerns. Meanwhile, software-based vendor solutions such as NVIDIA's NVML are far more accessible, but suffer from limitations in temporal granularity and measurement accuracy. These tools offer convenient integration and broad support across data center infrastructures but struggle with capturing rapid transients in power consumption, which are crucial for real-time container energy attribution.

In the context of this thesis, GPU energy consumption is acknowledged as an important yet currently impractical aspect of container energy measurement. The relative immaturity of fine-grained, scalable monitoring solutions for GPUs, combined with the relatively small role of GPUs in Kubernetes clusters, justifies this exclusion. Although the utilization of GPU accelerators in Kubernetes environments is expected to grow, current measurement methods do not yet support the level of precision and scalability required for effective implementation. As such, this thesis will focus on more readily measurable server components, with the understanding that future advancements in GPU power analysis may enable their integration into Kubernetes-based energy efficiency strategies.

2.4.4 Storage Devices

Various sudies have investigated the power consumption of Storage devices. In 2008, Hylick et al[52] investigated real-time HDD energy consumption and found significant differences in power consumption between standby, idle and active power states. Cho et al[53] propose various energy estimation models for SSDs after measuring and comparing the energy consumption of different models. The most notabble model-based energy consumption estimation methanisms are presented in section 2.5.4

In constrast to CPU or GPU components, storage devices (HDD, DDS or NVMe-drives) cannot make use of physical power sensors. While a BMC-measurement-based solution would technically be feasable, real-world implementation is impractical: While a BMC might me able to measure the power supply to a storage device, it typically is not exposed through IPMI or redfish. Such measurements would further be compllicated by the use of backplane devices, making measurements for individual devices impossible. For these reasons, storage device energy consumption is typically modelled, not measured (see section 2.5.4).

While storage devices don't expose any energy-consumption specific metrics, many other related metrics are available (and can be used for modeling approaches):

- `NVMe-cli`[54] exposes many metrics of NVMe-drives, including the maximum power draw for each power state (including idle power), the number of power states supported, the current power state and temperature, and others.

- `smartctl`[55] exposes metrics of the *SMART (Self-Monitoring, Analysis and Reporting Technology)*-Functionality implemented in many modern storage drives. While these metrics are vendor-specific, they often include temperature metrics, throughput, and other metrics. Often, HDD speed is exposed. Notably, *SMART* metrics are typically more focussed on lifecycle information such as power-on hours, wear indicators and others.

- Many other performance metrics are exposed by various tools such as `iostat`, `sar`, `/proc/diskstats`, and `blkstat`, such as read/write IOPS, throughput, queue length, latency, utilization, and others. Additional information (such as the interface) is also exposed.

2.4.5 Network devices and other PCIe devices

Peripherals like the Network Interace Card (NIC) are almost always connected via PCIe. As such, many cards support Device power states[56] as specified by the PCIe specifications. Notably, not all NICs support all (or any) power states.These device states allow the the server and device to negotiate a power state for the device, which typically means choosing a trade-off between power consumption and wake-up latency. For devices, PCIe specifies the following device states:

- D0 state (Fully on)

- D1 and D2 states (Intermediate power states)

- D3 State (Off State), with the distinction between D3hot and D3cold

Unfortunately, device power states are not in any way related to physical power specifications: While a specific power state might be useful for simple deductions (e.g. if a device is idling or active), no power figures can be deduced. In the event that a device's idle or maximum power are known, power states might potentially be used for a first estimation (i.e. an idling device is unlikely to consume its specified maximum power, and vice versa), but since a device's power characteristics cannot reliably estimated (especially beyond just NICs), devices power states cannot be used to reliably estimate device power consumption. An attempt at the estimation of NIC power consumption is covered in section 2.5.5.

2.5 Model-based estimation techniques

In the absence of actual power data, power consumption models can be formulated that map variables (such as CPU or Memory utilization) related to a server's state to its power consumption. Due to the strong correlation between CPU uzilization and server power, a great number of models use CPU metrics as the only indicator of server power. Fan et al[57] proposed a linear interpolation between idle power and full power, which they further refine into a non-linear form, with a parameter γ to be fitted to minimize mean square error. Similar research was done to further reduce error by introducting more complex non-linear models, such as Hsu and Poole[58], who studied the SPECpower_ssj2008-dataset of systems released between December

2007 and August 2010, and suggested the adaptation of two non-linear terms:

$$P_{\text{server}} = \alpha_0 + \alpha_1 u_{\text{cpu}} + \alpha_2 \left(u_{\text{cpu}}\right)^{\gamma_0} + \alpha_3 \left(1 - u_{\text{cpu}}\right)^{\gamma_1} \tag{2.3}$$

The division of server power consumption into idle (generally static) and dynamic power (modeled with many different methods thoughought related research) has historically been a popular suggestion[59]. Other, broadly similar attempts to model server energy consumption based on only a few variables exist, such as modeling server consumption based on CPU frequency[60].

2.5.1 Component-level Power models

While models like the ones listed above might work well when custom-fitted to specific, multi-purpose servers, they have since been surpassed by the more common approach of modelling server power is to consider it an assembly of its components, such as Song et al[61] propose as:

$$P_{\text{server}} = P_{\text{cpu}} + P_{\text{memory}} + P_{\text{disk}} + P_{\text{NIC}} + C \tag{2.4}$$

where C denotes the server's base power, which includes the power consumption of other components (regarded as static). This approach can easily be extended to include various other components such as GPUs, FPGAs or other connected components.

2.5.1.1 Advantages and disadvantages of component-level power models

A component-based approach to modeling server power consumption offers increased granularity and adaptability across a diverse range of server architectures. Modern data centers deploy heterogeneous hardware configurations optimized for specific workloads, such as CPU-intensive computing nodes, GPU-accelerated servers for machine learning, or memory-rich systems for in-memory databases. These configurations lead to vastly different power distribution profiles across components[62]. By modeling the energy consumption of individual components, it becomes possible to reflect these structural differences more accurately. Additionally, such models can reveal energy characteristics that would be obscured in aggregate metrics (for instance, a workload that imposes significant stress on storage devices without engaging the CPU may go undetected in simplistic, CPU-centric models). Finally, component-level analysis enables more precise evaluation of energy optimization techniques: the impact of mechanisms like dynamic voltage and frequency scaling (DVFS) or idle power states can be assessed not just in isolation but in terms of their contribution to overall server efficiency.

Despite offering finer granularity, component-based power modeling faces several inherent challenges. While servers are composed of individual components, they function as tightly integrated systems in which no component operates in isolation. The power consumption of one subsystem often depends on the behavior of others (for example, memory access patterns can influence CPU power states, and I/O activity may trigger CPU wake-ups or increased cache usage). These inter-component interactions are difficult to capture accurately and are frequently overlooked in component-level models[13], leading to potentially misleading or incomplete estimations. Furthermore, the development of detailed and accurate models

for each component is significantly more complex than holistic server-level modeling. Such models often require extensive empirical data, sophisticated estimation techniques, and continuous updates to remain valid across hardware generations. This not only increases the research burden but also demands a higher level of expertise for interpretation and practical application, compared to simpler, utilization-based models.

2.5.2 CPU

Existing CPU power models generally model CPU power as a combination of other, existing power figures. Fan et al[57] propose a linear interpolation between idle power and full power based on CPU utilization. Basmadjian et al[63] observe that individual cores in a multi-core CPU can be modeled as individual cores, in addition to an overall CPU idle consumption. Non-linear models are also widely adopted, with Lou et al[64] proposing a polynomial model as a univariate function of CPU utilization. Other models include individual CPU components[65, 66] for more fidelity.

While these models may be helpful to examine the dynamics of CPU power consumption in relation to different inputs, they are not helpful in finding CPU power consumption of an unknown CPU, i.e. without previously known idle or maximum power consumption. The existance of a model capable to accurately estimation CPU power consumption based solely on generalizable input factors (such as utilization or frequency) is questionable due to the great variance in architectures and technologies, as well as technological progress. Unsurprisingly, the author of this thesis was not able to find a model to estimate CPU power consumption.

2.5.3 Memory

Many Memory power models were proposed in literature, many of them with an idle and a dynamic component of memory power consumption. While the idle power consumption is generally assumed to be known, dynamic memory power consumption has been modeled to depend on memory usage[67] or memory accesses[68], the number of cache misses[69], memory state[63], or other factors. Similar to the CPU models presented in the previous section, these models do not propose generalizable models able to predict memory consumption in situations where idle or maximum memory consumption is not previously known, instead focussing on examining power consumption dynamics. The same objections preventing generalizable CPU models apply to memory models as well, namely great variety, specialization and technological progress. As a result, no model-based approaches exist that are capable of accurately estimating memory power consumption based solely on performance metrics.

2.5.4 Storage devices

2.5.4.1 Generalization-based estimation

There is an urgent need in the storage industry for research into the area of workload-dependant power estimation[70]. Estimating the energy consumption of a storage device is challenging especially due to the great variation between different devices. Some of these model variables can be determined on a running server system (e.g. device type, I/O operation type, access pattern, workload intensity, state and more),

while other variables are unknown to the server (e.g. Flash Transition Layer and flash factor, NAND organization, garbage collection, and more). A large storage device market has led to a high variation in devices with sometimes drastically different target uses (e.g. low-latency storage devices, high-concurrency storage devices, low-power storage devices).

Storage controllers further complicate the energy consumption estimation of storage devices by introducing an additional layer of abstraction between the operating system and the physical storage hardware. Their internal operations consume energy independently of the actual read/write workload observed by the host system. This makes it difficult to directly correlate application-level I/O activity with actual device-level power usage. Moreover, in many server configurations, multiple drives are managed behind a single controller, obscuring per-device energy attribution and introducing variability that model-based estimations often cannot accurately capture.

Scope clarification In this thesis, only storage devices physically installed in the server are considered for power estimation. This includes devices such as HDDs, SSDs, and NVMe drives directly attached to the server. Dedicated external storage systems such as Storage Area Networks (SAN) or Network-Attached Storage (NAS) are not within the scope of this analysis. While such systems are important in data center environments, their energy consumption is not attributable at the granularity required for the workload-level estimation pursued in this thesis.

As a result, research into storage device energy consumption measurement that is generally applicable to all devices has been limited. For practical applications, generalizations are often used, such as the following tables 2.3a to 2.3d. While these approximations cannot be used in the context of this thesis, they may serve as an initial guideline.

HDD Type	Read/write power (W)	Idle Power (W)	Standby Power (W)
HDD (2.5″ SATA)	1.5 – 3.0	0.5 – 1.2	0.1 – 0.3
HDD (3.5″ SATA)	6 – 12	4 – 8	0.5 – 2.0
HDD (Enterprise)	7 – 15	5 – 10	0.5 – 2.5

(A) Typical HDD power consumption[71]

HDD Type	Read/write power (W)	Idle Power (W)	Standby Power (W)
5400 RPM HDD	6 – 9	4 – 6	0.5 – 1.5
7200 RPM HDD	8 – 12	6 – 8	0.6 – 1.8
10,000+ RPM HDD	10 – 16	8 – 12	1.0 – 2.5

(B) Common HDD RPM power consumption[71]

SSD Type	Read Power (W)	Write Power (W)	Idle Power (W)
2.5″ SATA	4.5 – 8	4.5 – 8	0.30 – 2
mSATA	1 – 5	4 – 8	0.20 – 2
M.2 SATA	2.5 – 6	4 – 9	0.40 – 2

(C) Typical SATA SSD power consumption[72]

NVMe Type	Read/write power (W)	Peak Power (W)	Standby Power (W)
M.2 NVMe PCIe 3.0	3 – 5	6 – 9	0.4 – 1.5
M.2 NVMe PCIe 4.0	5 – 7	8 – 12	0.5 – 2
M.2 NVMe PCIe 5.0	8 – 12	12 – 18	0.8 – 3

(D) Typical NVMe SSD power consumption[72]

TABLE 2.3: Power consumption for various storage device types.

Apart from simple estimations like shown in tables 2.3a to 2.3d, a few works have concentrated on the energy consumption of individual storage devices. In 2015, Cho et al developed Energysim[53], an SSD energy modeling framework avancing the understanding of component-level (i.e. the subcomponents of a storage device) energy consumption in storage devices. Its validation against real-world SSD measurements against an Intel X25-M yielded a less than 8% error. The work underscores the difficulty of modeling storage energy accurately due to high variability across architectures and workloads. Unfortunately, Energysim uses many model parameters such as NAND organization, idle and active current consumption, and as a result cannot reneralized to other storage devices where these are unknown.

In 2014, Li and Long[73] present a workload-aware modeling framework to estimate the energy consumption of storage systems, challenging the assumption that SSDs are inherently more energy-efficient than HDDs. By classifying I/O workloads into capability workloads (performance-driven) and capacity workloads (storage size-driven), they develop mathematical models that account for the number of devices needed, workload execution time, and device power states (active, idle, standby). Their validation, based on empirical measurements using Seagate HDDs and a Samsung SSD, shows that SSDs are generally more efficient for high-performance workloads, while HDDs can outperform SSDs in archival or low-access scenarios,particularly when effective power management (e.g., spin-down) is employed. Unfortunately,

similar to the research by Cho et al, the presented models make use of various non-generalizable variables, most notably a devices idle, standby and busy power consumption. In the context of these thesis, these are unknown and the presented model consequently cannot be applied.

2.5.4.2 GSPN Modeling for hybrid storage systems (active power states)

In 2022, Borba et al[74] proposed a number of models based on generalized stochastic Petri nets (GSPN) for performance and energy consumption evaluation for individual and Hybrid (HDD + SSD) storage systems. GSPN is a suitable formalism for storage system design, as, differently from queueing network models (for instance), synchronization, resource sharing, and conflicts are naturally represented. Also, phase approximation technique may be applied for modeling non-exponential activities, and events with zero delays (e.g., workload selection) may adopt immediate transitions.

The authors propose a single-storage model (either for a single storage device or a hybrid system as a blackbox) and a multiple storage model.

The Hybrid storage power consumption model proposed by Borba is parameterized by I/O type (read/write), access pattern (sequential/random), object size (4KB, 1MB), and thread concurrency. The model explicitely imcorporates power consumption per operation (e.g. random-read-4KB on SSD).

The following notation is adoped:

- $E\{\#p\}$ represents the mean value of the inner expression, in which $\#p$ denotes the number of tokens in place.
- $W(T)$ represents the firing rate associated with transition T.
- $\eta : T_{\text{imm}} \rightarrow [0,1]$ maps each immediate transition ($t \in T_{\text{imm}}$) to a normalized weight. Weights represent the transition firing probablilty in a conflict set.
- $pRequests(N)$ denotes the amount of concurrent requests from simultaneous clients (workers)

Single-device storage energy consumption is estimated as follows:

$$\begin{aligned} EP_w = \kappa \cdot (&EP_{w1} \cdot \alpha \cdot \beta + EP_{w2} \cdot (1-\alpha) \cdot \beta \\ &+ EP_{w3} \cdot \alpha \cdot (1-\beta) + EP_{w4} \cdot (1-\alpha) \cdot (1-\beta)) \end{aligned} \tag{2.5}$$

$$\begin{aligned} EP_r = (1-\kappa) \cdot (&EP_{r5} \cdot \alpha \cdot \beta + EP_{r6} \cdot (1-\alpha) \cdot \beta \\ &+ EP_{r7} \cdot \alpha \cdot (1-\beta) + EP_{r8} \cdot (1-\alpha) \cdot (1-\beta)) \end{aligned} \tag{2.6}$$

$$EC = (EP_w + EP_r) \cdot TH \cdot \text{time} \tag{2.7}$$

where EP_w and EP_r are the mean power consumption for a read (r) or write (w) operation, which is estimated using the mean power of each workload feature. For instance, EP_{w1} denotes the power of a write operation (w) using random access (α) and a small object (β). System throughput (i.e., IOPS) is estimated as $TH = E\{\#p_{\text{Ack}}\} \times W(t_{\text{Communicating}})$.For the single device model, the following weights are taken into

account: $\eta(t_{\text{Write}}) = \kappa$; $\eta(t_{\text{Read}}) = 1 - \kappa$; $\eta(t_{\text{Random}}) = \alpha$; $\eta(t_{\text{Sequential}}) = 1 - \alpha$; $\eta(t_{\text{Small}}) = \beta$; and $\eta(t_{\text{Large}}) = 1 - \beta$.

The marking of place $pResource(R)$ (for both read or write activity) may denote the adopted technology. For instance, for traditional SSDs (SATA interface), the marking place *pResource* is 1, as only one operation it the time is carried out. Concerning SSDs-NVMe, *pResource* assumes the number of threads of concurrently processing I/O requests (generally 8).

The proposed multi-storage model expands the model for multiple devices:

$$EC_h = \left(\sum_{d=0}^{n} \eta(tForward_d) \cdot EP_d \right) \cdot TH_h \cdot \text{time} \tag{2.8}$$

where the immediate transitions $tForward_d$ denote a request redirection to storage d.

Validation The model proposed by Borba et al. was validated using controlled experiments with the Fio benchmarking tool, which generated synthetic I/O workloads to measure and correlate storage system performance and energy consumption across varying request sizes, access patterns, and read/write ratios. Model estimates consistently falling within the 95% confidence intervals of observed system metrics. This statistical consistency indicates that the model's predictions are not significantly different from real-world values, supporting its applicability for performance and energy analysis in large-scale storage systems.

Limitations The authors acknowledge that a large number of devices significantly increases modeling complexity due to state space size explosion and recommend simulation as a viable workaround. Additionally, the authors acknowledge their focus on active energy states (not idle, standby states or state transitions), treating them as delays between requests.

In a running server system, this approach could be adapted to create an accurate and fine-grained energy consumption estimation of a read/write workload on specific storage devices, albeit with limitations:

- Instead of needing to be estimated, (device-specific) throughput (TH) can be measured.
- An intial calibration run is necessary to experimentally determine the respective device-specific variables.
- In a multi-storage device server, the resulting state explosion may lead to significan calculation overhead, resulting also in higher energy consumption of the measurement itself.
- Instead of modelling transitions to a storage device as a function (as done in $\eta(tForward_d)$), device usage would actively need to be measured, which would essentially transform the multi-storage model into a simple addition of single-storage models. This would drastically reduce the number of total states, making calculations less demanding.

- Due to the authors not considering idle and standby-states, a small, constant idle power consumption would need to be added to the model. This is especially important for accurate storage device power consumption modeling on idling or overprovisioned servers.

2.5.5 Network devices

Estimating the total power consumption of a network infrastructure requires a clear definition of system boundaries. Since most server clusters operate within larger, interconnected systems, a full assessment of network energy consumption (such as for CO_2 footprint calculations) is generally infeasible. This thesis limits the system boundary to the server itself, considering only internal network components, primarily the Network Interface Card (NIC). While this allows detailed modeling of NIC power usage, it excludes broader network activity, such as inter-node communication in multi-node clusters.

Although the overall energy consumption of a data center network could be estimated by including access, aggregation, and core switches, attributing this consumption to specific workloads remains highly challenging. This chapter therefore focuses on model-based methods for estimating NIC-level power as a proxy for server-side network energy usage.

2.5.5.1 NIC power consumption characteristics

While a lot of research was done to analyze the power consumption of network equipment like switches, routers or gateways, NICs (especially non-wireless NICs) have not received as much attention. While there are several methods that modern NICs employ to save power (e.g. PCIe Link power states and D-states, *Active State Power Management (ASPM)*) or *Energy Efficient Ethernet (EEE)*, there are not widely available mechanisms for fine-grained NIC power consumption estimation. As a consequence, NIC power can only be approximated based on the few available metrics,

Sohan et al[75] measured and compared the power consumption of six 10 Gbps and four multiport 1 Gbps NICs at a fine-grained level. While he does not provide a method to estimate NIC energy consumption, and notices great variation in power consumption between different NICs. Unfortunately, it cannot be ruled out that some of the results are cherry-picked: Solarflare-NICs tend to dominate the introduced metics, and a Communication spokesperson is prominently credited with contact information. Regardless, some findings are found irrespective of the NIC manufacturer, and remain consistent with other literature sources[76]. While these findings cannot directly contribute to a potential NIC power consumption estimation approach, they are relevant to understand underlying mechanisms and to assess the relative importance of the NIC compared to other server components.

- Idle Power
  - The measured NICs show a power consumption of between 5–20W
  - Link connection status has little effect on idle energy consumption

  - Physical media influences power consumption: CX4 models have the lowest power consumpttion due to the simple design of the CX4 interconnect. This is followed by Fober models. Finally the Base-T models consume significantly more power due to the signal processing component int he card.

- Active Power
  - There is very little difference in the power usage of an active NIC compared to an idle one. For all measured NICS, the differenc in power usage was less than 1W.
  - Throughput performance varied widely, and no correlation between power usage and performance was observed.
  - Power consumption increases in correlation to the number of ports.

In 2012, Basmadjian et al[77] modelled a NIC by separating NIC power consumption into idle mode and dynamic mode (same as they did for their CPU and RAM models). If $P_{NIC_{idle}}$ is the power of the idle interface and $P_{NIC_{dynamic}}$ is the power when active, the total NIC energy consumption would be given by

$$E_{NIC} = P_{NIC_{idle}} T_{idle} + P_{NIC_{dynamic}} T_{dynamic} \tag{2.9}$$

where T_{idle} and $T_{dynamic}$ are the total idle and dynamic times, respectively. Consequently, the average power during period T would be given by

$$P_{\text{NIC}} = \frac{(T - T_{\text{dynamic}}) P_{NIC_{idle}} + P_{NIC_{dynamic}} T_{\text{dynamic}}}{T} \tag{2.10}$$

$$= P_{NIC_{idle}} + (P_{NIC_{dynamic}} - P_{NIC_{idle}})\rho \tag{2.11}$$

where $\rho = \frac{T_{\text{dynamic}}}{T}$ is the channel utilization. While this formula is only helpful when NIC idle and max power consumption are already known, we can additionally see that the NIC power consumption is assumed to rise linearly with channel utilization.

Arjona Aroca et al[78] modeled NIC efficiency based on their previous measurements. They find that NIC efficiencies for both sending and receiving are almost linear with the transfer rate and deduct a linear dependency to the network throughput.

In 2016, De Maio et al[79] proposed a network energy consumption model for node-to-node transfers in order to estimate the entire energy consumption required for virtual machine migration. Unfortnuately their model does not specifically handle NIC energy consumption, opting for modelling the entire node energy consumption of the node.

Another approach is presented by Dargie and Wen[80] in 2013, who use stochastic modelling to examine the relationship between the utilization of a NIC and its power consumption, expressing these quantities as random variables or processes. They use curve fitting to determine the relationship between utilization and measured energy consumption of their specific NIC, after creating a data set using a SPECpower benchmark. They assume a uniformly distributed bandwidth utilization in the interval [0,125] MBps. Interestingly, their model shows only a slight effect

of utilization on the predicted power consumption, mirroring the findings of Sohan et al.

The most recent NIC power model was proposed by Baneshi et al[81] in 2024, analyze per-application energy consumption. The authors cite that NIC idle power consumption may contribute up to 90% of the total NIC energy consumption. They propose the following model for per-application NIC power consumption:

$$E_{\text{active}} = \sum_i \left(BW_i \cdot T_{\text{interval}} \cdot \frac{P_{\max} - P_{\text{idle}}}{BW_{\text{aggregated}}} \right) \tag{2.12}$$

$$E_{\text{idle}} = \sum_i \left(BW_i \cdot T_{\text{interval}} \cdot \frac{P_{\text{idle}}}{BW_{\text{used}}} \right) \tag{2.13}$$

where BW_i is the bandwidth of application i, $BW_{\text{aggregated}}$ is the aggregate bandwith of both the uplink and downlink of the NIC, and BW_{used} is the used bandwidth of links (uplinks, downlinks or both). The authors combine these formulae with power figures of their specific use case (Total network power consumption in a fog computing scenario), which unfortunately are not applicable in the context of this thesis. Regardless, while these formulae cannot be used to estimate the maximum and idle NIC power, they can be applied irrespective of server specifications in the event that idle and maximum power NIC power consumption are known.

In contrast to the formulae presented by Basmadjian and Arjona Aroca, the formulae not only account for time intervals, but also bandwidth used. As a result, the formulae presented by Baneshi et al represent the current best approach to estimate NIC power consumption, even though this estimation still requires an initial guess of the idle and maximum power consumption. The author of this thesis is not aware of a more detailed formuala currently available. A generalizable formula for estimating overall NIC power consumption is unlikely to exist due to the vast variety of NICs, and the great differences between manufacturers (as found by Sohan et al).

2.5.6 Other devices

While much of the research on server energy consumption focuses on primary components such as the CPU, memory, storage, and network interfaces, a complete energy model must also account for additional hardware subsystems. These include the motherboard, power supply unit (PSU), system fans, and potentially other auxiliary devices. Though their individual energy consumption may appear minor compared to high-performance components, they collectively contribute a non-negligible share to the overall server power draw.

Despite their importance, these secondary components have received limited attention in energy modeling literature. In most cases, they are either omitted or treated as part of the residual power not attributable to the main computational subsystems.

The motherboard, for instance, includes voltage regulators, chipset logic, and peripheral interfaces. While these components may not be individually monitored, the Baseboard Management Controller (BMC) (see section 2.3.2.1)may expose aggregated power telemetry via vendor-specific sensors or interfaces like IPMI or Redfish. However, this level of detail varies greatly across hardware platforms and is seldom fine-grained enough for component-level attribution.

2.5.6.1 PSU

The power supply unit (PSU) is another often-overlooked consumer. When modeling the power usage of a PSU itself (distinct from the power it delivers to other components), the key factor is its conversion efficiency. PSUs consume more power than they deliver due to losses during AC–DC transformation and voltage regulation. According to Basmadjian et al. [77], the power consumed by the PSU can be approximated for various scenarios: If the monitoring system provides information at the PSU level, its power consumption is given by

$$P_{PSU} = \frac{measuredPower \cdot (100 - e)}{100} \tag{2.14}$$

where e is the efficiency of the PSU.

If the monitoring system proves information at the server level, the power consumption of any of the n PSU is given by the following formuala, assuming that measured power is evenly distributed among PSUs:

$$P_{PSU} = \frac{\frac{measuredPower}{n} \cdot (100 - e)}{100} \tag{2.15}$$

If the monitoring system does not provide PSU power consumption, it can be deduced by

$$P_{PSU} = \frac{P_{Mainboard} + P_{Fans}}{n \cdot e} \cdot 100 - \frac{P_{Mainboard} + P_{Fans}}{n} \tag{2.16}$$

2.5.6.2 Fans

Cooling systems, particularly fans, also represent a meaningful share of the total energy budget. Most servers employ multiple fans controlled via Pulse-Width Modulation (PWM). While the BMC or operating system tools (e.g. `lm-sensors`) often report fan RPM or PWM duty cycle, actual fan power consumption is rarely exposed directly. Furthermore, RPM alone is insufficient to estimate power accurately, as fan power depends on physical factors such as the fan diameter, pressure increase or air flow delived[77]:

$$P_{Fan} = d_p \cdot q = \frac{F}{A} \cdot \frac{V}{t} = \frac{F \cdot d}{t} \tag{2.17}$$

where p_d denotes total pressure increase of the fan (Pa or N/m^2), q denotes the air volume flow (m^3/s), F denotes force (N), A denotes fan area (m^2), V denotes volume (m^3), and t denotes time (seconds).

Based on observations, F is proportional with to the sqare of RPM. This can be combined with formula 2.17:

$$P_{Fan} = \frac{c \cdot RPM^2 \cdot d}{3600} \tag{2.18}$$

where for each individual fan, $c = \frac{3600 \cdot P_{Max}}{RPM_{Max}{}^2 \cdot d}$ remains constant.

Unfortunately, with the wide variety of fans in servers (especially with fan size restrictions due to server heights), these formulae are only helpful when paired with more detailed information on fan characteristics like Maximum RPM and Power.

While these can more reasonably be assumed, this remains a rough estimation at best.

2.5.6.3 Attribution of secondary component power consumption to individual workloads

Despite these measurement limitations, estimating the energy consumption of secondary components is often less critical for attributing energy to workloads. This is because components like fans, mainboards, and PSUs primarily support the operation of primary subsystems. Their power consumption scales with the activity level of CPU, memory, disk, and networking devices: more computation leads to higher heat dissipation, increased power delivery, and thus greater fan and PSU activity.

Consequently, if the total server power consumption is known—for example, via wall power monitoring or BMC/Redfish readings—the residual power (i.e., total power minus the sum of measured CPU, RAM, disk, and network power) can be reasonably attributed to power delivery and thermal management subsystems. This residual can then be proportionally distributed among active workloads based on the power consumption of the primary components they utilize. In this context, fine-grained modeling of secondary components becomes unnecessary for workload attribution, as their energy use correlates closely with that of the primary subsystems they support.

2.5.7 Issues with model-based power estimation techniques

This thesis pursues two inherently conflicting objectives: achieving high-resolution, accurate energy consumption measurements while simultaneously developing a solution that remains broadly applicable across heterogeneous server environments without requiring extensive manual calibration, or the manual input of complex device-specific information. Striking a balance between these goals is particularly challenging in the context of model-based energy estimation for devices that do not expose power telemtry data: Due to the wide variability among devices for a variety of factors, energy consumption models must necessarily abstract away much of the underlying complexity. Developing a model that is simultaneously fine-grained, highly accurate, and universally applicable across different technologies is, in practice, an unattainable goal.

As a result, any model integrated into a general-purpose energy estimation framework must err on the side of relative simplicity to preserve generality. While this approach diminishes the precision of device-specific energy attribution, it remains valuable for broader optimization tasks. For instance, autoscaling mechanisms, load balancers, and schedulers can still benefit significantly from approximate energy profiles. Likewise, cluster administrators aiming to improve energy efficiency holistically, or developers seeking to optimize their workloads, can gain useful directional insights even from coarse-grained models.

However, this simplicity imposes significant limitations for use cases that require device-specific energy optimization. General-purpose models are ill-suited for evaluating the energy efficiency of different device types, testing firmware-level adjustments, or validating the impact of power-saving features such as low-power states. In such scenarios, the model's abstraction may not just be insufficient, but actively misleading.

In the context of this thesis, this limitation is considered acceptable. The overarching objective is to facilitate scalable and portable energy estimation mechanisms for containerized environments, not to provide a diagnostic tool for hardware-level energy analysis. Nonetheless, this constraint should be kept in mind when interpreting the results and assessing their suitability for device-centric evaluation tasks.

2.6 Power Modeling based on Machine Learning Algorithms

In a taxonomy of power consumption modeling approaches, Lin et al[13] analyze various machine learning-based power models in current literature, categorizing them into supervised, unsupervised, and reinforcement learning. A detailed reiteration of this taxonomy (as well as a methodological overview of machine learning and neural networks) is omitted here.

In the context of this thesis, machine learning-based approaches are not considered for the following reasons:

- The author was unable to identify a promising and reliable approach that is sufficiently generalizable to function across a wide range of server configurations. Likewise, no component-level models were found that met these criteria. While it is certainly possible to train machine learning models to estimate energy consumption for specific server setups with high accuracy, the aim of this thesis is to provide generalizable estimation methods applicable to varied systems.

- Machine learning fundamentally relies on large datasets that are both highly accurate and granular, ideally matching the quality expectations of the resulting model. As discussed in previous sections, such high-quality training data is rarely available in the domain of fine-grained power measurement. When such datasets do exist, they typically reflect highly specific hardware and workload configurations, making them unsuitable for generalization. Although it would be theoretically possible to generate a large dataset by systematically benchmarking thousands of CPUs, memory modules, GPUs, storage, and network devices across millions of configurations, such an undertaking is not practically feasible. Furthermore, any such dataset would require ongoing expansion to remain representative of new hardware generations.

- Finally, many of the technical implementation details underlying key telemetry features remain proprietary or undocumented. A notable example is the RAPL interface, whose internal workings are not publicly disclosed. At the same time, existing RAPL metrics already offer abstracted energy readings suitable for direct integration into power estimation tools, eliminating the need for an intermediate machine learning-based step.

In theory, machine learning-based power estimation models hold significant promise and may one day be realized. Such models could leverage complex, nonlinear relationships between hardware components and workloads—relationships that are inherently difficult or even impossible to capture through traditional analytical models. The predictive and adaptive capabilities of machine learning offer the potential for highly accurate, fine-grained estimations across a broad range of configurations. However, as of today, the development of such a comprehensive and generalizable

model remains out of reach. Realizing this vision would require extensive collaboration between original equipment manufacturers, cloud providers, data center operators, and research institutions to generate, standardize, and share high-quality telemetry data across diverse hardware and workload scenarios. Given the role of cloud computing in the current economic landscape, where proprietary knowledge and performance optimization constitute a competitive advantage at the corporate, national, and geopolitical levels, such broad cooperation appears unlikely. Consequently, machine learning remains a promising but currently impractical direction for universal server power modeling.

2.7 Component-specific summaries

This section offers practical guidelines for measuring or estimating the energy consumption of individual hardware components. While earlier chapters focused on theory and research, the focus here is on implementation: what works best, what challenges to expect, and how to improve accuracy with minimal effort. For each component, the most accurate method is highlighted, along with alternatives and fallback options. Simple improvements like entering datasheet values or running calibration workloads are discussed where relevant.

2.7.1 CPU

The most accurate and widely adopted method for measuring CPU energy consumption is Intel's RAPL interface. It offers high temporal resolution, low overhead, and requires no external hardware. Among the available interfaces, `perf-events` is generally recommended due to its balance between usability, performance, and access control. The `powercap` interface offers simpler integration via `sysfs`, though with some limitations in domain structure and overflow handling. MSR access is discouraged due to complexity and privilege requirements. eBPF-based methods are powerful and used in advanced tools, but introduce high development complexity, kernel dependency, and lower portability.

On AMD systems, the `amd_energy` driver offers a RAPL-compatible interface, but it exposes fewer domains (e.g., no DRAM) and is generally less feature-rich than Intel's implementation.

Despite some limitations (such as non-atomic register updates, idle power inaccuracies, and the absence of timestamps), RAPL is considered sufficiently accurate for both research and production use. Its drawbacks can often be mitigated through careful sampling strategies (e.g., overflow-safe polling intervals, timestamp alignment), and correction techniques for overflow and measurement jitter.

ACPI, while historically relevant, does not expose real-time power data and is unsuitable for precise energy measurement. Although some theoretical estimation based on P-states is possible, it is coarse-grained and impractical for modern CPUs with dynamic frequency scaling.

If RAPL is unavailable, statistical models based on utilization, frequency, and other metrics may be used. However, these require prior calibration or hardware-specific profiling. Without access to idle or peak power values, such models become highly unreliable due to architectural variability. In such cases, estimation accuracy can be

modestly improved by inserting static power values from processor datasheets or using fixed coefficients for known CPU families.

2.7.1.1 Container-level implications

RAPL's granularity and domain separation make it suitable for correlating CPU energy usage with container activity, allowing for reasonably accurate attribution when combined with CPU usage metrics (e.g., cgroup CPU accounting or eBPF). In contrast, model-based estimations or ACPI-derived values are too coarse and lack temporal resolution, limiting their use to static or linear power distribution based on workload share, insufficient for fine-grained or bursty container workloads.

2.7.2 Memory

Memory power consumption can be measured using the DRAM domain exposed by Intel RAPL on supported server-grade processors. When available, this provides low-overhead, fine-grained energy telemetry integrated with other CPU domains. However, DRAM measurement accuracy depends heavily on processor architecture. It is generally reliable for Haswell-generation CPUs, but later architectures may exhibit a constant power offset or measurement inaccuracies due to off-DIMM voltage regulators and evolving memory subsystems.

If RAPL DRAM telemetry is unavailable or deemed unreliable, no equivalent in-band method exists. In such cases, estimation must rely on model-based approaches. Many models in literature attempt to correlate memory power with usage, memory access frequency, or cache behavior, but they are not generalizable across systems. Most require prior calibration using known idle and peak memory power figures, which are rarely available in practice. Without these, estimation accuracy remains low. Manual insertion of idle and active power values from vendor datasheets can slightly improve results, but still yields only coarse-grained estimates.

2.7.2.1 Container-level implications

The RAPL DRAM domain, when accurate, allows correlation between energy consumption and workload-level memory metrics such as usage or memory bandwidth. This enables container-level attribution if per-container memory activity is available. Without RAPL, model-based estimates only support static or proportional energy attribution based on usage share, which is insufficient for capturing the energy impact of memory-intensive or bursty workloads.

2.7.3 GPU

Accurate GPU power measurement remains a challenge in containerized environments. The most accessible solution is NVIDIA's NVML interface (e.g., via nvidia-smi), which exposes power metrics through on-board sensors. While widely used, NVML suffers from sampling delays, averaging artifacts, and limited temporal resolution—especially during transient workloads. Nevertheless, it offers acceptable accuracy for steady-state measurements and is supported across many data center deployments.

Alternative tools, such as AccelWattch and FinGraV, provide finer temporal granularity and more precise modeling, but are either architecture-specific or tightly

coupled to particular hardware (e.g., AMD MI300X). Hardware-based solutions like PowerSensor3 achieve excellent accuracy at high sampling rates but are cost-prohibitive and impractical for large-scale deployment. No general-purpose, software-only solution currently matches the accuracy and portability of CPU-side tools like RAPL.

2.7.3.1 Container-level implications

GPU power attribution in Kubernetes is limited by the granularity and accuracy of current tools. While NVML can be queried from within containers or sidecars, it does not natively support multi-tenant attribution, and virtualization layers (e.g., vGPU, MIG) complicate per-container visibility. Accurate, container-level GPU energy tracking remains an open problem, requiring either architectural integration (e.g., with MIG-aware scheduling) or improved temporal sampling. As such, GPU measurements are currently only viable for coarse-grained, workload-level profiling—not fine-grained container energy attribution.

2.7.4 Storage devices

Storage device energy consumption is typically estimated rather than measured. Unlike CPUs or GPUs, storage devices lack onboard power sensors, and BMC-based per-device readings are generally unavailable, especially when using backplanes, RAID controllers, or SATA interfaces. Consequently, power usage is inferred from device metrics and modeled behavior.

Telemetry is only available for specific device types. For NVMe drives, `nvme-cli` exposes detailed metrics such as supported power states, current power state, idle/active power ratings, and temperature. However, these are not available for SATA SSDs or HDDs. `smartctl` provides vendor-specific SMART data (e.g., temperature, power-on hours, wear) if available, but energy-related insights are limited. Standard Linux tools (`iostat`, `sar`, `/proc/diskstats`, etc.) expose generic performance counters such as IOPS, throughput, queue length, and utilization, which can support rough estimation.

Various model-based approaches estimate power using activity-based metrics (e.g., read/write rates or interface speed), often requiring idle and active power values from datasheets. These models are only accurate when tailored to specific hardware. No general-purpose estimator exists for unknown or heterogeneous storage types without prior calibration.

2.7.4.1 Container-level implications

Because disks are shared resources, and per-container telemetry is unavailable, energy attribution must rely on proportional estimation using observable metrics like I/O volume or latency. This approach works for long-lived workloads but lacks the granularity to capture energy dynamics of bursty or short-lived container activity. Accurate container-level attribution remains infeasible for SATA SSDs and HDDs and is only marginally better for NVMe devices, assuming access to detailed device metrics.

2.7.5 Network devices

NIC power consumption cannot be measured directly via software. Although many cards support PCIe power states (e.g. D0–D3), these states only approximately correlate with actual power draw and are not sufficient for energy estimation. Furthermore, NICs lack onboard power sensors, and BMC-based per-device readings are generally unavailable. As such, NIC energy consumption must be estimated using model-based approaches.

Various research models estimate NIC power using idle and dynamic components. The most promising approach, proposed by Baneshi et al., linearly scales NIC power with bandwidth utilization, assuming idle and maximum power values are known. While earlier models correlate energy use with channel utilization or throughput, they often oversimplify or lack generalizability. Real-world measurements show minimal power variation between idle and active states (often <1W difference), with idle power dominating overall NIC energy use. Estimates can be improved slightly by incorporating known idle and peak wattage from datasheets, but generalization across different NICs remains unreliable due to architectural and vendor variability. In the absence of these values, the only remaining option is to guess these values based on NIC PHY medium.

Telemetry support is limited: tools like `ethtool` expose link speed and status, but do not report power. No standard Linux tool provides direct NIC energy metrics, and throughput-based estimators must rely on indirect metrics like bytes transmitted per interval.

2.7.5.1 Container-level implications

Because NICs are shared across containers and lack per-container telemetry, only indirect attribution is possible. Energy consumption can be distributed proportionally based on container-level bandwidth usage (e.g., via cgroup network statistics), assuming idle and peak NIC power are known. However, the minimal dynamic variation in NIC power limits the usefulness of fine-grained attribution. In practice, NIC power is best modeled as a mostly static overhead, with marginal gains from utilization-based scaling.

2.7.6 Other Devices

Secondary components such as the motherboard, PSU, and fans contribute a non-trivial share to total server power consumption but are rarely modeled with precision. These devices typically lack direct power telemetry, and their energy use is either approximated or inferred indirectly.

PSU losses can be estimated from efficiency ratings if total input or output power is known. Fan power is difficult to measure and depends on physical factors like airflow and pressure; at best, it can be roughly estimated using RPM and vendor data. The motherboard and onboard controllers (e.g., voltage regulators, chipset) are usually modeled as part of residual power.

2.7.6.1 Best-practice approach

If system-level power data is available (e.g. via IPMI or Redfish), the difference between total server power and known component estimates can be treated as residual

power. This residual can be linearly distributed across containers based on the finer-grained power estimation of the CPU, assuming secondary device power scales with primary component power consumption.

2.7.6.2 Container-level implications

Because these components do not map directly to container usage, their energy must be attributed indirectly. Linear distribution based on known, container-attributed metrics (e.g., CPU time or workload duration) is a practical, though imprecise, fall-back for ensuring full power accounting in containerized environments.

Chapter 3

Attributing Power Consumption to Containerized Workloads

3.1 Introduction and Context

While the previous chapter focused on system-level and component-level power measurement and estimation, this chapter shifts focus to an equally complex task: attributing measured server power consumption to the individual containers or workloads responsible for it.

Attributing energy consumption in this context is inherently difficult due to multi-tenant, multi-layered workloads across multiple CPU cores and devices, as well as temporal granularty mismatch issues. Consequently, direct one-to-one mapping of energy consumption to workload is generally not possible.

Nonetheless, various techniques have emerged to approach this problem. The goal is to create an accurate and fair approximation of how much energy a given container or process is responsible for at any point in time. This chapter provides a conceptual foundation for these techniques. The subsequent Chapter **??** will examine how selected tools implement these ideas in practice. While some implementation aspects will be referenced for illustration, this chapter is focused on general methodologies, not tool-specific behavior.

3.2 Power Attribution Methodology

3.2.1 The Central Idea Behind Power Attribution

At its core, the concept of power attribution is simple: a task should be held accountable for the energy consumed by the resources it actively uses. If a task occupies the CPU for a given period, it is attributed the energy consumed by the CPU during that time. By summing the energy usage of all tasks belonging to a container, one can estimate the total energy consumption of that container. Since energy is the integral of power over time, the average power consumption of a container can be calculated by dividing its attributed energy by the total duration of interest. Depending on the use case, either energy (in joules) or power (in watts) may provide more meaningful insight. Energy is often used to quantify cost or carbon footprint, while power helps identify peak loads and inefficiencies.

While this model appears intuitive, its implementation in real systems is far from trivial. One major complication stems from the intricacies of multitasking on modern systems, which is discussed in subsection 3.2.2. Subsection 3.2.3 examines and compares different utilization tracking mechanisms, in Linux and Kubernetes. A a result of the fine-grained temporal control of multitasking, another major challenge is temporal granularity. Power consumption is typically sampled at much coarser intervals than kernel resource usage statistics. These mismatched update rates and resolutions must be reconciled to build meaningful correlations. This issue is elaborated in subsection 3.2.4.

Consequently, power attribution becomes a complex algorithmic process, involving summation, weighting, and interpolation across multiple metrics. It must strike a balance between data availability and estimation accuracy. A perfectly accurate system is not feasible, especially in heterogeneous or production-grade environments. Limitations and accuracy trade-offs are further discussed in subsection 3.2.5.

Real-world systems also introduce additional complications such as idle power, shared background processes, or missing hardware counters. These edge cases and practical constraints are addressed in section **??**. Finally, section 3.3 discusses the different philosophies of different attribution models to take account for different key demographics.

Despite these difficulties, power attribution serves a critical role in understanding container behavior. If applied consistently across all containers and system resources, it can uncover the dynamic patterns of energy usage within a server. This insight forms a foundational building block for cluster-level energy optimization. Administrators or automated systems can use this data to analyze the effect of configuration changes, improve workload scheduling, or optimize performance-per-watt, whether during runtime or post-execution.

3.2.2 A short recap of Linux Multitasking and Execution Units

Linux is a multitasking operating system that enables multiple programs to run concurrently by managing how processor time is divided among tasks. This capability is central to container-based computing and directly impacts how workload activity is linked to energy consumption.

Multitasking in Linux operates on two levels: time-sharing on a single core and true parallel execution across multiple cores. On a single-core system, the kernel scheduler rapidly switches between tasks by allocating short time slices, creating the illusion of parallelism. On multi-core systems, tasks can run simultaneously on different cores, increasing throughput but also complicating the task of correlating resource usage with measured power consumption.

At the kernel level, the smallest unit of execution is a *task*. This term covers both user-space processes and threads, which the kernel treats uniformly in terms of scheduling and resource accounting. Each task is represented by a `task_struct`, which tracks its state, scheduling data, and resource usage.

A *process* is typically a task with its own address space. Threads, by contrast, share memory with their parent process but are scheduled independently. As a result,

a multi-threaded program or container may generate several concurrent tasks, potentially running across multiple cores. These tasks are indistinguishable from processes in kernel metrics, which complicates aggregation unless care is taken to associate related threads correctly.

In containerized environments, tasks belonging to the same container are grouped using Linux control groups (cgroups) and namespaces. These mechanisms allow the kernel to apply limits and collect resource usage statistics at the container level, making them central to energy attribution in Kubernetes-based systems.

3.2.3 Resource Utilization Tracking in Linux and Kubernetes

In modern Linux-based systems, particularly within Kubernetes environments, multiple methods exist to track resource utilization [82–87]. These methods vary significantly in terms of temporal granularity, scope, and origin. While they often expose overlapping information, their internal mechanisms differ, leading to trade-offs in precision, resolution, and suitability for certain use cases such as energy attribution.

3.2.3.1 CPU Utilization Tracking

- **/proc/stat:** A global, cumulative snapshot of CPU activity since boot. It records jiffies spent in user, system, idle, and iowait modes. Temporal resolution is high, but data is coarse and not process- or cgroup-specific.
- **/proc/<pid>:** Provides per-task CPU statistics including time spent in user and kernel mode. Offers fine-grained tracking on a per-process level but must be polled manually at high frequency to detect short-lived changes. Contains information about task container and namespace.
- **cgroups:** Tracks cumulative CPU usage in nanoseconds per cgroup. In Kubernetes, each container runs in its own cgroup, enabling container-level usage attribution. Granularity is high, and this is a foundational metric for tools like KEPLER and cAdvisor.
- **eBPF:** eBPF enables near-real-time tracking of per-task CPU cycles and execution and allows correlation of resource usage to kernel events (e.g. context switches). It is especially valuable when precise attribution to short-lived tasks or containers is required.
- **Hybrid tools:** Many tools provide aggregated metrics and statistics based on the afforementioned methods. While user-friendly, these are usually provide lower temporal precision, but may be useful in some instances. **cAdvisor:** collects and aggregates CPU usage per container by reading from cgroups. While widely used, its default update interval is coarse. Data is sampled and averaged, which limits its use in high-resolution analysis. **metrics-server (metrics.k8s.io):** exposes aggregated CPU usage via the Kubernetes API. It pulls metrics from Kubelet (which relies on cAdvisor) and is updated every 15 seconds. Not suitable for precise or historical analysis.

3.2.3.2 Memory Utilization Tracking

- **/proc/meminfo:** Provides a system-wide view of memory usage but lacks per-task or per-container resolution.
- **/proc/<pid>/status:** Exposes memory-related counters for each process (e.g., RSS, PSS, virtual set size). Temporal granularity is fine but requires frequent polling.
- **cgroups (memory):** Records memory usage for groups of processes. 'memory.usage_in_bytes' shows current memory usage per cgroup, allowing container-level tracking. High granularity and reliability, frequently used in both monitoring and enforcement.
- **cAdvisor and metrics-server:** As with CPU, memory stats are aggregated from cgroup data. These APIs offer lower resolution and no historical data.

3.2.3.3 Disk I/O Utilization Tracking

- **/proc/<pid>/io:** Tracks per-process I/O activity (bytes read/written, syscall counts). Useful for attributing I/O behavior, but coarse in how it correlates to actual disk access timing.
- **cgroups-v1 (blkio) / cgroups-v2 (io):** Reports aggregated I/O stats per cgroup (bytes, ops, per-device). Allows container-level attribution. Granularity depends on polling rate and support by the underlying I/O subsystem.
- **eBPF (tracepoints, kprobes):** Enables real-time tracing of block I/O syscalls, bio submission, and completion.

3.2.3.4 Network I/O Utilization Tracking

- **/proc/net/dev:** Shows network statistics per interface. Updated continuously, but lacks process/container granularity.
- **cgroups-v1 (net_cls, net_prio):** Used to mark packets with cgroup IDs, enabling traffic shaping and classification. Attribution is possible if paired with packet monitoring tools, but rarely used directly. While there is not direct equivalent in **cgroups-v2**, support was added in `iptables` to allow BPF filters that hook on cgroup v2 pathnames to allow control of network traffic on a per-cgroup basis.
- **eBPF:** Allows tracing of network activity at various points in the stack (packet ingress, egress, socket calls). Offers very high granularity and can attribute traffic to specific containers.

3.2.3.5 eBPF-based Collection of utilization Metrics

The extended Berkeley Packet Filter (eBPF) is a Linux kernel subsystem that allows the safe execution of user-defined programs within the kernel without modifying kernel source code or loading custom modules. Originally developed for low-level network packet filtering, eBPF has evolved into a general-purpose observability

framework that can trace and monitor system events with high precision and minimal overhead. eBPF can be used to dynamically attach probes to kernel events such as context switches, system calls, I/O events, and tracepoints. In the context of system monitoring, this enables the collection of fine-grained utilization metrics, including CPU usage per process, memory allocations and I/O activity without modifying the monitored application. These probes run within the kernel and populate BPF maps, which can then be accessed by user-space tools to aggregate or export metrics.

Compared to traditional monitoring approaches such as reading from `/proc`, eBPF offers several key advantages. First, it supports high temporal resolution, enabling near real-time tracking of events. Second, it avoids the need for intrusive instrumentation or static tracepoints, making it suitable for black-box applications. Finally, its dynamic and event-driven nature reduces performance overhead by eliminating polling. As a consequence, eBPF has often been used for utilization monitoring: KEPLER uses eBPF to monitor CPU cycles and task scheduling events, enabling accurate attribution of resource usage to short-lived or highly dynamic workloads. It complements cgroup and perf-based metrics, allowing power attribution models to track containers that would otherwise be indistinguishable using standard polling-based methods.

As demonstrated by Cassagnes et al. [88], eBPF currently represents the best practice for non-intrusive, low-overhead, and high-resolution utilization monitoring on Linux systems. Its ability to gather container- or process-level metrics in production environments makes it uniquely well-suited for accurate correlation with system-wide power measurements.

3.2.3.6 Performance Counters and `perf`-based Monitoring

Modern processors expose hardware-level performance counters (PMCs) that can be used to obtain precise measurements of internal execution characteristics. These counters are accessible via tools such as `perf`, and include metrics such as retired instructions, CPU cycles, cache misses, branch mispredictions, and stalled cycles. Unlike traditional utilization metrics, which measure time spent in various CPU states, PMCs offer insight into how effectively the processor is executing instructions.

A particularly relevant metric is *instructions per cycle* (IPC), which quantifies how much useful work is being done per clock cycle. An IPC close to the CPU's architectural maximum indicates efficient execution, while lower values often signal bottlenecks such as memory stalls. As shown by Gregg[89], a low IPC may reveal that the processor is heavily stalled, even when CPU utilization appears high.

These metrics provide a powerful alternative for workload analysis and energy estimation. For instance, instruction counts can be used to normalize energy usage per task, enabling attribution models that go beyond time-based utilization.

However, access to PMCs is not always guaranteed. In virtualized environments and some container runtimes, performance counters may be inaccessible or imprecise due to hypervisor restrictions. Moreover, interpreting raw PMC values requires architectural knowledge and hardware-specific calibration.

Source	Granularity	Scope	Notes
/proc/stat	Medium	Global	Jiffy-based, coarse
/proc/<pid>/stat	High	Per-process	Fine-grained, must poll manually
cgroups	High	Per-cgroup	Foundation for container metrics
cAdvisor	Medium-Low	Per-container	Aggregated from cgroups, limited rate
eBPF	Very High	Per-task, system-wide	Real-time, customizable, low overhead
perf/PMCs	Very High	Per-task, core-level	Tracks cycles, instructions, stalls

TABLE 3.1: Comparison of resource usage tracking mechanisms

3.2.3.7 Comparative Summary

3.2.4 Temporal Granularity and Measurement Resolution

To correlate CPU usage with power consumption, time must be considered at an appropriate granularity. The Linux kernel tracks CPU usage at the level of scheduler ticks, which are driven by a system-wide timer interrupt configured via `CONFIG_HZ`. Typical values range from 250 to 1000 Hz, meaning time slices of 4 to 1 milliseconds, respectively. These ticks, or *jiffies*, represent the smallest scheduling time unit and are used to increment counters such as `utime` and `stime` for each task.

More modern interfaces (such as cgroup v2's `cpu.stat`) provide higher-resolution timestamps, often in nanoseconds, depending on the kernel version and configuration.

In contrast, power measurement tools generally operate at coarser time resolutions. Intel RAPL, for example, may expose updates every few milliseconds to hundreds of milliseconds, while BMC- or IPMI-based readings typically update once per second or slower. As a result, power attribution techniques must reconcile the high-frequency task activity data with lower-frequency power measurements, often through aggregation or interpolation over common time intervals.

A clear understanding of these execution and timing units is essential for building reliable power attribution models. These concepts underpin all subsequent steps, including metric fusion, resource accounting, and workload-level aggregation.

3.2.5 Challenges

System monitoring and the attribution of power metrics based on system (and component) utilization metics introduces several challenges that need to be addressed by a power attribution methodology. Some of these represent natural tradeoffs that an architect needs to be aware of, while others pose issues that simply cannot be circumvented without major issues that cannot be solved with a suitable architecture.

3.2.5.1 Temporal Granularity and Synchronization

A central challenge in power attribution is the mismatch in temporal granularity between system and power metrics. High-resolution sources, such as eBPF-based

monitoring, can distinguish variations within individual CPU time slices. In contrast, coarse-grained power metrics (such as IPMI) often update only once per second, rendering them unable to reflect fine-grained container activity. Metrics like RAPL fall in between, typically sampled at up to 1000 Hz but practically stable at around 50 Hz. Model-based estimators may match the granularity of their input metrics or, in simpler cases, use time-based assumptions with theoretically unlimited granularity.

These disparities make straightforward correlation difficult. While coarse metrics like IPMI provide broad system power data (including components invisible to fine-grained tools) they should not be interpolated to finer time scales, as doing so introduces artificial detail and potential misattribution. Instead, they are best treated as low-frequency anchors to validate or constrain high-resolution estimates. For example, summed RAPL readings can be compared to IPMI over aligned intervals, though their differing measurement scopes add complexity.

Another complication is temporal skew. Even metrics with similar frequencies are rarely sampled simultaneously, and some introduce unknown or variable delays. This misalignment creates ambiguity between observed utilization and corresponding power draw, particularly for short-lived or rapidly changing workloads. Naïve smoothing may reduce noise but also obscures meaningful transient behavior.

Effective attribution therefore requires more than just aligning timestamps. It demands awareness of each metric's origin, behavior, and limitations, and careful coordination to avoid erroneous correlations and preserve meaningful detail.

3.2.5.2 Challenges in CPU Metric Interpretation

CPU utilization is one of the most accessible and commonly used metrics to quantify processing activity on modern systems. It is widely reported by system monitoring tools such as `top`, `htop`, and cloud APIs, and is frequently used in both performance diagnostics and energy attribution models. However, despite its ubiquity, the interpretation of CPU utilization is far from straightforward, and in many contexts, it is misleading.

At its core, CPU utilization is a time-based metric that represents the proportion of time a CPU spends executing non-idle tasks. In Linux, this value is computed from counters in `/proc/stat` and reported in units of "jiffies". It distinguishes between various states (user, system, idle, I/O wait, interrupts) but ultimately expresses how long the CPU was busy, not how much useful work it performed[90].

A fundamental limitation is that CPU utilization conflates time with effort. Not all CPU time is equally productive: some cycles may execute complex, compute-intensive instructions, while others may stall waiting for memory I/O. Modern CPUs are frequently memory-bound due to the growing performance gap between processor speed and DRAM latency. As a result, a high CPU utilization value may indicate that the processor was merely stalled, not that it was the performance bottleneck[89].

These nuances have direct consequences for energy attribution. When energy models allocate power proportionally to CPU utilization, they assume a linear relationship between time and energy. However, power consumption depends heavily on

the instruction mix, CPU frequency scaling, Turbo Boost, and simultaneous multithreading. In such environments, identical utilization values across different processes or intervals may reflect vastly different energy profiles.

A more accurate alternative is to use hardware performance counters (PMCs), which track low-level metrics such as instructions retired, cache misses, and stalled cycles. For example, the "instructions per cycle" (IPC) value provides insight into how effectively the CPU executes work during its active time. An IPC significantly below the processor's theoretical maximum often indicates a memory-bound workload, while high IPC values suggest instruction-bound behavior. Tools like `perf` or `tiptop` can expose such metrics, though their use may be restricted in virtualized environments.

In summary, CPU utilization should be treated with caution, especially in the context of energy-aware scheduling and workload attribution. As Cockcroft already argued in 2006, utilization as a metric is fundamentally broken[91]. Practitioners are advised to:

- Avoid assuming a linear relationship between CPU utilization and power consumption.
- Consider supplementing utilization metrics with performance counters (e.g., IPC, cycles, instructions) when available.
- Be mindful of the measurement interval and sampling effects in tools like *Scaphandre*.
- In energy models, explicitly account for idle power, and avoid assigning it solely to active processes.
- Prefer instruction-based metrics for finer granularity and better correlation with energy use.

3.2.5.3 Availability of Metrics

The availability of system and power metrics varies widely between platforms. While some systems offer high-resolution data, others may only expose coarse values or lack direct power data entirely. An effective attribution system should dynamically adapt to the metrics available, incorporating new sources (such as wall power meters) as they are added.

Ideally, such a system would also communicate the tradeoffs involved, indicating how metric availability affects accuracy and granularity. This transparency ensures that attribution results are interpreted with appropriate context and helps guide improvements in monitoring fidelity.

Attribution in Multi-Tenant and Shared Environments

In multi-tenant systems, not all resources can be cleanly partitioned or measured with sufficient precision for container-level attribution. Some components are inherently shared, and their energy use cannot be isolated to individual workloads. Additionally, system-wide energy consumers like power supplies, cooling fans, and idle background services contribute to total power draw but are not tied to any specific container. Attribution models must account for these shared and unaccountable

energy domains. Addressing these concerns requires careful modeling and philosophical choices about how to treat unassigned energy, which are further discussed in section 3.3.

Measurement Overhead

All monitoring systems inherently introduce some degree of overhead. While modern tools such as eBPF are designed to minimize this impact, they still consume CPU cycles and memory bandwidth. Lightweight tools can reduce overhead without sacrificing data quality, but complete elimination is not possible.

Notably, the cost of monitoring increases with temporal resolution. Fine-grained metrics require higher sampling rates, more frequent data transfers, and additional processing effort. Since container-level power metrics typically do not require subsecond resolution, it is essential that high-resolution analysis and correlation occur as early as possible in the data pipeline. By aggregating and attributing power consumption close to the source, downstream systems can operate on compact, coarse-grained results, reducing both computational and storage overhead while preserving attribution accuracy.

Support for Evolving Models

As hardware platforms and research in power estimation continue to evolve, new measurement interfaces and modeling approaches are regularly introduced. These may offer improved accuracy, reduced overhead, or better coverage of previously unobservable components. To remain relevant and effective, container-level power attribution systems must be designed with adaptability in mind. A modular architecture enables the integration of new data sources or estimation models without reengineering the entire system. This flexibility ensures long-term maintainability and allows the system to benefit from ongoing advancements in energy modeling and monitoring infrastructure.

3.3 Attribution Philosophies

Attributing server power consumption to individual containers requires decisions that go beyond data collection. Some components are inherently shared, some workloads contribute system-level overhead, and some energy is consumed by idle hardware. The way these factors are treated reflects the underlying attribution philosophy. This section outlines three main approaches, each suitable for different goals and users.

3.3.1 Container-Centric Attribution

This model attributes energy solely based on the direct activity of containers, ignoring system services and shared infrastructure. Remaining resources are pooled and can be declared as system resources. This means that a container is not accountable for its own orchestration and energy wasted through system idling.

- **Advantages:** Isolates workload impact; consistent across system loads.
- **Limitations:** Understates real-world cost; excludes orchestration and idling overhead.

- **Suitable for:** Developers optimizing containerized applications.

Notably, container-centric attribution places a strong emphasis on individual containers and their respective energy consumption, striving to maintain relative consistency irrespective of overall cluster activity. While such granular insights can be valuable to developers, container-centric attribution typically does not represent the primary practical use case of an energy monitoring system for Kubernetes containers. This is largely due to the container isolation principle, which usually restricts detailed visibility into broader system dynamics. Additionally, container-level optimization is often more effectively achieved through simpler CPU and memory metrics readily accessible via the container's own `/proc` filesystem. Hence, although technically feasible, container-centric energy attribution often remains primarily a theoretical or research-oriented concept rather than a widely implemented practical approach.

3.3.2 Shared-Cost Attribution

Here, all power consumption is distributed across active containers, either equally or proportionally to usage. As a consequence, a container is accountable for its own orchestration, it's share of OS resources and even energy wasted through system idling.

- **Advantages:** More accurately reflects total system cost
- **Limitations:** Attribution fluctuates with container count; depends on arbitrary distribution logic.
- **Suitable for:** Cluster operators optimizing cluster orchestration.

3.3.3 Explicit Residual Modeling

Beyond the container-centric and shared-cost attribution models lies a more nuanced approach that explicitly incorporates the efficiency characteristics of server hardware. In this model, total power consumption is divided not only among containers and system services but also includes separate terms for idle power and high-utilization overhead. Idle power represents the baseline energy required to keep the system operational, even when no meaningful work is being performed. However, this value is difficult to isolate, as it often overlaps with low-level system activity such as kernel threads, background daemons, or monitoring agents.

At the other end of the spectrum, when utilization approaches system limits, energy efficiency typically degrades due to resource contention, frequent context switching, and thermal throttling[92]. These effects increase energy consumption without proportional performance gains. To account for these dynamics, this model introduces two residual domains (*idle waste* and *efficiency overhead*) which reflect conditions not attributable to any specific container. While this model is more complex, it enables more accurate assessment of workload behavior, infrastructure utilization, and waste, making it particularly valuable for research, performance engineering, and sustainability analysis.

Challenges in Measuring Residuals. Despite its advantages, implementing this model is non-trivial due to the difficulty of distinguishing idle consumption and overhead effects from general system resource usage:

Idle power estimation

- **Shared background activity:** Even in idle states, kernel tasks and system services introduce minimal but nonzero load, making it hard to define a "pure" idle baseline.
- **C-state transitions:** CPUs may briefly exit low-power states due to timers or interrupts, causing fluctuations even during apparent idleness.
- **Isolation difficulty:** In production or multi-tenant environments, isolating a server to a truly idle state is often impractical.

High-utilization overhead

- **Lack of a clear baseline:** There is no standard definition of "ideal" energy usage at full utilization, complicating quantification of overhead.
- **Architecture-specific behavior:** Overheads from cache contention, memory stalls, or I/O bottlenecks depend heavily on the workload and hardware architecture.

Due to their complexity and variability, high-utilization overhead effects are excluded from the scope of this thesis. This is a minor limitation, as assigning this energy to general *system* consumption remains a valid and conservative approach.

Practical Approach to Idle Estimation. In practice, idle consumption can be estimated pragmatically by recording power usage while no user workload is running. While this conflates pure idle consumption with background system activity, the trade-off is acceptable given its simplicity and reproducibility.

The resulting hybrid model separates power into three categories:

$$P_{\text{total}} = \sum P_{\text{container}} + P_{\text{system}} + P_{\text{idle}} \tag{3.1}$$

Residual power not attributed to container workloads is explicitly labeled as *system* or *idle* consumption. (In some literature, the term *static* is used in place of *idle*.)

- **Advantages:** Transparent; enables both container-level and infrastructure-level analysis.
- **Limitations:** Requires high-quality telemetry; boundaries between idle and system power are inherently fuzzy.
- **Best suited for:** Research, cluster optimization, and sustainability reporting.

In real-world scenarios, this model provides cluster operators with the foundation for quantitative infrastructure efficiency analysis. At the same time, developers benefit from a consistent, workload-centric power metric that reflects the true resource cost of their container, independent of the activity of co-located workloads.

3.3.4 Understanding the Distinction Between CPU Idling and Process Idling

A CPU is considered **idle** when it has no runnable tasks. In this case, the Linux scheduler runs a special task called the *idle task* (PID 0), and the processor may enter a low-power idle state to save energy. The time spent in this state is what is reported as CPU idle time. The *idle task* is not shown in `/proc` and similar interfaces because it is only internally used by the scheduler, and not a regular process.

A process, on the other hand, does not truly idle in kernel terms. When a process is not using the CPU (because it is waiting for I/O, a timer, or another event) it is in a *sleeping* or *blocked* state. Although it may appear inactive, it is still managed by the scheduler and may resume execution when its blocking condition is resolved.

The key distinction is that **only CPUs idle** in the kernel's formal sense. A CPU idles when it has no work to do, while a process never truly idles: it either runs, waits, or is terminated.

Chapter 4

Existing Approaches to Container Energy Consumption

xxxxxxxxxxxxxxxxxxxxxxxxxxx

4.1 Introduction

4.2 Non-container-focused Energy Monitoring Tools

4.2.1 Server-Level Energy Monitoring

While not directly translatable to container-level energy monitoring, server-level energy consumption is an important aspect of it. Scientific works and tools in this domain generally don't provide the temporal resolution required for container-level energy monitoring.

4.2.1.1 Kavanagh and Djemame: Energy Modeling via IPMI and RAPL Calibration

Overview and Architecture Kavanagh and Djemame[16] present their findings on combining IPMI and RAPL (interface unspecified) data to estimate server energy consumption, achieving improved accuracy through calibration with an external server-level Watt meter. For calibration, they induce artificial CPU workloads and rely on CPU utilization metrics with 1-minute averaging windows, necessitating extended calibration intervals to obtain stable readings. While the resulting model is tailored to their specific hardware and not generally portable, their work provides valuable insights into the complementary use of IPMI and RAPL. The authors recognize that the respective limitations of these tools (RAPL's partial scope and IPMI's low resolution) can be mitigated when used in combination.

Attribution Method and Scope Although the model operates at the physical host level, it supports attribution to VMs or applications using CPU-utilization-based proportional allocation. Several allocation rules are proposed, including utilization ratio, adjusted idle sharing, and equal distribution. However, no container-level attribution is attempted, and runtime flexibility is limited due to the static nature of the calibration.

Validation and Limitations With their Watt-meter-calibrated model using segmented linear regression, the authors report an average error of just -0.17%. More relevant to practical application, they also construct a model based solely on IPMI and RAPL(calibrated via Watt meter data) which achieves a reduced error of -5.58%, compared to -15.75% without calibration. Limitations of their approach include the need for controlled, synthetic workloads, coarse-grained sensor input, and the assumption of relatively stable system conditions during calibration.

Key Contributions

- **Hybrid use of IPMI and RAPL is analyzed**, showing that these tools compensate for each other's limitations. RAPL underestimates total system power, while IPMI captures more components but at lower resolution.
- IPMI accuracy is significantly improved through external Watt meter calibration.
- The authors provide practical calibration guidelines:
  - Use long, static workload plateaus to align with averaging windows and reduce synchronization complexity.
  - Discard initial and final measurement intervals to avoid transient noise and averaging artifacts.
  - Ensure calibration workloads exceed the IPMI averaging window to capture valid steady-state values.

Relevance to Proposed Architecture This work informs the proposed architecture by demonstrating how combining RAPL and IPMI can yield more accurate system-level power estimation. The use of plateau-based calibration and composite data models is especially applicable. However, the lack of container-level granularity, reliance on offline calibration, and limited attribution scope underscore the need for more dynamic, fine-grained, and container-aware approaches in Kubernetes-based environments.

4.2.1.2 CodeCarbon

CodeCarbon[93] is a Python package designed to estimate the carbon emissions of a program's execution. While its implementation is general-purpose, it is primarily aimed at machine learning workloads.

Overview and Architecture CodeCarbon estimates a workload's energy consumption by relying on RAPL *package-domain* CPU metrics via the `powercap` RAPL file system interface. A fix for the RAPL MSR overflow issue was implemented[94]. In the absence of RAPL support, it falls back to a simplified model based on the CPU's Thermal Design Power (TDP), obtained from an internal database, and combines it with CPU load metrics from `psutil`. For memory, a static power value is assumed based on the number and capacity of installed DIMMs. GPU power consumption is estimated via NVIDIA's NVML interface. The default measurement interval is 15 seconds, with the authors citing lightweight design as the primary motivation.

The component-level estimations are then aggregated and multiplied by a region-specific net carbon intensity (based on the local electricity grid's energy mix) to estimate the program's total CO_2 emissions. CodeCarbon is typically executed as a wrapper around code blocks, scripts, or Python processes.

Limitations There is no direct attribution of CPU activity to individual power metrics: CodeCarbon estimates energy use indirectly, based on the number of active cores and average CPU utilization, while making many assumptions that could be prevented. Combined with the relatively long measurement intervals, this results in background system processes also being attributed to the measured Python program. Consequently, CodeCarbon does not contribute directly to the goals of this thesis, which seeks fine-grained, container-level attribution.

However, the tool highlights several interesting secondary considerations. The integration of regional CO_2 intensity data is a valuable extension to conventional energy measurement and is well implemented. Additionally, the Python-based design offers high accessibility and ease of use, which may serve as inspiration for future developer-facing tools.

4.2.1.3 AI Power Meter

AI Power Meter[95] is a lightweight Python-based tool designed to monitor the energy consumption of machine learning workloads. It gathers power consumption data for the CPU and RAM via Intel RAPL using the `powercap` interface, and for the GPU via NVIDIA's NVML library. While the authors acknowledge that other system components (e.g., storage, network) also contribute to energy usage, these are not currently included and are considered an accepted limitation of the tool.

Unlike more advanced attribution tools, AI Power Meter does not distinguish between individual processes or workloads. Instead, it provides coarse-grained, system-level energy consumption measurements over time. In this respect, its scope is similar to *CodeCarbon*, focusing on ease of use and integration into ML pipelines rather than precise, per-process energy attribution. As such, while not directly applicable to container-level measurement or power attribution, AI Power Meter demonstrates the growing interest in accessible energy monitoring tools within the machine learning community.

4.2.2 Telemetry-Based Estimation Frameworks

4.2.2.1 PowerAPI Ecosystem[96] (PowerAPI, HWPC, SmartWatts)

PowerAPI[97] is an open-source middleware toolkit for assembling software-defined power meters that estimate real-time power consumption of software workloads. Developed as a generalized and modular framework, PowerAPI evolved alongside specific implementations such as *SmartWatts*, detailed in section 4.3.3. It allows power attribution at multiple granularity levels, including processes, threads, containers, and virtual machines. A distinctive strength of PowerAPI is the continuous self-calibration of its power models, enabling accurate real-time energy estimation under varying workloads and execution conditions. This makes PowerAPI particularly suited to heterogeneous computing infrastructures.

Overview and Architecture PowerAPI uses an actor-based model for modularity, enabling easy customization of its internal components with minimal coupling. It supports raw metric acquisition from diverse sensors (e.g., physical meters, processor interfaces, hardware counters, OS counters) and delivers power consumption data through various output channels (including files, network sockets, web interfaces, and visualization tools). As middleware, PowerAPI facilitates assembling power meters "*à la carte*" to accommodate specific user requirements and deployment scenarios.

Core Components

- **powerapi-core**: Middleware orchestrating real-time/post-mortem interactions between sensors and formulas. It defines the essential interfaces for sensor data ingestion and output channels (e.g., MongoDB, InfluxDB, CSV, socket, Prometheus), and includes built-in capabilities for data preprocessing, post-processing, and reporting.
- **hwpc-sensor**: A telemetry probe designed to gather low-level hardware performance counters (HWPCs), including instructions, cycles, and RAPL energy metrics. This sensor leverages *perf* and *cgroups-v2*, critical for fine-grained telemetry in containerized environments. It also provides detailed CPU performance state metrics via MSR events (`TSC`, `APERF`, `MPERF`).
- **SmartWatts-formula**[98]: A power model implementation (in Python) using HWPC data to estimate power consumption dynamically. It employs online linear regression provided by the Python *scikit-learn*[99] library, enabling accurate runtime learning of workload-specific power signatures. SmartWatts is further detailed in section 4.3.3.
- **SelfWatts-controller**: Dynamically selects hardware performance counters for software-defined power models, facilitating automatic configuration and unsupervised deployment in heterogeneous infrastructures. Currently, its development has stalled for several years, limiting its practical applicability.
- **pyRAPL**: A convenient Python wrapper around RAPL for CPU, DRAM, and iGPU energy metrics collection, providing easy access to hardware-based power data.

Relevance and Integration The modular and extensible architecture of PowerAPI positions it as a highly suitable foundation for further research and development of specialized power attribution tools. Researchers can readily extend or adapt its components to address evolving or niche requirements. However, its current implementation does not incorporate certain critical metrics, such as IPMI-based telemetry, which could limit its completeness in some practical deployment scenarios. Nonetheless, PowerAPI represents a significant advancement toward the creation of generalized, plug-and-play power models that operate without extensive manual calibration. This emphasis on practical deployability and general applicability highlights a key strength of the project and sets a clear direction for future research and development efforts in the domain of software-defined energy monitoring.

4.2.2.2 Green Metrics Tool

The *Green Metrics Tool* (GMT)[100] is an open-source framework designed to measure the energy consumption of containerized applications across various phases of the software lifecycle, including installation, boot, runtime, idle, and removal. It uses small, modular metric collectors to gather host-level energy and system data (e.g., CPU and DRAM energy via RAPL, IPMI power readings), and is orchestrated through declarative usage scenarios.

While GMT provides reproducible, lifecycle-aware measurements in controlled environments, it does *not* perform container-level or process-level energy attribution. The developers explicitly avoid splitting energy consumption across containers, citing the lack of reliable attribution models.

4.3 Container-Focused Energy Attribution Tools

4.3.1 Kepler

4.3.1.1 Overview and Goals

Kepler (*Kubernetes-based Efficient Power Level Exporter*)[101] is a modular, Kubernetes-native framework for monitoring, modeling, and estimating energy consumption in containerized environments. As the most prominent tool for container-level power estimation in Kubernetes, Kepler enables detailed observability of energy usage at the level of individual processes, containers, pods, and nodes[102].

Kepler integrates seamlessly with Kubernetes and Prometheus-based observability stacks. It supports both real-time energy metrics (e.g., RAPL, ACPI, NVML) and model-based estimation through trained regression models, making it applicable across a wide range of deployment environments, from bare-metal servers to virtual machines. Developed as an open-source CNCF project, Kepler's architecture is designed to be extensible, allowing researchers and practitioners to contribute new power models and adapt it to diverse system architectures.

4.3.1.2 Architecture and Metric Sources

Kepler's architecture consists of several interconnected components, with the core functionality centered around a privileged monitoring agent that runs on every node. While the framework supports model-based estimation for environments without hardware telemetry, this thesis focuses on the direct collection of real-time power and utilization metrics available in bare-metal deployments.

Deployment Models Kepler supports multiple deployment scenarios depending on the availability of energy sensors on the host system. In bare-metal environments, Kepler can directly collect power metrics using RAPL, ACPI, or Redfish/IPMI interfaces. This is the most accurate and relevant mode for the purpose of this thesis. In contrast, on virtual machines (VMs), where access to hardware counters or power interfaces is restricted, Kepler relies on trained regression models to estimate node-level energy consumption. A third, currently unimplemented deployment model proposes a passthrough mechanism where a host-level Kepler instance would expose power metrics to a nested Kepler instance inside the VM. These deployment models are visualized in figure 4.1.

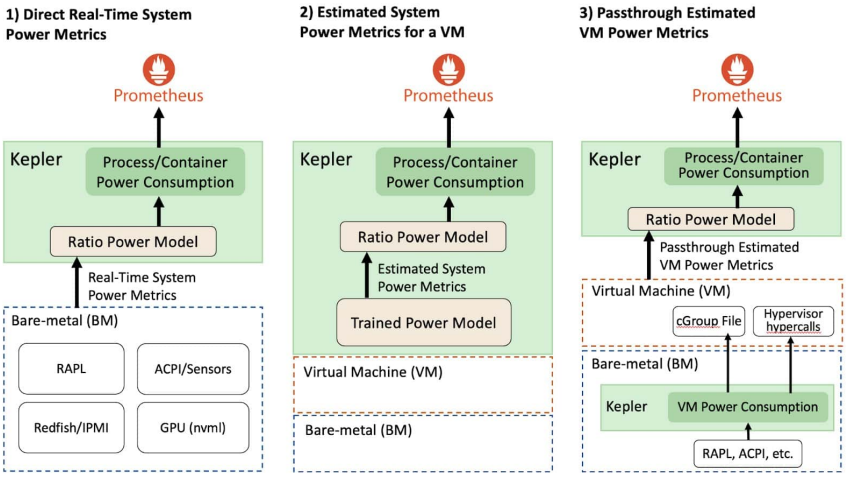


FIGURE 4.1: Kepler deployment models: direct power measurement on bare-metal, estimation on VMs, and the proposed passthrough model (currently not implemented)[103]

Kepler Agent and Exporter The core monitoring functionality is handled by the Kepler Agent, which is deployed as a privileged DaemonSet pod on each Kubernetes node. It collects energy and resource utilization metrics using a combination of eBPF instrumentation and hardware performance counters exposed via `perf_event_open`. A kprobe attached to the `finish_task_switch` kernel function enables accurate tracking of per-process context-switch activity. Container and pod attribution is performed after parsing the cgroup path from `/proc/<pid>/cgroup` and querying the Kubelet API for container metadata. The generated metrics are exported via a Prometheus-compatible endpoint for downstream processing and visualization. A generalized information flow is shon in figure 4.2.

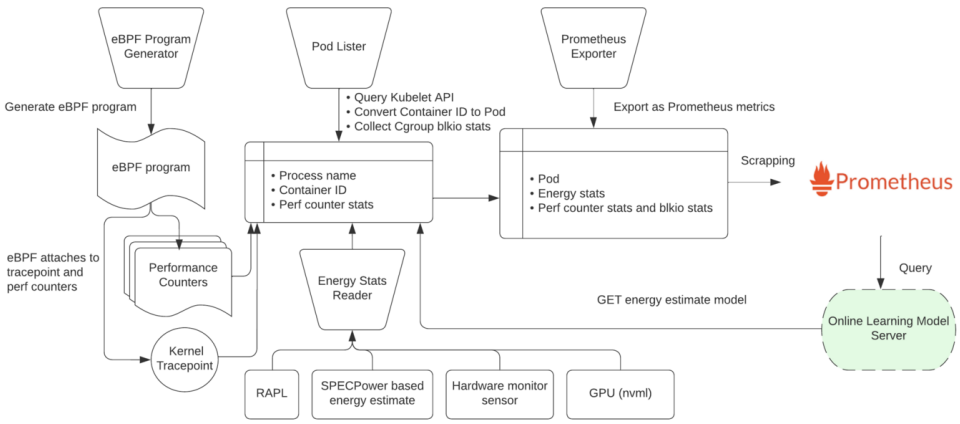


FIGURE 4.2: Simplified architecture of the Kepler monitoring agent and exporter components[103]

Resouce Utilization via eBPF-based Hardware/Software Counters To measure low-level CPU activity, Kepler uses the Linux syscall `perf_event_open` to configure hardware performance counters on each core. The following events are tracked:

- `PERF_COUNT_HW_CPU_CYCLES`: Total CPU cycles (affected by DVFS)
- `PERF_COUNT_HW_REF_CPU_CYCLES`: Frequency-independent cycles

- `PERF_COUNT_HW_INSTRUCTIONS`: Retired instructions
- `PERF_COUNT_HW_CACHE_MISSES`: Last-level cache misses

These counters are accessed via BPF perf event arrays. On each context switch, current counter values are sampled, and deltas are computed against previously stored values. These deltas represent the CPU activity of the process leaving the CPU and are stored in BPF maps for later aggregation.

In addition to hardware counters, Kepler collects several software-level metrics that are not natively exposed by the Linux kernel. These include CPU time, page cache activity, and interrupt handling statistics. Because these metrics are unavailable through standard interfaces, Kepler uses custom eBPF programs to infer them from kernel behavior.

Since version 0.7, Kepler has migrated to `libbpf` and uses a BTF-enabled kprobe to instrument the `sched_switch` function. This allows Kepler to safely extract process IDs and timing data without relying on fragile symbol offsets. On each context switch, Kepler records timestamps and uses them to increment the `CPUTime` counter, providing fine-grained accounting of CPU residency per process.

Other software counters include:

- `PageCacheHit`: Tracks read and write access to the page cache using eBPF programs attached to `mark_page_accessed` and `writeback_dirty_folio`.
- `IRQNetTX`, `IRQNetRX`, `IRQBlock`: Count the number of softirq events attributed to a process, using the `softirq_entry` tracepoint.

Each of these metrics is manually accumulated in eBPF maps keyed by process ID and periodically read by the user-space collector. This enhances energy attribution, especially in scenarios where hardware counters are insufficient or unavailable.

Node Component-level energy consumption via RAPL Kepler supports component-level power estimation by reading RAPL energy counters, focusing on the `core`, `uncore`, `package`, and `dram` domains. These energy values are read via the `PowerCap` Framework using the `/sys/class/powercap` interface. The sysfs path tree is parsed dynamically to detect available domains and sockets, ensuring compatibility across architectures and CPU generations. Energy values are read directly from files such as `energy_uj` and divided by 1000 to yield millijoule-level readings. A wraparound detection mechanism ensures robustness even when energy counters overflow.

The core logic is implemented in `UpdateProcessEnergy()`, which is invoked periodically by the main metrics collection loop (and also calls the process attribution logic immediately after the metrics update). However, despite RAPL's native ability to provide energy readings at approximately millisecond-level resolution, Kepler limits energy sampling to a coarse default interval of three seconds (defined in `config.SamplePeriodSec`). This choice reflects a trade-off between performance overhead and metric granularity, but may limit the accuracy for short-lived or bursty workloads.

Importantly, Kepler does not rely on eBPF or perf events to retrieve energy values;

energy is obtained entirely through file-based reads from sysfs or, on some platforms, via MSR or hwmon fallbacks. The collected energy values are later exposed to Prometheus and used in model training and runtime inference. The measurement cadence, attribution methodology, and available domains are validated using an internal tool that checks domain availability and collects average power readings across repeated samples.

Platform-Level Energy Consumption Kepler supports platform-level energy monitoring through external power interfaces exposed by the underlying server hardware. These measurements represent the total energy consumed by the entire node, as opposed to specific hardware components or processes. The implementation is modular, with each power source encapsulated in a corresponding `source` module. Currently supported backends include ACPI (via the `/sys/class/hwmon` interface), Redfish (via the Redfish REST API), and a stub for IBM's HMC interface on `s390x` systems.

Among these, Redfish provides the most detailed and reliable node-level power data. It queries the server's Baseboard Management Controller (BMC) for the `PowerConsumedWatts` value using a REST endpoint. This value is then converted into energy (in milliJoules) by multiplying with the time elapsed since the previous query. Kepler spawns a background goroutine that polls this value at regular intervals (user-configurable via the `REDFISH_PROBE_INTERVAL_IN_SECONDS` parameter in the Kepler configuration). This design allows Redfish to provide cumulative energy measurements with known sampling resolution.

ACPI-based sources offer an alternative when Redfish is unavailable. These rely on instantaneous power averages and do not necessarily represent total node power. The HMC source, by contrast, is currently a non-functional placeholder used only on unsupported platforms. Overall, platform-level metrics are treated as node-wide aggregate energy values without internal attribution, but they offer valuable ground truth for cross-validating other metrics or monitoring infrastructure-level power trends.

Metadata Inputs for Container, System, and VM Attribution To organize energy and resource metrics by container, Kepler collects metadata that maps processes to Kubernetes pods, containers, and namespaces. Kepler supports both cgroups v1 and v2 and dynamically traverses `/sys/fs/cgroup` to map `PID` or `cgroupID` values to container IDs, extracting the last 64 characters of valid cgroup paths.

If the container ID is not cached, Kepler queries Kubernetes for pod metadata using either the Kubelet's local `/pods` API or, if enabled, the Kubernetes API server via a dedicated watcher. Both approaches extract metadata from `ContainerStatuses` fields in pod objects and populate a shared cache. This includes support for init and ephemeral containers. Container ID prefixes are stripped using regex to standardize the format.

Processes not associated with a container are labeled using fallback logic. If the process belongs to the root cgroup (`cgroupID = 1`) and cgroup-based resolution is enabled, it is labeled `container="kernel_processes"`, `namespace="kernel"`. All other unmapped processes are labeled `container="system_processes"`, `namespace="system"`. This includes host services such as `kubelet`, `containerd`.

The Linux idle thread (PID 0), which does not appear in `/proc` and cannot be queried like regular processes, is not explicitly handled by Kepler; however, since it belongs to the root cgroup implicitly, it is effectively included under the `kernel_processes` label.

In addition to container and system process tracking, Kepler supports experimental attribution for virtual machines running under QEMU/KVM on the host. When executed on the hypervisor, Kepler scans `/proc/<pid>/cgroup` for scope names matching the systemd pattern `machine-qemu-*.scope` to identify VM processes. If a match is found, the VM ID is extracted and used to create a `VMStats` structure, allowing the VM to be tracked similarly to a container. Optionally, a metadata lookup via the libvirt API can be used to resolve human-readable VM identifiers. This enables Kepler to expose VM-level resource and energy metrics on systems that mix containers and virtual machines. However, this mechanism only applies to Kepler instances running on the VM host and does not provide visibility into containers running inside the VM.

This metadata layer allows Kepler to cleanly separate containerized workloads, system processes, kernel activity, and virtual machines, ensuring complete coverage of energy attribution in subsequent stages.

GPU Power- and Resource Utilization via NVML Kepler collects per-process GPU utilization statistics using the NVIDIA Management Library (NVML), accessed through internal Go bindings. Specifically, Kepler queries both compute engine usage and memory utilization for each process interacting with an NVIDIA GPU. This data is retrieved via the `ProcessResourceUtilizationPerDevice()` method, which internally calls NVML functions such as `nvmlDevice.GetProcessUtilization` to return per-process statistics, including Streaming Multiprocessor utilization (`SmUtil`), memory utilization (`MemUtil`), and optional encoding/decoding activity. Additionally, KEPLER collects GPU energy consumption using NVML's `nvmlDevice.GetPowerUsage`.

To support NVIDIA's Multi-Instance GPU (MIG) architecture, Kepler first inspects whether MIG slices exist on a given device. If so, it iterates over each slice and retrieves+ per-process utilization data individually. Otherwise, it queries the full physical GPU. In both cases, the parent GPU ID is used as the key to unify resource attribution. For each PID returned by NVML, GPU utilization metrics are appended to the `ResourceUsage` field of the Kepler-internal `ProcessStats` structure. The following metrics are collected:

- `GPUComputeUtilization`: The percentage of compute engine usage (SM activity) over the sampling interval.
- `GPUMemUtilization`: The percentage of frame buffer memory usage per process.

Because NVML does not provide high-resolution energy counters, Kepler approximates GPU energy consumption by multiplying the instantaneous device-level power draw (`GetPowerUsage`) with the sampling interval duration (`SamplePeriodSec`). That is, energy per device is estimated as `energy = power (mW)` $\times$ `SamplePeriodSec (s)`. This coarse-grained sampling, typically performed every few seconds, limits the temporal resolution and may miss short-lived GPU activity.

If `GetProcessUtilization()` is not supported by the hardware or fails at runtime, Kepler falls back to using `GetComputeRunningProcesses()` combined with per-process memory usage to estimate GPU utilization. In this mode, energy is attributed proportionally to memory usage rather than compute activity. Alternative backends such as Habana or DCGM are supported but do not offer per-process utilization data, and are thus not used for fine-grained attribution in Kepler's default configuration.

Summary if Inputs Tables 4.1 and 4.2 summarize the distinct types of input Kepler collects. Metric inputs provide raw resource and energy data, while metadata inputs allow this data to be mapped to containers, VMs, or system processes.

Metric Type	Source	Purpose
CPU hardware counters	`perf_event_open` via eBPF	Track cycles, instructions, cache misses per process
CPU software counters	Custom eBPF programs	Capture CPU time, IRQs, page cache activity
RAPL energy counters	`/sys/class/powercap`	Direct energy measurement for CPU and memory domains
Platform power	Redfish REST API, ACPI sysfs	Estimate total node energy from external sensors
GPU utilization	NVML (via Go bindings)	Track per-process compute and memory usage
GPU energy (approx.)	NVML power reading × interval	Estimate per-device energy usage

TABLE 4.1: Metric inputs used by Kepler for energy and resource monitoring

Metadata Type	Source	Purpose
Container ID extraction	`/proc/<pid>/cgroup`	Identify container context of each process
Pod and namespace info	Kubelet API or Kubernetes API server	Map container ID to pod, namespace, container name
System process fallback	Internal constants + missing container ID	Label processes outside containers as `system_processes`
Kernel thread fallback	`cgroupID = 1`	Label root-cgroup processes as `kernel_processes`
Virtual machine ID	`machine-qemu-*.scope` + optional libvirt lookup	Label QEMU/KVM-based VMs by scope or libvirt metadata
GPU process association	PID-based matching via NVML	Associate GPU usage with Linux processes

TABLE 4.2: Metadata inputs used by Kepler to organize and label monitored workloads

Export Interface and Metric Exposure All metrics collected by Kepler are ultimately exposed via Prometheus after power attribtion. This includes both raw utilization metrics (e.g. CPU time, instructions, cache misses) and derived energy metrics (e.g. power estimates per container). The Prometheus export format allows flexible time series queries and integration with Grafana or other observability platforms.

Estimator Sidecar and Model Inference In addition to its lightweight internal estimator, Kepler optionally supports an estimator sidecar container that uses more sophisticated regression models for power inference. The sidecar communicates with the exporter via a Unix domain socket and loads pre-trained models (e.g. Scikit-learn, XGBoost) suitable for online estimation in environments lacking direct power metrics. This mechanism is mainly intended for telemetry-sparse environments and is not used in the real-time deployment mode considered in this thesis.

Model Server and Model Training For environments where energy measurement is available (e.g. via RAPL or Redfish), Kepler provides a separate model server that facilitates training of machine learning-based energy models. This component ingests time-aligned Prometheus metrics and energy readings, applies preprocessing steps, and generates regression models that estimate power consumption based on utilization metrics. Both absolute and dynamic models can be trained, using different feature groups (e.g., time-based vs instruction-based) and labeled according to power domain (e.g. core, package). The resulting models are stored in a hierarchical directory format and can be deployed in other environments via the estimator sidecar. The model server supports modular pipelines, allowing flexible input data sources, energy isolation techniques, and regression backends. This training loop is essential in contexts where hardware telemetry is available during development or profiling but not in production. While model training plays a key role in extending Kepler to new environments[104], it is not central to the goals of this thesis.

External Integration Kepler is designed to integrate seamlessly with existing observability tools. Prometheus is used to scrape metrics from the Kepler agent, and dashboards can be built using Grafana to visualize energy consumption across nodes, containers, or applications. While not required for the core functionality, this integration facilitates operational awareness and debugging.

4.3.1.3 Attribution Model and Implementation Flow

Overview Kepler estimates the energy consumption of containers by attributing measured or estimated node-level power metrics to individual processes, and aggregating them to container or VM level. This section presents the methodology and implementation approach.

Attribution Methodology At each sampling interval, Kepler separates absolute node power into idle and dynamic components. Process-level dynamic energy is estimated proportionally:

$$E_{\text{process}}^{\text{dyn}} = \frac{U_{\text{process}}}{\sum_i U_i} \cdot E_{\text{node}}^{\text{dyn}}$$

where U is the process's utilization of a given resource (e.g., CPU time).

Data Collection and Flow The data pipeline is composed of modular collectors:

- **Resource Collectors:** Gather CPU and GPU usage via eBPF, perf events, and device APIs.
- **Node Power Collectors:** Query power interfaces (e.g., RAPL, IPMI, Redfish).

- **Energy Estimators:** Calculate idle and dynamic energy.
- **Process Attribution:** Assign energy to processes using ratio-based models.
- **Aggregation Layers:** Summarize process metrics per container or VM.

Modular Design Kepler dynamically activates modules based on the availability of telemetry:

- Real-time power available → measurement-based attribution.
- Real-time unavailable → model-based estimation.
- Optional components (e.g., GPU attribution, platform power) are enabled by configuration.

Implementation Notes The main orchestration is handled by the `Collector` class, which delegates tasks to dedicated modules: `process_bpf_collector`, `process_gpu_collecto` `node_energy_collector`, and `process_energy_collector`.

4.3.1.4 Attribution Model and Output

general Notes dynamic power directly related to resource utilization and idle power is constant power that does not vary regardless of load. SPLIT DIFFERENTLY ACROSS Processes

POWER RATIO MODEL -> divides dynamic power across all processes. The Ratio Power model calculates the ratio of a process's resource utilization to the entire system's resource utilization and then multiplying this ratio by the dynamic power consumption of a resource. This allows us to accurately estimate power usage based on actual resource utilization, ensuring that if, for instance, a program utilizes 10% of the CPU, it consumes 10% of the total CPU power.

The idle power estimation follows the GreenHouse Gas (GHG) protocol guideline, which defines that the constant host idle power should be split among processes/-containers based on their size (relative to the total size of other containers running on the host).

Additionally, it's important to note that different resource utilizations are estimated differently in Kepler. We utilize hardware counters to assess resource utilization in bare-metal environments, using CPU instructions to estimate CPU utilization, collecting cache misses for memory utilization, and assessing Streaming Multiprocessor (SM) utilization for GPUs utilization

- funamentally 3 different models (see model.go)

Process Power Estimation Model

Node Platform Estimation Model

Node Component Estimation Model

Division into idle and dynamic power

Information Flow:

$\text{metric}_c ollector.go process_b pf_c ollector process_g pu_c ollector node_e nergy_c ollector process_e nergy_c ollector$

type Collector struct // NodeStats holds all node energy and resource usage metrics NodeStats stats.NodeStats

// ProcessStats hold all process energy and resource usage metrics ProcessStats map[uint64]*stats.ProcessStats

// ContainerStats holds the aggregated processes metrics for all containers ContainerStats map[string]*stats.ContainerStats

// VMStats holds the aggregated processes metrics for all virtual machines VMStats map[string]*stats.VMStats

// bpfExporter handles gathering metrics from bpf probes bpfExporter bpf.Exporter // bpfSupportedMetrics holds the supported metrics by the bpf exporter bpfSupportedMetrics bpf.SupportedMetrics

func (c *Collector) Update() // collect process resource utilization and aggregate it per node, container and VMs c.updateResourceUtilizationMetrics() // NOTE: no node resource utilization metrics to aggregate c.updateProcessResourceUtilizationMetrics() resourceBpf.UpdateProcessBPFMetrics(c.bpfExporter, c.ProcessStats) // UpdateProcessBPFMetrics reads the BPF tables with process/pid/cgroupid metrics (CPU time, available HW counters) updateSWCounters(mapKey, ct, processStats, bpfSupportedMetrics) AddDeltaStat updateHWCounters(mapKey, ct, processStats, bpfSupportedMetrics) AddDeltaStat // UpdateProcessGPUUtilizationMetrics reads the GPU metrics of each process using the GPU (if enabled) accelerator.UpdateProcessGPUUtilizationMetrics(c.Proce - for all GPU devices (or if applicable, for all MIG devices) addGPUUtilizationToProcessStats(d, processStats, migDevice.(dev.GPUDevice), migDevice.(dev.GPUDevice).ParentID) for pid, processUtilization := range processesUtilization // if the pid is within a container, it will have an container ID –> fail // if the pid is within a VM, it will have an VM ID –> fail addDeltaStat // aggregate processes' resource utilization metrics to containers, virtual machines and nodes c.AggregateProcessResourceUtilizationMetrics() // for - for - for EVERYTHING AddDeltaStat

// collect node power and estimate process power c.UpdateEnergyUtilizationMetrics() c.UpdateNodeEnergyUtilizationMetrics() // UpdateNodeEnergyUtilizationMetrics collects real-time node resource power utilization // if there is no real-time power meter, use the container resource usage metrics to estimate the node's resource power

energy.UpdateNodeEnergyMetrics(c.NodeStats) UpdateNodeComponentsEnergy(nodeStats)
// UpdateNodeComponentsEnergy updates each node component power consumption, i.e., the CPU core, uncore, package/socket and DRAM nodeComponentsEnergy := components.GetAbsEnergyFromNodeComponents() for socket, energy := range nodeComponentsEnergy strID := strconv.Itoa(socket) nodeStats.EnergyUsage[config.AbsEnergy energy.Pkg) UpdateNodeGPUEnergy(nodeStats) gpuEnergy := gpu.Device().AbsEnergyFromDevice() for gpu, energy := range gpuEnergy nodeStats.EnergyUsage[config.AbsEnergyInGPU].SetDeltaStat(fr UpdatePlatformEnergy(nodeStats) if platform.IsSystemCollectionSupported() nodePlatformEnergy, : = *platform.GetAbsEnergyFromPlatform*()*forsourceID*, *energy* := *rangenodePlatformEnergynodeStats.EnergyUsage*[*config.AbsEnergyInPlatform*].*SetDeltaStat*(*source timesystempowermetricsareaccessible.//Whenthenodepowermodelestimatorisutilized*, *theidlepowerisupd*

// the idle energy is only updated if we find the node using less resources than previously observed // TODO: Use regression to estimate the idle power when real-time system power metrics are available, instead of relying on the minimum power consumption. nodeStats.UpdateIdleEnergyWithMinValue(isComponentsSystemCollectionSupported ne.CalcIdleEnergy(config.AbsEnergyInGPU, config.IdleEnergyInGPU, config.GPUComputeUtilizatio ne.CalcIdleEnergy(config.AbsEnergyInCore, config.IdleEnergyInCore, config.CPUTime) ne.CalcIdleEnergy(config.AbsEnergyInDRAM, config.IdleEnergyInDRAM, config.CPUTime) // TODO: we should use another resource for DRAM ne.CalcIdleEnergy(config.AbsEnergyInUnCore, config.IdleEnergyInUnCore, config.CPUTime) ne.CalcIdleEnergy(config.AbsEnergyInPkg, config.IdleEnergyInPkg, config.CPUTime) ne.CalcIdleEnergy(config.AbsEnergyInPlatform, config.IdleEnergyInPlatform, config.CPUTime) func (ne *NodeStats) CalcIdleEnergy(absM, idleM, resouceUtil string) newTotalResUtilization := ne.ResourceUsage[resouceUtil].SumAllDeltaValu currIdleTotalResUtilization := ne.IdleResUtilization[resouceUtil] for socketID, value := range ne.EnergyUsage[absM] // during the first power collection iterations, the delta values could be 0, so we skip until there are delta values currIdleDelta := ne.EnergyUsage[idleM][socketID].GetDelta() // verify if there is a new minimal energy consumption for the given resource // TODO: fix verifying the aggregated resource utilization from all sockets, the update the energy per socket can lead to inconsistency if (newTotalResUtilization <= currIdleTotalResUtilization) || (currIdleDelta == 0) if (currIdleDelta == 0) || (currIdleDelta >= newIdleDelta) ne.EnergyUsage[idleM].SetD newIdleDelta) ne.IdleResUtilization[resouceUtil] = newTotalResUtilization continue // as the dynamic and absolute power, the idle power is also a counter to be exported to prometheus // therefore, we accumulate the minimal found idle if no new one was found ne.EnergyUsage[idleM].SetDeltaStat(socketID, currIdleDelta)

nodeStats.UpdateDynEnergy() // UpdateDynEnergy calculates the dynamic energy. func (s *Stats) UpdateDynEnergy() for pkgID := range s.EnergyUsage[config.AbsEnergyInPkg] s.CalcDynEnergy(config.AbsEnergyInPkg, config.IdleEnergyInPkg, config.DynEnergyInPkg, pkgID) s.CalcDynEnergy(config.AbsEnergyInCore, config.IdleEnergyInCore, config.DynEnergyInCor pkgID) s.CalcDynEnergy(config.AbsEnergyInUnCore, config.IdleEnergyInUnCore, config.DynEnergyInUnCore, pkgID) s.CalcDynEnergy(config.AbsEnergyInDRAM, config.IdleEnergyInDRAM, config.DynEnergyInDRAM, pkgID) for sensorID := range s.EnergyUsage[config.AbsEnergyInPlatform] s.CalcDynEnergy(config.AbsEnergyInPlatform, config.IdleEnergyInPlatform, config.DynEnergyInPlatform, sensorID) // GPU metric if config.IsGPUEnabled() if acc.GetActiveAcceleratorByType(config.GPU) != nil for gpuID := range s.EnergyUsage[config.AbsEnergyInGPU] s.CalcDynEnergy(config.AbsEnergyInGP config.IdleEnergyInGPU, config.DynEnergyInGPU, gpuID) // CalcDynEnergy calculates the difference between the absolute and idle energy/power. func (s *Stats) CalcDynEnergy(absM, idleM, dynM, id string) if ,*exist* := *s.EnergyUsage*[*absM*][*id*]; !*existreturntotalP*

$s.EnergyUsage[absM][id].GetDelta()klog.V(6).Infof("AbsoluteEnergystat : idlePower :=$
$uint64(0)ifidleStat, found := s.EnergyUsage[idleM][id]; foundidlePower = idleStat.GetDelta()klog.V(6).In$
$calcDynEnergy(totalPower, idlePower)s.EnergyUsage[dynM].SetDeltaStat(id, dynPower)klog.V(6).Infof("$

// calcDynEnergy calculates the dynamic energy. func calcDynEnergy(totalE, idleE uint64) uint64 if (totalE == 0) || (totalE < idleE) return 0 return totalE - idleE

nodeStats.SetNodeOtherComponentsEnergy() // SetNodeOtherComponentsEnergy adds the latest energy consumption collected from the other node's components than CPU and DRAM // Other components energy is a special case where the energy is calculated and not measured func (ne *NodeStats) SetNodeOtherComponentsEnergy() // calculate dynamic energy in other components dynCPUComponentsEnergy := ne.EnergyUsage[config.DynEnergyInPkg].SumAllDeltaValues() + ne.EnergyUsage[config.DynEnergyInDRAM].SumAllDeltaValues() + ne.EnergyUsage[config.DynEnergyIn

dynPlatformEnergy := ne.EnergyUsage[config.DynEnergyInPlatform].SumAllDeltaValues()

if dynPlatformEnergy >= dynCPUComponentsEnergy otherComponentEnergy := dynPlatformEnergy - dynCPUComponentsEnergy ne.EnergyUsage[config.DynEnergyInOther].SetDeltaSta otherComponentEnergy)

// calculate idle energy in other components idleCPUComponentsEnergy := ne.EnergyUsage[config.IdleEn + ne.EnergyUsage[config.IdleEnergyInDRAM].SumAllDeltaValues() + ne.EnergyUsage[config.IdleEnergyI

idlePlatformEnergy := ne.EnergyUsage[config.IdleEnergyInPlatform].SumAllDeltaValues()

if idlePlatformEnergy >= idleCPUComponentsEnergy otherComponentEnergy := idlePlatformEnergy - idleCPUComponentsEnergy ne.EnergyUsage[config.IdleEnergyInOther].SetDeltaSta otherComponentEnergy)

c.UpdateProcessEnergyUtilizationMetrics() // UpdateProcessEnergyUtilizationMetrics estimates the process energy consumption using its resource utilization and the node components energy consumption energy.UpdateProcessEnergy(c.ProcessStats, c.NodeStats) // UpdateProcessEnergy resets the power model samples, add new samples to the power models, then estimates the idle and dynamic energy // reset power sample slide window processPlatformPowerModel.ResetSampleIdx() processComponentPowerModel.ResetSampleIdx() // add features values for prediction processIDList := addSamplesToPowerModels(processesMetrics, nodeMetrics) addEstimatedEnergy(processIDList, processesMetrics, idlePower) addEstimatedEnergy(processIDList, processesMetrics, absPower) // addEstimatedEnergy estimates the idle power consumption if processComponentPowerModel.IsEnabled() processComponentsPower, errComp = processComponentPowerModel.GetComponentsPower(isIdlePower) if config.IsGPUEnabled() processGPUPower, errGPU = processComponentPowerModel.GetGPUPower(isIdl // estimate the associated power consumption of platform for each process if processPlatformPowerModel.IsEnabled() processPlatformPower, errPlat = processPlatformPowerModel.GetPlatformPower(isIdlePower) .... for i, processID := range processIDList for pkg, core, dram, uncore, GPU energy = processComponentsPower[i].Pkg * config.SamplePeriodSec() if isIdlePower processesMetrics[processID].EnergyUsage[config.IdleEnergyInP energy) else processesMetrics[processID].EnergyUsage[config.DynEnergyInPkg].SetDeltaStat(utils.Gener energy) // estimate other components power if both platform and components power are available // TODO: verify if Platform power also includes the GPU into

consideration var otherPower uint64 if processPlatformPower[i] <= (processComponentsPower[i].Pkg + processComponentsPower[i].DRAM) otherPower = 0 else otherPower = processPlatformPower[i] - processComponentsPower[i].Pkg - processComponentsPower[i].DRAM energy = otherPower * config.SamplePeriodSec() if isIdlePower processesMetrics[processID].EnergyUsage[config.IdleEnergyInOther].SetDeltaStat(utils.
energy) else processesMetrics[processID].EnergyUsage[config.DynEnergyInOther].SetDeltaStat(utils
energy) // aggregate the process metrics per container and/or VMs c.AggregateProcessEnergyUtiliza
// AggregateProcessEnergyUtilizationMetrics aggregates processes' utilization metrics to containers and virtual machines func (c *Collector) AggregateProcessEnergyUtilizationMetrics() // for - for - for EVERYTHING AddDeltaStat

"self-calibrating and extensible"

4.3.1.5 Attribution Timing and Export Granularity

Kepler attributes energy at fixed internal intervals (default: 3 seconds). These deltas are exposed via Prometheus metrics such as `kepler_container_joules_total`, but the Prometheus scrape interval is user-defined and may not align with Kepler's update loop. If the export interval is not a multiple of the internal interval (e.g., 5s vs. 3s), metric misalignment can occur, leading to visible fluctuations even for stable workloads. Since Kepler does not smooth or interpolate values, this behavior is expected and intentional. For more stable time series, it is recommended to configure the export interval as a multiple of Kepler's internal interval.

4.3.1.6 Validation and Research Context

- a short power model accuracy validation is done in the original KEPLER paper: CPU time leads to better accuracy than CPU cycles, (further investigation is necessary) but generally, more metrics always improve accuracy esults indicate that Linear regression did not yield satisfactory outcomes after incorporating cache and branch miss metrics

Detailed Validation by Pijnacker[105, 106] –> DO AFTER

4.3.1.7 Limitations and Open Issues

documentation often outdated in significant ways, e.g. the switch from tracepoint to kprobe Documentation also outdated in the github (cinsistency, i.e. uml diagrams etc)

4.3.2 Scaphandre

4.3.2.1 Overview and Goals

Scaphandre[107] is an energy monitoring agent designed to expose power consumption metrics at fine granularity, particularly in containerized and virtualized environments. Its name, derived from the French word for "diving suit" reflects its goal of providing deep insights into system-level energy consumption.

Scaphandre aims to attribute energy usage to individual processes, containers, or pods. It supports multiple deployment models: it can be installed directly on a physical host (where it can also monitor qemu/KVM-based virtual machines) or deployed in a container, measuring the energy usage of the host system, provided that

the `/sys/class/powercap` and `/proc` directories are mounted as volumes. Most relevant to this thesis, Scaphandre can also be deployed on a Kubernetes cluster via a Helm chart, optionally alongside *Prometheus* and *Grafana*, allowing for convenient monitoring of Kubernetes nodes and containers.

4.3.2.2 Architecture and Metric Sources

Scaphandre is written in Rust and features a modular, extensible architecture built around two core components: *sensors* and *exporters*. Sensors gather energy-related data from the host system, while exporters format and expose this data to external systems. This design allows seamless integration into cloud-native observability stacks and automation pipelines.

Sensors Scaphandre's *Sensors* subsystem collects utilization and energy consumption metrics from the host system and makes them available to exporters. An abstraction layer, implemented in `sensors/mod.rs`, manages system topology (tracking sensors, CPU sockets, RAPL domains, and processes) and manages the generation of metric records.

The default Linux implementation, `PowercapRAPLSensor` (located in `powercap_rapl.rs`), constructs a power topology by reading Intel RAPL data from the `/sys/class/powercap` interface. By default, it identifies all available domains (across all sockets) by matching directories with the pattern `intel-rapl:<socket>:<domain>`, each containing a domain name and an `energy_uj` file reporting cumulative energy in microjoules. If no domain-level directories are found, Scaphandre triggers a fallback mechanism that omits subdomain detail and instead reads from the socket-level file, if available. This file represents the `package` (PKG) domain, which aggregates the energy consumption of the entire CPU socket. While PKG is a standalone RAPL domain, it also serves as the root of the domain hierarchy. Notably, Scaphandre does not currently handle overflow in the `energy_uj` counter, a known limitation that has not yet been resolved despite user-reported inaccuracies [108].

In addition to the Powercap-based sensor, a Windows-only sensor is available that reads energy consumption directly from Intel or AMD-specific Model-Specific Registers (MSRs).

System utilization metrics are collected by helper functions implemented in `utils.rs`, which rely on Rust's `sysinfo` and `procfs` crates. These libraries extract data from virtual filesystems such as `/proc`, `/sys`, and `/dev`. When container support is enabled, cgroups are also read, but solely to map processes to containers. The collected metrics are stored in a custom data structure along with timestamps, while cgroup information is attached to each process as additional metadata labels. Apart from metrics of processes, CPU and memory, disk read/write operations are also collected.

The *sensors* abstraction layer also applies key data preprocessing steps with are discussed in section 4.3.2.3.

Exporters *Exporters* are responsible for collecting metrics from sensors, storing them temporarily, and exporting them to external systems. Crucially, they also perform the attribution of energy and system metrics to specific scopes, which is discussed

in detail in the following section. Similar to the sensor subsystem, Scaphandre uses an abstraction layer (implemented in `exporters/mod.rs`) to manage common exporter logic such as metric preparation and attribution. Individual exporters are implemented as pluggable modules and operate on a shared core. The `MetricGenerator` plays a central role by consolidating and attributing metrics across all scopes: the Scaphandre agent itself, the host, CPU sockets, RAPL domains, system-wide statistics, and individual processes.

The modular exporter architecture enables easy implementation of new output formats, and Scaphandre explicitly encourages developers to implement custom exporters if needed. Currently, exporters exist for *Prometheus*, *Stdout*, *JSON*, and *QEMU*, among others.

The *QEMU* exporter is particularly notable. It is designed for use on QEMU/KVM hypervisors and allows energy metrics to be exposed to virtual machines (VMs) in the same way that the `powercap` kernel module does on bare-metal systems. Specifically, the QEMU exporter writes VM-specific energy metrics to a virtual file system. Inside the VM, a second Scaphandre instance can read these files as if they were native energy interfaces. This mechanism mimics a bare-metal powercap environment, allowing software running in the VM to access energy data without direct access to RAPL or hardware sensors. This architecture is especially useful for breaking the opacity of power monitoring in virtualized environments, where such metrics are usually unavailable. If adopted by cloud providers, this could significantly improve visibility into energy consumption within public cloud VMs, supporting energy-aware software design even in virtualized infrastructures.

Measurement Interval Scaphandre's measurement interval is fixed at two seconds, as defined in the `show_metrics` function of the file `/exporters/prometheus.rs`. This hardcoded interval determines how frequently the agent performs a new measurement by reading cumulative energy values from the RAPL interface. While RAPL counters are updated approximately every millisecond, Scaphandre does not take advantage of this high-resolution data. Reducing the interval could improve measurement accuracy and temporal granularity (especially for short-lived processes) but would also increase system overhead. As of now, modifying the interval would require changes to the source code.

4.3.2.3 Attribution Model

Scaphandre attributes power consumption to processes and containers using a proportional model based on CPU usage over time. At each sampling interval, it reads the system's total energy consumption from Intel RAPL counters and distributes this energy among all active processes according to their normalized CPU utilization.

The core of this attribution logic is based on CPU time as reported in `/proc`. For each process, Scaphandre accumulates CPU time in active states and excludes time spent in inactive states such as `idle`, `iowait`, `irq`, and `softirq`. This filtering is applied both at the per-process and system level, so that only time considered "active" is included in the normalization denominator. The result is a time-based utilization model that excludes idle-related CPU time from consideration.

Per-process energy is then computed using the following formula, conceptually implemented in `get_process_power_consumption_microwatts()`:

$$E_{\text{proc}}(t) = \frac{E_{\text{RAPL}}(t) \cdot \text{CPU}_{\text{proc}}(t)}{\sum_i \text{CPU}_i(t)} \tag{4.1}$$

where $E_{\text{RAPL}}(t)$ denotes the energy delta over the sampling interval t, and $\text{CPU}_{\text{proc}}(t)$ is the active CPU time of the process. The denominator includes the sum of active CPU time across all processes, excluding inactive states.

Container-level attribution is achieved by inspecting the cgroup path of each process and mapping it to a container or Kubernetes pod using runtime-specific metadata. When container context is available, Scaphandre enriches its per-process metrics with labels such as `container_id`, `kubernetes_pod_name`, and `namespace`. These metrics, such as `scaph_process_power_consumption_microwatts`, are then exposed via Prometheus and can be aggregated at container or pod level.

In addition to process- and container-level metrics, Scaphandre also reports host-level power consumption using the *PSYS* RAPL domains. These host metrics include total platform or package energy consumption, as available, and are exposed via metrics such as `scaph_host_power_microwatts` and `scaph_host_energy_microjoules`. Notably (as pointed out in section 2.4.2), the *PSYS* domain typically does not exist on server-grade systems, in which case Scaphandre adds the *PKG* and *DRAM*-packages to calculate host consumption. Unlike per-process energy values, these host-level metrics reflect total system energy use and are not adjusted to exclude idle CPU time.

Idle power consumption Scaphandre does not compute idle power directly, but it is possible to infer a residual estimate by comparing the total reported host power to the sum of per-process power:

$$P_{\text{idle}} = P_{\text{Host}} - \sum P_{\text{proc}}$$

This difference may include contributions from idle power, system activity outside of user processes, or RAPL domain coverage gaps. However, this value is not part of Scaphandre's exported metrics and must be computed externally if needed.

This attribution methodology is used consistently across the Scaphandre exporters, including the Prometheus exporter. It forms the basis for container-aware energy attribution without requiring additional instrumentation or hardware performance counter support. Further discussion of this methodology's strengths, assumptions, and limitations follows in the next section.

4.3.2.4 Validation and Research Context

To date, no formal validation of Scaphandre's attribution methodology has been published. While the underlying RAPL interface used for energy measurement is widely accepted and validated in prior research, the specific proportional attribution model employed by Scaphandre has not been systematically evaluated against ground-truth data or instruction-based models.Despite this, Scaphandre has been used in academic and applied contexts for estimating the energy consumption of

software systems. In most cases, it is treated as a black-box exporter of container-level energy metrics, with limited investigation into its internal measurement and attribution logic.

A more detailed assessment is presented by Tarara[90], who compares Scaphandre's reported per-process power consumption against both CPU utilization and instruction-based measurements. His case study reveals that Scaphandre tends to overestimate power consumption for lightweight processes under low-load conditions, due to its policy of distributing all observed energy among the small set of active processes. While the model excludes idle time from CPU usage calculations, it does not exclude idle power from total energy, leading to attribution artifacts. Tarara concludes that Scaphandre improves upon naïve utilization-based models, but cannot match the precision of instruction-level approaches using tools such as `perf`.

4.3.2.5 Limitations and Open Issues

While Scaphandre offers a lightweight and transparent approach to energy attribution, several limitations emerge from its current implementation.

First, its fixed measurement interval of 2 seconds limits temporal resolution. Although the underlying RAPL interface supports much finer granularity, Scaphandre aggregates CPU activity over 2-second windows. During this time, the Linux scheduler may switch between dozens or hundreds of processes, depending on system activity. As a result, Scaphandre can only attribute energy based on average CPU utilization over the interval, potentially masking short-lived or bursty behavior.

Second, Scaphandre attempts to refine CPU-based attribution by subtracting inactive time (`idle`, `iowait`, `irq`, `softirq`) from the denominator of its proportional model. While this adjustment excludes clearly non-active states, it does not fully resolve the underlying ambiguity of CPU utilization as a proxy for energy. As noted by Tarara[90], when overall system load is low, the total observed energy (especially idle platform power) is still fully distributed among a small set of active processes. This can result in inflated or misleading per-process energy values, particularly for lightweight workloads.

Scaphandre relies exclusively on Intel's RAPL interface for energy measurements. The developers of Scaphandre acknowledge the lack of detailed public documentation, especially for the `PSys` domain, which they assume to represent total SoC power. Their experiments also highlight ambiguity regarding domain overlaps (for instance, whether the `DRAM` domain is already included in `PKG`, or whether it must be treated as a separate component). These uncertainties may affect the interpretation of host-level energy metrics and the completeness of total system energy accounting.

In a comparison experiment between Scaphandre and other tools, Raffin[26] was unable to push scaphandres measurement frequency beyond 28 Hz, indicating a suboptimal implementation. Despite being written in Rust, Scaphandre performed significantly worse than the python-based tool *CodeCarbon*. Raffin measured Scaphandre's overhead at a non-negligible 3-4% at 10 Hz on an Intel Server.

Finally, Scaphandre does not incorporate instruction-level or performance counter-based metrics. It cannot distinguish between processes with different execution intensities or memory behavior, and assumes a uniform relationship between CPU time and energy use. This limits its ability to reflect differences in instruction throughput, stalling, or architectural efficiency across workloads.

4.3.3 SmartWatts

4.3.3.1 Overview and Goals

The PowerAPI-implementation *SmartWatts*[98] is a software-defined, self-calibrating power 'formula' designed for estimating power consumption of containers, processes, and VMs. It aims to address the shortcomings of static power models by using online model adaptation (sequential learning) and runtime performance counters. Unlike many academic models that require manual calibration or architecture-specific training, SmartWatts adapts automatically to the host system and workload.

4.3.3.2 Architecture and Metric Sources

SmartWatts is written in Pyhton. Understanding the architecture of SmartWatts and its differences from other energy monitoring tools is crucial. Using HWPC, RAPL, and CPU process metrics, SmartWatts collects performance data. At runtime, it uses power models based on cgroups and perf events alone to estimate, for each resource $\text{res} \in \{\text{CPU}, \text{DRAM}\}$, the host power consumption $\hat{p}_{\text{res}}$ and the power consumption $\hat{p}_{\text{res}}(c)$ for all containers. SmartWatts uses $\hat{p}_{\text{res}}$ to continuously assess the accuracy of the managed power models $M_{\text{res},f}$ to ensure that estimated power consumption does not diverge from the RAPL baseline measurement $p_{\text{res}}^{\text{rapl}}$. When the estimation diverges beyond a configurable threshold ϵ_{res}, SmartWatts triggers a new online calibration process for the model. When the machine is at rest (e.g., after a reboot), this method is also used to isolate the static energy consumption. A simple architecture can be seen in Figure 4.3.

In practical terms, SmartWatts implements a server-side pwoermeter (referred to as *power meter*) that consumes input samples and produces power estimations accordingly. The power meter is responsible for power modelling, power estimation and model calibration. In addition, a client-side sensor (referred to as *sensor*) is deployed as a lightweight daemon on all cluster nodes. The sensor is responsible static power isolation, event selection, cgroups and event monitoring. This separation allows for heterogeneous cluster nodes.

4.3.3.3 Attribution Model

As discussed in the previous subsection, the SmartWatts attribution model does not use RAPL metrics, opting only for process metrics. SmartWatts separates host energy consumption into static and dynamic power consumption:

$$p_{res}^{rapl} = p_{res}^{static} + p_{res}^{dynamic} \tag{4.2}$$

Static power is estimated by periodically logging RAPL package and DRAM power consumption. The *median* value and the *interquartilerange* (IRQ) are gathered from

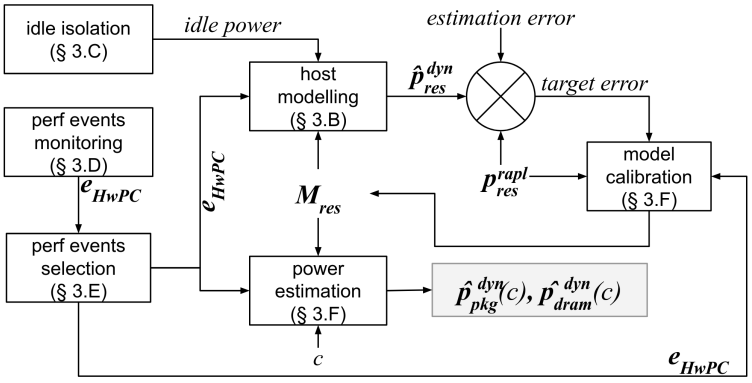


FIGURE 4.3: SmartWatts Architecture

teh measurements to define the static host power consumption as

$$p_{res}^{static} := median_{res} - 1.5 \cdot IRQ_{res} \tag{4.3}$$

This approach is meant to filter out RAPL outliers.

Dynamic power is estimated by correlating the CPU frequency f and the raw metrics reported by HWCP:

$$\exists f \in F, \hat{p}_{res}^{dyn} = M_{res}^{f} \cdot E_{res}^{f} \tag{4.4}$$

where E_{res}^{f} denotes all *events*. The model M_{res}^{f} is build from *elastic net* regression applied on the last k samples. To ensure that all container power consumptions are linear with regards to global power consumption, positive inference coefficients are enforced, and the intercept (or *bias term* is within the range $[0, TDP]$.

HWPC metrics are dynamically chosen based on the list of available events exposed by the host's *Performance Monitoring Units* (PMU), essentially creating a custom model based on available metrics. Not all available metrics are used, and statistical analysis (Pearson coefficient) is used to determining worthy candidates.

Container power consumption is estimated by applying the inferred power model M_{res}^{f} at the scale of the container's events $E_{res}^{f}(c)$, as seen in formula 4.5. In formula 4.6, the intercept i is distributed proportionally to the dynamic part of the consumption of c.

$$\exists f \in F, \forall c \in C, \hat{p}_{res}^{dyn}(c) = M_{res}^{f} \cdot E_{res}^{f}(c) \tag{4.5}$$

$$\forall c \in C, \tilde{p}_{res}^{dyn}(c) = \hat{p}_{res}^{dyn}(c) - i \cdot (1 - \frac{\hat{p}_{res}^{dyn}(c) - i}{\hat{p}_{res}^{dyn} - i}) \tag{4.6}$$

In theory, one can expect $\hat{p}_{res}^{dyn} \stackrel{!}{=} p_{res}^{dyn}$ if the model perfectly estimates the dynamic power consumption, but in practice, an error $\epsilon_{res} = \left| p_{res}^{dyn} - \hat{p}_{res}^{dyn} \right|$. Therefore, container power consumption is capped at

$$\forall c \in C, \left\lceil \tilde{p}_{res}^{dyn}(c) \right\rceil = \frac{p_{res}^{dyn} \cdot \tilde{p}_{res}^{dyn}(c)}{\hat{p}_{res}^{dyn}} \tag{4.7}$$

This approach also allows to calculate a confidence intercal of the power consumption of containers by scaling down the observed global error:

$$\forall c \in C, \epsilon_{res}(c) = \frac{\tilde{p}_{res}^{dyn}(c)}{\hat{p}_{res}^{dyn}} \cdot \epsilon_{res} \tag{4.8}$$

In order improve estimation accuracy, the following configurable parameters are used:

- CPU-TDP in Watt (default: 125)
- CPU base clock in MHz (default: 100)
- CPU base frequency in MHz (default: 2100)
- CPU and DRAM error threshold in Watt (default: 2)
- Minimum of samples required before trying to learn a power model (default: 10)
- Size of the history window used ot keep samples from (default: 60)
- Measurement frequency in milliseconds (default: 1000)

4.3.3.4 Validation and Research Context

With RAPL being used as ground truth for dynamic power estimation model recalibration, it is important to note that the SmartWatts validation is focused on the model accuracy when compared to RAPL values instead of values obtained by an external source of power data. The SmartWatts validation focused on the quality of power estimation in sequential and parallel workloads, the accuracy and sstability of power models and the overhead of the *sensor* component. Standard benchmarks like Stress-NG and NAS parallel Benchmarks were chosen.

While there is no advanced statistical analysis, the validation shows that, for a error threshold for CPU and DRAM of 5 and 1 Watt respectively, power consumption can be reliably estimated with less than 3 Watts and 0.5 Watts, respectively. The only case where the error grows beyond the threshold is at the CPU idle frequency. The model stability is shown to significantly improve when lower recalibration frequencies are used. SmartWatts succeeds to reuse a give power model for up to 594 estimations, depending on frequency. The monitoring overhead is observed to be at 0.333 Watts for the PKG domain and 0.030 Watts for the DRAM domain on average, at a measurement frequency of 2 Hz. The authors consider this overhead negligable.

4.3.3.5 Limitations and Open Issues

SmartWatts offers a compelling solution for dynamic, container-level power estimation through self-calibrating models based on performance counters. However, its applicability remains domain-specific. The central assumption is that RAPL, while accurate, is too coarse-grained for attributing power to individual containers or processes. This premise is debatable: RAPL offers low-overhead, high-frequency measurements, and may be sufficient for many use cases, particularly in homogeneous

or single-tenant systems. Whether SmartWatts' added complexity is justified depends on how fine-grained the attribution needs to be.

SmartWatts shines when more granular telemetry (e.g., perf events) is available and container-level attribution is critical. Yet its current implementation models only CPU and DRAM domains, limiting its ability to offer a comprehensive energy profile.

The design allows operators to supply hardware-specific values (e.g. CPU TDP), while falling back to sensible defaults. This improves usability without sacrificing model accuracy.

Finally, while SmartWatts' runtime calibration and dynamic event selection enhance adaptability, they introduce complexity. The event selection mechanism relies on statistical heuristics, which may not generalize well across systems. Moreover, under highly dynamic conditions, frequent recalibrations may affect stability.

In summary, SmartWatts is well-suited for environments requiring high-resolution attribution beyond RAPL's capabilities, but its scope, complexity, and assumptions warrant careful consideration depending on the target use case.

4.4 Comparison of Container-Level Tools

4.4.1 Feature Comparison

4.4.2 Granularity and Metric Sources

4.4.3 Platform Compatibility and Integration

4.5 Relevance to Proposed Architecture

4.5.1 Lessons Learned from Existing Tools

4.5.2 Identified Gaps and Opportunities

4.5.3 Implications for Chapter ??

4.6 Summary

4.6.0.1 Overview and Architecture

4.6.0.2 Metrics and Data Sources

4.6.0.3 Attribution Method and Scope

4.6.0.4 Validation and Limitations

4.6.0.5 Relevance to Proposed Architecture

Chapter 5

Designing a Container Power Attribution Architecture

5.1 Design Goals Generalizability, minimal overhead, real-time capability, etc. 5.2 Architecture Components Metric sources, data aggregation, correlation layer, exporter 5.3 Attribution Logic CPU (RAPL + cgroups), RAM, NET, DISK, idle, system 5.4 Proposed Correlation Model Hybrid models (e.g. direct for CPU, proportional for network) 5.5 Open Questions and Future Improvements

Notes VM nesting (like scaphandre) in Kepler -> passthrough explicitely handle PID0, idle task let the user define interal measurement intervals kepler exporter: multiple of measurement intervals. fundamentally: SHOULD we cross-reference node and process consumption, with drastically different accuracies? did kepler just stop midway? no activity in the github, but many TODO, currently 1053.

Chapter 6

Methodology for Evaluation and Benchmarking

6.1 Requirements for Evaluation Ground-truth measurement, reproducibility, load generation 6.2 Testbed Design Kubernetes cluster layout, benchmarking tools (e.g., stress-ng, fio, iperf3) 6.3 Benchmarking Scenarios Synthetic benchmarks (CPU/memory-heavy, IO-heavy, mixed) Real-world workloads (web apps, ML inference, etc.) 6.4 Evaluation Metrics Attribution accuracy, overhead, scalability, stability

Chapter 7

Conclusion and Outlook

7.1 Summary of Findings 7.2 Implications for Practice 7.3 Limitations of the Study 7.4 Outlook on Future Work

Solutions that would improve the general situation - homogeneous servers, making more specific analysis financially viable - more research of course

Appendix A

Appendix Title

Bibliography

[1] Caspar Wackerle. *PowerStack: Automated Kubernetes Deployment for Energy Efficiency Analysis*. GitHub repository. 2025. URL: https://github.com/casparwackerle/PowerStack.

[2] International Energy Agency. *Energy and AI*. Licence: CC BY 4.0. Paris, 2025. URL: https://www.iea.org/reports/energy-and-ai.

[3] Ryan Smith. *Intel's CEO Says Moore's Law Is Slowing to a Three-Year Cadence — But It's Not Dead Yet*. Accessed: 2025-04-14. 2023. URL: https://www.tomshardware.com/tech-industry/semiconductors/intels-ceo-says-moores-law-is-slowing-to-a-three-year-cadence-but-its-not-dead-yet.

[4] Martin Keegan. *The End of Dennard Scaling*. Accessed: 2025-04-14. 2013. URL: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/.

[5] Uptime Institute. *Global PUEs – Are They Going Anywhere?* Accessed: 2025-04-14. 2023. URL: https://journal.uptimeinstitute.com/global-pues-are-they-going-anywhere/.

[6] Eric Masanet et al. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020), pp. 984–986. DOI: 10.1126/science.aba3758. eprint: https://www.science.org/doi/pdf/10.1126/science.aba3758. URL: https://www.science.org/doi/abs/10.1126/science.aba3758.

[7] Amit M. Potdar et al. "Performance Evaluation of Docker Container and Virtual Machine". In: *Procedia Computer Science* 171 (2020), pp. 1419–1428. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.04.152.

[8] Roberto Morabito. "Power Consumption of Virtualization Technologies: An Empirical Investigation". In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. Dec. 2015, pp. 522–527. DOI: 10.1109/UCC.2015.93. (Visited on 05/21/2025).

[9] OpenAI. *ChatGPT (Version 4o)*. Used for document generation and formatting. 2025. URL: https://chat.openai.com.

[10] Saiqin Long et al. "A Review of Energy Efficiency Evaluation Technologies in Cloud Data Centers". In: *Energy and Buildings* 260 (Apr. 2022), p. 111848. ISSN: 0378-7788. DOI: 10.1016/j.enbuild.2022.111848. (Visited on 04/20/2025).

[11] Chaoqiang Jin et al. "A Review of Power Consumption Models of Servers in Data Centers". In: *Applied Energy* 265 (May 2020), p. 114806. ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2020.114806. (Visited on 03/16/2025).

[12] Hannes Tröpgen et al. "16 Years of SPEC Power: An Analysis of X86 Energy Efficiency Trends". In: *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. Sept. 2024, pp. 76–80. DOI: 10.1109/CLUSTERWorkshops61563.2024.00020. (Visited on 04/20/2025).

[13] Weiwei Lin et al. "A Taxonomy and Survey of Power Models and Power Modeling for Cloud Servers". In: *ACM Comput. Surv.* 53.5 (Sept. 2020), 100:1–100:41. ISSN: 0360-0300. DOI: 10.1145/3406208. (Visited on 04/20/2025).

[14] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 455–470. DOI: 10.1145/2989081.2989088. (Visited on 05/21/2025).

[15] Richard Kavanagh, Django Armstrong, and Karim Djemame. "Accuracy of Energy Model Calibration with IPMI". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. June 2016, pp. 648–655. DOI: 10.1109/CLOUD.2016.0091. (Visited on 04/23/2025).

[16] Richard Kavanagh and Karim Djemame. "Rapid and Accurate Energy Models through Calibration with IPMI and RAPL". In: *Concurrency and Computation: Practice and Experience* 31.13 (2019), e5124. ISSN: 1532-0634. DOI: 10.1002/cpe.5124. (Visited on 04/23/2025).

[17] Joseph P. White et al. "Monitoring and Analysis of Power Consumption on HPC Clusters Using XDMoD". In: *Practice and Experience in Advanced Research Computing*. Portland OR USA: ACM, July 2020, pp. 112–119. ISBN: 978-1-4503-6689-2. DOI: 10.1145/3311790.3396624. (Visited on 04/23/2025).

[18] Thomas-Krenn.AG. *Redfish - Thomas-Krenn-Wiki*. Accessed: April 27, 2025. n.d. URL: https://www.thomas-krenn.com/de/wiki/Redfish.

[19] Yewan Wang et al. "An Empirical Study of Power Characterization Approaches for Servers". In: *ENERGY 2019 - The Ninth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. June 2019, p. 1. (Visited on 04/23/2025).

[20] UEFI Forum. *Advanced Configuration and Power Interface Specification Version 6.6*. Accessed April 2025. Sept. 2021. URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.6.pdf.

[21] Project Exigence. *Running Average Power Limit (RAPL)*. https://projectexigence.eu/green-ict-digest/running-average-power-limit-rapl/. Accessed April 2025. n.d.

[22] AMD. *amd_energy: AMD Energy Driver*. Accessed: 2025-04-28. 2023. URL: https://github.com/amd/amd_energy.

[23] Robert Schöne et al. "Energy Efficiency Aspects of the AMD Zen 2 Architecture". In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2021, pp. 562–571. DOI: 10.1109/Cluster48925.2021.00087. (Visited on 04/28/2025).

[24] Mathilde Jay et al. "An Experimental Comparison of Software-Based Power Meters: Focus on CPU and GPU". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. May 2023, pp. 106–118. DOI: 10.1109/CCGrid57682.2023.00020. (Visited on 04/21/2025).

[25] Tom Kennes. *Measuring IT Carbon Footprint: What Is the Current Status Actually?* June 2023. DOI: 10.48550/arXiv.2306.10049. arXiv: 2306.10049 [cs]. (Visited on 04/23/2025).

[26] Guillaume Raffin and Denis Trystram. "Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis". In: *IEEE Transactions on Parallel and Distributed Systems* 36.1 (Jan. 2025), pp. 96–107. ISSN: 1558-2183. DOI: 10.1109/TPDS.2024.3492336. (Visited on 04/02/2025).

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3B, Chapter 16.10: Plaform Specific Power Management Support. Available online: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. 2024.

[28] Robert Schöne et al. "Energy Efficiency Features of the Intel Alder Lake Architecture". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. London United Kingdom: ACM, May 2024, pp. 95–106. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645040. (Visited on 04/07/2025).

[29] Daniel Hackenberg et al. "An Energy Efficiency Feature Survey of the Intel Haswell Processor". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 896–904. DOI: 10.1109/IPDPSW.2015.70. (Visited on 04/28/2025).

[30] Daniel Hackenberg et al. "Power Measurement Techniques on Standard Compute Nodes: A Quantitative Comparison". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 194–204. DOI: 10.1109/ISPASS.2013.6557170. (Visited on 04/28/2025).

[31] Lukas Alt et al. "An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 71–82. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645052. (Visited on 04/02/2025).

[32] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018), 9:1–9:26. ISSN: 2376-3639. DOI: 10.1145/3177754. (Visited on 04/07/2025).

[33] Marcus Hähnel et al. "Measuring Energy Consumption for Short Code Paths Using RAPL". In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012), pp. 13–17. ISSN: 0163-5999. DOI: 10.1145/2425248.2425252. (Visited on 05/21/2025).

[34] "Detailed and Simultaneous Power and Performance Analysis - Servat - 2016 - Concurrency and Computation: Practice and Experience - Wiley Online Library". In: (). (Visited on 05/21/2025).

[35] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on X86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063. (Visited on 05/21/2025).

[36] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers*. Tech. rep. 335592-081US. Accessed 2025-04-28. Intel Corporation, Sept. 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/671098.

[37] Green Coding Berlin. *RAPL, SGX and energy filtering - Influences on power consumption*. Accessed May 2025. 2022. URL: https://www.green-coding.io/case-studies/rapl-and-sgx/.

[38] Intel Corporation. *Running Average Power Limit (RAPL) Energy Reporting*. Accessed May 2025. 2022. URL: https://www.intel.cn/content/www/cn/zh/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[39] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. (Visited on 05/21/2025).

[40] Kubernetes Documentation. *GPUs in Kubernetes*. Accessed: 2025-05-09. 2025. URL: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

[41] Inc. Datadog. *2024 Container Report*. Accessed: 2025-05-09. 2024. URL: https://www.datadoghq.com/container-report/.

[42] TensorFlow Documentation. *Running TensorFlow on Kubernetes*. Accessed: 2025-05-09. 2025. URL: https://www.tensorflow.org/tfx/serving/serving_kubernetes.

[43] NVIDIA Corporation. *NVIDIA Virtualization Resources*. Accessed: 2025-05-09. 2025. URL: https://www.nvidia.com/de-de/data-center/virtualization/resources/.

[44] AMD Corporation. *AMD Instinct Virtualization Documentation*. Accessed: 2025-05-09. 2025. URL: https://instinct.docs.amd.com/projects/virt-drv/en/latest/index.html.

[45] NVIDIA Corporation. *NVIDIA Multi-Instance GPU (MIG) User Guide*. Accessed: 2025-05-09. 2025. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[46] NVIDIA Corporation. *NVIDIA GPU Passthrough Documentation*. Accessed: 2025-05-09. 2025. URL: https://docs.nvidia.com/datacenter/tesla/gpu-passthrough/index.html.

[47] NVIDIA Corporation. *NVIDIA System Management Interface (nvidia-smi)*. Accessed: 2025-05-09. 2025. URL: https://developer.nvidia.com/system-management-interface.

[48] Zeyu Yang, Karel Adamek, and Wesley Armour. "Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor". In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2024, pp. 1–17. DOI: 10.1109/SC41406.2024.00028. (Visited on 05/09/2025).

[49] Vijay Kandiah et al. "AccelWattch: A Power Modeling Framework for Modern GPUs". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 738–753. ISBN: 978-1-4503-8557-2. DOI: 10.1145/3466752.3480063. (Visited on 05/09/2025).

[50] Varsha Singhania, Shaizeen Aga, and Mohamed Assem Ibrahim. *FinGraV: Methodology for Fine-Grain GPU Power Visibility and Insights*. Mar. 2025. DOI: 10.48550/arXiv.2412.12426. arXiv: 2412.12426 [cs]. (Visited on 05/09/2025).

[51] Steven van der Vlugt et al. *PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool*. Apr. 2025. DOI: 10.48550/arXiv.2504.17883. arXiv: 2504.17883 [cs]. (Visited on 05/09/2025).

[52] Anthony Hylick et al. "An Analysis of Hard Drive Energy Consumption". In: *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*. Sept. 2008, pp. 1–10. DOI: 10.1109/MASCOT.2008.4770567. (Visited on 05/21/2025).

[53] Seokhei Cho et al. "Design Tradeoffs of SSDs: From Energy Consumption's Perspective". In: *ACM Trans. Storage* 11.2 (Mar. 2015), 8:1–8:24. ISSN: 1553-3077. DOI: 10.1145/2644818. (Visited on 05/18/2025).

[54] Linux NVMe Maintainers. *nvme-cli: NVMe management command line interface*. https://github.com/linux-nvme/nvme-cli. Accessed May 2025. 2025.

[55] smartmontools developers. *smartmontools: Control and monitor storage systems using S.M.A.R.T.* https://github.com/smartmontools/smartmontools/. Accessed May 2025. 2025.

[56] TechNotes. *Deciphering the PCI Power States*. Accessed June 2025. Feb. 2024. URL: https://technotes.blog/2024/02/04/deciphering-the-pci-power-states/.

[57] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. "Power Provisioning for a Warehouse-Sized Computer". In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 13–23. ISSN: 0163-5964. DOI: 10.1145/1273440.1250665. (Visited on 05/21/2025).

[58] Chung-Hsing Hsu and Stephen W. Poole. "Power Signature Analysis of the SPECpower_ssj2008 Benchmark". In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2011, pp. 227–236. DOI: 10.1109/ISPASS.2011.5762739. (Visited on 05/21/2025).

[59] Anton Beloglazov et al. "Chapter 3 - A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems". In: *Advances in Computers*. Ed. by Marvin V. Zelkowitz. Vol. 82. Elsevier, Jan. 2011, pp. 47–111. DOI: 10.1016/B978-0-12-385512-1.00003-7. (Visited on 06/09/2025).

[60] E. N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. "Energy-Efficient Server Clusters". In: *Power-Aware Computer Systems*. Ed. by Babak Falsafi and T. N. Vijaykumar. Berlin, Heidelberg: Springer, 2003, pp. 179–197. ISBN: 978-3-540-36612-6. DOI: 10.1007/3-540-36612-1_12.

[61] Shuaiwen Leon Song, Kevin Barker, and Darren Kerbyson. "Unified Performance and Power Modeling of Scientific Workloads". In: *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*. E2SC '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 1–8. ISBN: 978-1-4503-2504-2. DOI: 10.1145/2536430.2536435. (Visited on 05/21/2025).

[62] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. "Energy Efficiency in Cloud Computing Data Center: A Survey on Hardware Technologies". In: *Cluster Computing* 25.1 (Feb. 2022), pp. 675–705. ISSN: 1573-7543. DOI: 10.1007/s10586-021-03431-z. (Visited on 06/04/2025).

[63] Robert Basmadjian et al. "A Methodology to Predict the Power Consumption of Servers in Data Centres". In: *Proceedings of the 2nd International Conference on Energy-Efficient Computing and Networking*. E-Energy '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 1–10. ISBN: 978-1-4503-1313-1. DOI: 10.1145/2318716.2318718. (Visited on 06/09/2025).

[64] L. Luo, W.-J Wu, and F. Zhang. "Energy modeling based on cloud data center". In: *Ruan Jian Xue Bao/Journal of Software* 25 (July 2014), pp. 1371–1387. DOI: 10.13328/j.cnki.jos.004604.

[65] Robert Basmadjian and Hermann De Meer. "Evaluating and modeling power consumption of multi-core processors". In: *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*. 2012, pp. 1–10.

[66] Osman Sarood et al. "Maximizing throughput of overprovisioned hpc data centers under a strict power budget". In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 807–818.

[67] Weiwei Lin et al. "A cloud server energy consumption measurement system for heterogeneous cloud environments". In: *Information Sciences* 468 (2018), pp. 47–62.

[68] Patricia Arroba et al. "Server power modeling for run-time energy optimization of cloud computing facilities". In: *Energy Procedia* 62 (2014), pp. 401–410.

[69] Aman Kansal et al. "Virtual machine power metering and provisioning". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 39–50.

[70] Miriam Allalouf et al. "Storage Modeling for Power Estimation". In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. SYSTOR '09. New York, NY, USA: Association for Computing Machinery, May 2009, pp. 1–10. ISBN: 978-1-60558-623-6. DOI: 10.1145/1534530.1534535. (Visited on 05/18/2025).

[71] StoreDbits. *Hard Drive Power Consumption (HDD)*. Accessed May 2025. 2023. URL: https://storedbits.com/hard-drive-power-consumption/.

[72] StoreDbits. *SSD Power Consumption*. Accessed May 2025. 2023. URL: https://storedbits.com/ssd-power-consumption/.

[73] Yan Li and Darrell D.E. Long. "Which Storage Device Is the Greenest? Modeling the Energy Cost of I/O Workloads". In: *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. Sept. 2014, pp. 100–105. DOI: 10.1109/MASCOTS.2014.20. (Visited on 05/19/2025).

[74] Eric Borba, Eduardo Tavares, and Paulo Maciel. "A Modeling Approach for Estimating Performance and Energy Consumption of Storage Systems". In: *Journal of Computer and System Sciences* 128 (Sept. 2022), pp. 86–106. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2022.04.001. (Visited on 05/18/2025).

[75] Ripduman Sohan et al. "Characterizing 10 Gbps Network Interface Energy Consumption". In: *IEEE Local Computer Network Conference*. Oct. 2010, pp. 268–271. DOI: 10.1109/LCN.2010.5735719. (Visited on 05/30/2025).

[76] Corey Gough, Ian Steiner, and Winston A. Sanders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, 2015. ISBN: 9781430266372. DOI: 10.1007/978-1-4302-6638-9.

[77] Robert Basmadjian et al. "Cloud Computing and Its Interest in Saving Energy: The Use Case of a Private Cloud". In: *Journal of Cloud Computing: Advances, Systems and Applications* 1.1 (June 2012), p. 5. ISSN: 2192-113X. DOI: 10.1186/2192-113X-1-5. (Visited on 06/01/2025).

[78] Jordi Arjona Aroca et al. "A Measurement-Based Analysis of the Energy Consumption of Data Center Servers". In: *Proceedings of the 5th International Conference on Future Energy Systems*. E-Energy '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 63–74. ISBN: 978-1-4503-2819-7. DOI: 10.1145/2602044.2602061. (Visited on 06/01/2025).

[79] Vincenzo De Maio et al. "Modelling Energy Consumption of Network Transfers and Virtual Machine Migration". In: *Future Generation Computer Systems* 56 (Mar. 2016), pp. 388–406. ISSN: 0167-739X. DOI: 10.1016/j.future.2015.07.007. (Visited on 06/04/2025).

[80] Waltenegus Dargie and Jianjun Wen. "A Probabilistic Model for Estimating the Power Consumption of Processors and Network Interface Cards". In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. July 2013, pp. 845–852. DOI: 10.1109/TrustCom.2013.103. (Visited on 06/04/2025).

[81] Saeedeh Baneshi et al. "Analyzing Per-Application Energy Consumption in a Multi-Application Computing Continuum". In: *2024 9th International Conference on Fog and Mobile Edge Computing (FMEC)*. Sept. 2024, pp. 30–37. DOI: 10.1109/FMEC62297.2024.10710253. (Visited on 05/30/2025).

[82] The Linux Kernel Community. *The proc Filesystem*. https://www.kernel.org/doc/html/latest/filesystems/proc.html. Accessed: 2025-06-17. 2025.

[83] The Linux Kernel Community. *Control Group v1 — Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/index.html. Accessed: 2025-06-17. 2025.

[84] The Linux Kernel Community. *Control Group v2 — Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html. Accessed: 2025-06-17. 2025.

[85] Cilium Authors. *eBPF and XDP Reference Guide*. https://docs.cilium.io/en/latest/reference-guides/bpf/index.html. Accessed: 2025-06-17. 2025.

[86] Google Inc. *cAdvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers*. https://github.com/google/cadvisor. Accessed: 2025-06-14. 2025.

[87] Kubernetes-SIG. *metrics-server: Scalable and efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines*. https://github.com/kubernetes-sigs/metrics-server. Accessed: 2025-06-17. 2025.

[88] Cyril Cassagnes et al. "The Rise of eBPF for Non-Intrusive Performance Monitoring". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2020, pp. 1–7. DOI: 10.1109/NOMS47738.2020.9110434. (Visited on 06/14/2025).

[89] Brendan Gregg. *CPU Utilization is Wrong*. Blog post. Accessed 29 June 2025. May 2017. URL: https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html.

[90] Arne Tarara. *CPU Utilization – A Useful Metric?* Green Coding Case Study. Accessed 29 June 2025. June 2023. URL: https://www.green-coding.io/case-studies/cpu-utilization-usefulness/.

[91] Adrian Cockcroft. "Utilization is virtually useless as a metric!" In: *Int. CMG Conference*. 2006, pp. 557–562.

[92] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. New York, NY, USA: Cambridge University Press, 2013. ISBN: 9781107027503.

[93] CodeCarbon Team. *mlco2/codecarbon: v2.4.1*. Version v2.4.1. May 2024. DOI: 10.5281/zenodo.11171501. URL: https://doi.org/10.5281/zenodo.11171501.

[94] CodeCarbon Contributors. *CodeCarbon issue #322: API endpoint and swagger docs*. https://github.com/mlco2/codecarbon/issues/322. Accessed: 2025-06-21. 2022.

[95] GreenAI-UPPA. *AI PowerMeter: A Tool to Estimate the Energy Consumption of AI Workloads*. Accessed: 2025-04-28. 2023. URL: https://greenai-uppa.github.io/AIPowerMeter/.

[96] Pierre Fieni et al. *PowerAPI is a green-computing toolbox to measure, analyze, and optimize the energy consumption of the various hardware/software levels composing an infrastructure*. https://github.com/powerapi-ng. Accessed June 2025. 2024.

[97] Guillaume Fieni et al. "PowerAPI: A Python Framework for Building Software-Defined Power Meters". In: *Journal of Open Source Software* 9.98 (June 2024), p. 6670. DOI: 10.21105/joss.06670. (Visited on 06/21/2025).

[98] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. "SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. May 2020, pp. 479–488. DOI: 10.1109/CCGrid49817.2020.00-45. (Visited on 05/21/2025).

[99] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[100] Arne Tarara. *Green Coding Documentation*. Accessed: 2025-04-28. 2023. URL: https://github.com/green-coding-solutions/green-metrics-tool.

[101] Inc. Meta Platforms. *Kepler: Kubernetes-based power and energy estimation framework*. Accessed: 2025-04-28. 2023. URL: https://github.com/sustainable-computing-io/kepler.

[102] Marcelo Amaral et al. "Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications". In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. July 2023, pp. 69–71. DOI: 10.1109/CLOUD60044.2023.00017. (Visited on 03/10/2025).

[103] Sustainable Computing. *Kepler: Kubernetes Efficient Power Level Exporter Documentation*. https://sustainable-computing.io/. Accessed: 2025-07-08. 2025.

[104] Sunyanan Choochotkaew et al. "Advancing Cloud Sustainability: A Versatile Framework for Container Power Model Training". In: *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Oct. 2023, pp. 1–4. DOI: 10.1109/MASCOTS59514.2023.10387542. (Visited on 07/02/2025).

[105] Bjorn Pijnacker. "Estimating Container-level Power Usage in Kubernetes". MA thesis. University of Groningen, Nov. 2024. (Visited on 03/17/2025).

[106] Bjorn Pijnacker, Brian Setz, and Vasilios Andrikopoulos. *Container-Level Energy Observability in Kubernetes Clusters*. Apr. 2025. DOI: 10.48550/arXiv.2504.10702. arXiv: 2504.10702 [cs]. (Visited on 07/02/2025).

[107] Hubblo. *Scaphandre: Energy consumption monitoring agent*. https://github.com/hubblo-org/scaphandre. Accessed: 2025-06-24. 2025.

[108] Hubblo. *Overflow in energy counter can lead to wrong power measurements*. https://github.com/hubblo-org/scaphandre/issues/280. GitHub issue #280, *hubblo-org/scaphandre*. Feb. 2024. (Visited on 06/25/2025).