Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

MASTER THESIS

Tycho:

An Accuracy-First Architecture for Server-Wide Energy Measurement and Process-Level Attribution in Kubernetes

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
January 31, 2026

Study program:
Computer Science, M.Sc.

Imprint

Project: Master Thesis
Title: Tycho: An Accuracy-First Architecture for Server-Wide Energy Measurement and Process-Level Attribution in Kubernetes
Author: Caspar Wackerle
Date: January 31, 2026
Keywords: process-level energy consumption, 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.

Contents

Appendix A: Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches

List of Figures

Appendix A

List of Tables

The global climate crisis is one of humanity's greatest challenges in this century.
With this work, I hope to contribute a small part in the direction we urgently need to go.

Chapter 1

Introduction

1.1 Motivation

Global data center energy consumption is increasing rapidly, driven by the rising demand for compute-heavy workloads such as artificial intelligence and large-scale analytics. Data centers consumed roughly 1.5% of global electricity in 2024 (about 415 TWh) and are expected to more than double their consumption by 2030 [2]. At the same time, traditional sources of efficiency improvement, such as Moore's law and Dennard scaling, are slowing down [3, 4], while gains from infrastructure optimizations approach diminishing returns [5, 6].

Containerized workloads form an increasingly large share of this growing footprint. They provide lightweight and scalable deployment mechanisms [7], and Kubernetes has become the dominant platform for orchestrating such workloads at scale. Despite their operational benefits and generally low overhead [8], containers make energy transparency difficult. Their shared-resource architecture and multi-layer abstraction obscure the relationship between individual workloads and the physical energy they consume, which complicates research on energy efficiency. As interest in energy-aware computing grows, accurate and fine-grained measurement becomes essential for research, performance engineering, and scheduling. Existing tools offer useful approximations but rely heavily on heuristic models and asynchronous telemetry, which limits their validity and reproducibility.

Kepler[9], a CNCF-backed project under active development, has become a widely adopted energy observability tool for Kubernetes clusters. Its design prioritizes portability, low overhead, and safe deployment across diverse environments, which makes it highly suitable for operational use at scale. A detailed analysis of its methodology and design philosophy is provided in Appendix A. In this context, Tycho is not intended to compete with Kepler or replace it. Instead, Tycho explores a complementary space: the upper bound of what measurement accuracy is achievable when higher-privilege monitoring methods, synchronized server-wide telemetry, and more fine-grained event-time alignment are permitted. While such techniques are often impractical for general-purpose deployments, they offer substantial value for research settings and large-scale R&D environments that require precise and reproducible energy measurements. Tycho is an attempt to fill this accuracy-focused niche by implementing methods that prioritize measurement fidelity over broad deployability.

1.2 Challenges and Limitations of Existing Approaches

Current approaches to workload-level energy estimation provide useful insights but remain limited in accuracy and reproducibility. Tools such as Kepler and Scaphandre[10] correlate resource utilization metrics with node-level energy telemetry, yet they operate under constraints that make fine-grained attribution difficult. Many systems depend on asynchronous and low-frequency sampling, rely on heterogeneous telemetry sources with differing clock domains, or employ heuristic models that introduce uncertainty. Purpose-built, high-accuracy measurement hardware exists, but its cost and operational overhead prevent widespread adoption in production environments. As a result, most available solutions offer high-level trends rather than precise, time-aligned measurements required for rigorous analysis. A more detailed discussion of these limitations is provided in § **??** and further expanded in Appendix A.

1.3 Problem Context

Modern container platforms such as Kubernetes run large numbers of heterogeneous, short-lived workloads on shared physical infrastructure. In multi-tenant clusters, containers continuously start, stop, migrate, and compete for CPU, memory, storage, and I/O resources. This dynamism complicates any attempt to determine how much energy a specific workload is responsible for.

At the hardware level, telemetry relevant to energy measurement is fragmented across multiple subsystems. Interfaces such as Intel RAPL, Redfish power sensors, GPU telemetry, and eBPF-based utilization metrics differ in sampling frequency, latency, and clock domain. Their values may arrive asynchronously and with inconsistent timing, making direct alignment difficult. Reliable attribution requires correlating these subsystems to a common time base, which existing platforms do not provide.

Kubernetes adds another layer of complexity. Processes are encapsulated inside containers, which in turn map to pods, cgroups, and namespaces that evolve over time. Accurate attribution therefore depends on robust tracking of process-to-container relationships and on resolving the timing mismatch between telemetry sources and the rapid state changes within the cluster. These combined factors make precise, container-level energy attribution a challenging problem.

1.4 Problem Statement

1.5 Goals of This Thesis

1.6 Research Questions

1.7 Contributions

1.8 Positioning Within Previous Work

This thesis builds directly on two earlier specialization projects completed as part of the Master's program. The first project (VT1) focused on the practical deployment

of a bare-metal Kubernetes environment for energy efficiency research. It provided a fully automated setup procedure for cluster installation, persistent storage, monitoring infrastructure, and stress-test workloads, including all required configuration and connectivity. VT1 initially deployed Kepler as its energy monitoring component but has since been adapted to install Tycho instead, enabling rapid provisioning of complete measurement and evaluation environments. The resulting *PowerStack* repository [1] works in conjunction with this thesis by providing the operational foundation on which Tycho can be deployed and evaluated, and is documented in Appendix B.

The second project (VT2) examined the theoretical foundations of energy measurement in modern servers and containerized environments. It analyzed hardware-level telemetry interfaces, attribution challenges, and the strengths and limitations of existing tools, providing a structured overview of the state of the art. VT2 identified the methodological gaps that hinder precise workload-level energy attribution, many of which motivate the design choices in Tycho. A detailed version of this analysis is included in Appendix A.

The present thesis consolidates these two lines of work. It integrates the practical insights from VT1 with the theoretical findings of VT2 and develops Tycho as an accuracy-first system for server-wide energy measurement and workload attribution. In this role, the thesis serves as the point where the implementation, methodological foundations, and design philosophy come together into a coherent system.

1.9 Scope and Boundaries

1.10 Target Audience

1.11 Origin of the Name “Tycho”

1.12 Methodological Approach

1.13 Ethical and Sustainability Considerations

1.14 Thesis Structure

1.15 System Environment for Development, Build and Debugging

This section documents the environment used to develop, build, and debug *Tycho*; detailed guides live in [11].

1.15.1 Host Environment and Assumptions

All development and debugging activities for *Tycho* were performed on bare-metal servers rather than virtualized instances. Development matched the evaluation target and preserved access to hardware telemetry such as RAPL, NVML, and BMC Redfish. The host environment consisted of Lenovo ThinkSystem SR530 servers (Xeon Bronze 3104, 64 GB DDR4, SSD+HDD, Redfish-capable BMC).

The systems ran Ubuntu 22.04 with a Linux 5.15 kernel. Full root access was available and required in order to access privileged interfaces such as eBPF. Kubernetes was installed directly on these servers using PowerStack[1], and served as the platform for deploying and testing *Tycho*. Access was via VPN and SSH within the university network.

1.15.2 Build Toolchain

Two complementary workflows are used: a dev path (local build, run directly on a node for interactive debugging) and a deploy path (build a container image, push to GHCR, deploy as a privileged DaemonSet via *PowerStack*).

1.15.2.1 Local builds

The implementation language is Go, using `go version go1.25.1` on `linux/amd64`. The `Makefile` orchestrates routine tasks. The target `make build` compiles the exporter into `_output/bin/<os>_<arch>/kepler`. Targets for cross builds are available for `linux/amd64` and `linux/arm64`. The build injects version information at link time through `LDFLAGS` including the source version, the revision, the branch, and the build platform. This supports traceability when binaries or images are compared during experiments.

1.15.2.2 Container images

Container builds use Docker Buildx with multi arch output for `linux/amd64` and `linux/arm64`. Images are pushed to the GitHub Container Registry under the project repository. For convenience there are targets that build a base image and optional variants that enable individual software components when required.

1.15.2.3 Continuous integration

GitHub Actions produces deterministic images with an immutable commit-encoded tag, a time stamped dev tag, and a latest for `main`. Builds are triggered on pushes to the main branches and on demand. Buildx cache shortens builds without affecting reproducibility.

1.15.2.4 Versioning and reproducibility

Development proceeds on feature branches with pull requests into `main`. Release images are produced automatically for commits on `main`. Development images are produced for commits on `dev` and for feature branches when needed. Dependency management uses Go modules with a populated `vendor/` directory. The files `go.mod` and `go.sum` pin the module versions, and `go mod vendor` materializes the dependency tree for offline builds.

1.15.3 Debugging Environment

The debugger used for *Tycho* is **Delve** in headless mode with a Debug Adapter Protocol listener. This provides a stable front end for interactive sessions while the debugged process runs on the target node. Delve was selected because it is purpose built for Go, supports remote attach, and integrates reliably with common editors without altering the build configuration beyond standard debug symbols.

1.15.3.1 Remote debugging setup

Debug sessions are executed on a Kubernetes worker node. The exporter binary is started under Delve in headless mode with a DAP listener on a dedicated TCP port. The workstation connects over an authenticated channel. In practice an SSH tunnel is used to forward the listener port from the node to the workstation. This keeps the debugger endpoint inaccessible from the wider network and avoids additional access controls on the cluster. To prevent metric interference the node used for debugging excludes the deployed DaemonSet, so only the debug instance is active on that host.

1.15.3.2 Integration with the editor

The editor is configured to attach through the Debug Adapter Protocol. In practice a minimal launch configuration points the adapter at the forwarded listener. Breakpoints, variable inspection, step control, and log capture work without special handling. No container specific extensions are required because the debugged process runs directly on the node.

The editor attaches over the SSH-forwarded DAP port; the inner loop is build locally with `make`, launch under Delve with a DAP listener, attach via SSH, inspect, adjust, repeat. When the goal is to validate behavior in a cluster setting rather than to step through code, the deploy oriented path is used instead. In that case the image is built and pushed, and observation relies on logs and metrics rather than an attached debugger.

1.15.3.3 Limitations and challenges

Headless remote debugging introduces some constraints. Interactive sessions depend on network reachability and an SSH tunnel, which adds a small amount of latency. The debugged process must retain the privileges needed for eBPF and access to hardware counters, which narrows the choice of where to run sessions on multi tenant systems. Running a second exporter in parallel on the same node would distort measurements, which is why the DaemonSet is excluded on the debug host. Container based debugging is possible but less convenient given the need to coordinate with cluster security policies. For these reasons, most active debugging uses a locally built binary that runs directly on the node, while container based deployments are reserved for integration tests and evaluation runs.

1.15.4 Supporting Tools and Utilities

1.15.4.1 Configuration and local orchestration

A lightweight configuration file `config.yaml` consolidates development toggles that influence local runs and selective deployment. Repository scripts read this file and translate high level options into concrete command line flags and environment variables for the exporter and for auxiliary processes. This keeps day to day operations consistent without editing manifests or code, and aligns with the two workflows in § 1.15.2. Repository scripts map configuration keys to explicit flags for local runs, debug sessions, and ad hoc deploys.

1.15.4.2 Container, cluster, and monitoring utilities

Supporting tools: Docker, kubectl, Helm, k3s, Rancher, Ansible, Prometheus, Grafana. Each is used only where it reduces friction, for example Docker for image builds, kubectl for interaction, and Prometheus/Grafana for observability.

1.15.5 Relevance and Limitations

1.15.5.1 Scope and contribution

The development, build, and debugging environment described in § 1.15.2 and § 1.15.3 is enabling infrastructure rather than a scientific contribution. Its purpose is to make modifications to *Tycho* feasible and to support evaluation, not to advance methodology in software engineering or tooling.

Documenting the environment serves reproducibility and auditability. A reader can verify that results were obtained on bare-metal with access to the required telemetry, and can reconstruct the build pipeline from source to binary and container image. The references to the repository at the start of this section in § 1.15 provide the operational detail that is intentionally omitted from the main text.

1.15.5.2 Boundaries and omissions

Installation steps, editor-specific configuration, system administration, security hardening, and multi tenant policy are out of scope; concrete commands live in the repository. Where concrete commands matter for reproducibility they are available in the repository documentation cited in § 1.15.

1.16 eBPF-collector-Based CPU Time Attribution

1.16.1 Scope and Motivation

The kernel-level `eBPF` subsystem in Tycho provides the foundation for process-level energy attribution. It captures CPU scheduling, interrupt, and performance-counter events directly inside the Linux kernel, translating them into continuous measurements of CPU ownership and activity. All higher-level aggregation and modeling occur in userspace; this section therefore focuses exclusively on the in-kernel instrumentation and the data it exposes.

Kepler's original `eBPF` design offered a coarse but functional basis for collecting CPU time and basic performance metrics. Its `sched_switch` tracepoint recorded process runtime, while hardware performance counters supplied instruction and cache data. However, the sampling cadence and aggregation logic were controlled from userspace, producing irregular collection intervals and temporal misalignment with energy readings. Kepler also treated all CPU time as a single undifferentiated category, omitting explicit representation of idle periods, interrupt handling, and kernel threads. As a result, a portion of the processor's activity (often significant under I/O-heavy workloads) remained unaccounted for in energy attribution.

Tycho addresses these limitations through a refined kernel-level design. New tracepoints capture hard and soft interrupts, while extended per-CPU state tracking distinguishes between user processes, kernel threads, and idle execution. Each CPU

maintains resettable bins that accumulate idle and interrupt durations within well-defined time windows, providing temporally bounded activity summaries aligned with energy sampling intervals. Cgroup identifiers are refreshed at every scheduling event to maintain accurate container attribution, even when processes migrate between control groups. The result is a stable, low-overhead data source that describes CPU usage continuously and with sufficient granularity to support fine-grained energy partitioning in the subsequent analysis.

1.16.2 Baseline and Architecture Overview

Kepler's kernel instrumentation consisted of a compact set of eBPF programs that sampled process-level CPU activity and a few hardware performance metrics. The core tracepoint, `tp_btf/sched_switch`, captured context switches and estimated per-process runtime by measuring the on-CPU duration between successive events. Complementary probes monitored page cache access and writeback operations, providing coarse indicators of I/O intensity. Hardware performance counters (CPU cycles, instructions, and cache misses) were collected through `perf_event_array` readers, enabling approximate performance characterization at the task level.

While effective for general profiling, this setup lacked the temporal resolution and system coverage required for precise energy correlation. The sampling process was driven entirely from userspace, leading to irregular collection intervals, and idle or interrupt time was never observed directly. Consequently, CPU utilization appeared complete only from a process perspective, leaving kernel and idle phases invisible to the measurement pipeline.

Tycho extends this architecture into a continuous kernel-side monitoring system. Each CPU maintains an independent state structure recording its current task, timestamp, and execution context. This allows uninterrupted accounting of CPU ownership, even between user-space scheduling events. New tracepoints for hard and soft interrupts measure service durations directly in the kernel, ensuring that all processor activity (user, kernel, or idle) is captured. Dedicated per-CPU bins accumulate these times within fixed analysis windows, which the userspace collector periodically reads and resets. Process-level metrics are stored in an LRU hash map, while hardware performance counters remain integrated via existing PMU readers.

In contrast to Kepler's snapshot-based sampling, Tycho's userspace collector consolidates all per-process and per-CPU deltas from the kernel maps once per polling interval into a single tick. This tick-based aggregation provides deterministic timing, reduces memory pressure, and guarantees temporal consistency across heterogeneous metric sources. Data therefore flows linearly from tracepoints to per-CPU maps and onward to the collector, forming a continuous and low-overhead measurement path that supports precise, time-aligned energy attribution.

1.16.3 Kernel Programs and Data Flow

Tycho's eBPF subsystem consists of a small set of tracepoints and helper maps that together maintain a continuous record of CPU activity. Each program updates per-CPU or per-task data structures in response to kernel events, ensuring that all processor time is accounted for across user, kernel, and idle contexts. The kernel side is event-driven and self-contained; aggregation into time-bounded ticks occurs later in userspace.

Scheduler Switch The central tracepoint, `tp_btf/sched_switch`, triggers whenever the scheduler replaces one task with another. It computes the elapsed on-CPU time of the outgoing process and updates its entry in the `processes` map, which stores cumulative runtime, hardware-counter deltas, and classification metadata such as `cgroup_id`, `is_kthread`, and command name. Hardware counters for instructions, cycles, and cache misses are read from preconfigured PMU readers at this moment, keeping utilization metrics temporally aligned with task execution. Each CPU also maintains a lightweight `cpu_state` structure that records the last timestamp, currently active PID, and task type. When the idle task (PID 0) is scheduled, this structure accumulates idle time locally, allowing continuous accounting even between user-space collection intervals. At polling time, the userspace collector drains these maps atomically, computing per-process deltas since the previous read and bundling all results into a single tick that represents the complete scheduler activity for that interval.

Interrupt Handlers To capture system activity outside user processes, Tycho introduces tracepoints for hard and soft interrupts. Pairs of entry and exit hooks (`irq_handler_entry,exit` and `softirq_entry,exit`) measure the time spent in each category by recording timestamps in the per-CPU state and adding the resulting deltas to dedicated counters. These durations are aggregated in `cpu_bins`, a resettable per-CPU array that also stores idle time. At each collection cycle, the userspace `bpfCollector` drains and resets these bins, incorporating their totals into the tick structure alongside the per-process deltas. This design maintains continuous coverage of kernel activity while preserving strict temporal alignment between CPU-state transitions and energy sampling.

Page-Cache Probes Kepler's original page-cache hooks (`fexit/mark_page_accessed` and `tp/writeback_dirty_folio`) are preserved. They increment per-process counters for cache hits and writeback operations, serving as indicators of I/O intensity rather than direct power consumption. These counters are read and reset as part of the same tick aggregation that handles scheduler and interrupt data.

Supporting Maps and Flow All high-frequency updates occur in per-CPU or LRU hash maps to avoid contention. `pid_time_map` tracks start timestamps for active threads, enabling precise runtime computation during context switches. `processes` holds per-task aggregates, while `cpu_states` and `cpu_bins` manage temporal accounting per core. PMU event readers for cycles, instructions, and cache misses remain shared with Kepler's implementation. At runtime, data flows from tracepoints to these maps and is drained periodically by the userspace collector, which consolidates the deltas into a single per-tick record before storing it in the ring buffer. This batched extraction forms a deterministic, lock-free telemetry path from kernel to analysis, ensuring high-frequency accuracy without per-event synchronization overhead.

1.16.4 Collected Metrics

The kernel `eBPF` subsystem exports a defined set of metrics describing CPU usage at process and system levels. These values are aggregated in kernel maps and periodically retrieved by the userspace collector for time-aligned energy analysis. Table 1.1 summarizes all metrics grouped by category.

Metric	Source hook	Description
Time-based metrics		
Process runtime	`tp_btf/sched_switch`	Per process. Elapsed on-CPU time accumulated at context switches.
Idle time	Derived from `sched_switch`	Per CPU. Time with no runnable task (PID 0).
IRQ time	`irq_handler_{entry,exit}`	Per CPU. Duration spent in hardware interrupt handlers.
SoftIRQ time	`softirq_{entry,exit}`	Per CPU. Duration spent in deferred kernel work.
Hardware-based metrics		
CPU cycles	PMU (`perf_event_array`)	Per process. Retired CPU cycle count during task execution.
Instructions	PMU (`perf_event_array`)	Per process. Retired instruction count.
Cache misses	PMU (`perf_event_array`)	Per process. Last-level cache misses; indicator of memory intensity.
Classification and enrichment metrics		
Cgroup ID	`sched_switch`	Per process. Control group identifier for container attribution.
Kernel thread flag	`sched_switch`	Per process. Marks kernel threads executing in system context.
Page cache hits	`mark_page_accessed`	Per process. Read or write access to cached pages; proxy for I/O activity.
IRQ vectors	`softirq_entry`	Per CPU. Frequency of specific soft interrupt vectors.

TABLE 1.1: Metrics collected by the kernel `eBPF` subsystem.

All metrics are aggregated once per polling interval into a single userspace tick that contains per-process and per-CPU deltas. This tick-based representation replaces the former per-sample storage model, ensuring temporal consistency across metrics while retaining the semantics listed above.

Together these metrics form a coherent description of CPU activity. Time-based data quantify ownership of processing resources, hardware counters capture execution intensity, and classification attributes link activity to its origin. This dataset serves as the kernel-level foundation for energy attribution and higher-level modeling in userspace.

1.16.5 Integration with Energy Measurements

The data exported from the kernel define how CPU resources are distributed among processes, kernel threads, interrupts, and idle periods during each observation window. When combined with energy readings obtained over the same interval, these temporal shares provide the basis for proportional energy partitioning. Instead of relying on statistical inference or coarse utilization averages, Tycho attributes energy according to directly measured CPU ownership.

Each collection tick consolidates all per-process runtime and performance-counter deltas together with per-CPU idle and interrupt bins. The sum of these components represents the total active time observed by the processor during that tick, matching the energy sample boundaries defined by the timing engine. This strict temporal alignment ensures that every joule of measured energy can be traced to a specific class of activity—user workload, kernel service, or idle baseline. Through this mechanism, the `eBPF` subsystem provides the precise temporal structure required for fine-grained, container-level energy attribution in the subsequent analysis stages.

1.16.6 Efficiency and Robustness

The kernel instrumentation is designed to operate continuously with negligible system impact while ensuring correctness across kernel versions. All high-frequency data reside in per-CPU maps, eliminating cross-core contention and locking. Each processor updates only its local entries in `cpu_states` and `cpu_bins`, while per-task data are stored in a bounded LRU hash that automatically removes inactive entries. Arithmetic within tracepoints is deliberately minimal (timestamp subtraction and counter increments only) so that the added latency per event remains near the measurement noise floor.

Userspace retrieval employs batched `BatchLookupAndDelete` operations, reducing system-call overhead and maintaining constant latency regardless of map size. Hardware counters are accessed through pre-opened `perf_event_array` readers managed by the kernel, avoiding repeated setup costs. Each polling interval consolidates the collected deltas into a single userspace tick, ensuring deterministic timing and consistent aggregation across all CPUs. This architecture allows the subsystem to record thousands of context switches per second while keeping CPU overhead low.

Correctness is maintained through several safeguards. CO-RE (Compile Once, Run Everywhere) field resolution protects the program from kernel-version differences in `task_struct` layouts. Cgroup identifiers are refreshed only for the newly scheduled task, ensuring accurate container labeling even when group membership changes. The idle task (PID 0) and kernel threads are handled explicitly to prevent user-space misattribution, and the resettable bin design enforces strict temporal separation between collection ticks. Together, these measures yield a stable and version-tolerant tracing layer that can run indefinitely without producing inconsistent or overlapping tick data.

1.16.7 Limitations and Future Work

Although the extended `eBPF` subsystem provides comprehensive temporal coverage of CPU activity, several limitations remain. Its precision is ultimately bounded by the granularity of available energy telemetry, as energy readings must be averaged over fixed collection intervals to remain stable. Within shorter ticks, power fluctuations introduce noise that limits the accuracy of direct attribution.

The current implementation also omits processor C-state and frequency information. While idle and active time are distinguished, variations in power state and dynamic frequency scaling are not yet represented in the collected data. Including tracepoints

such as `power:cpu_idle` and `power:cpu_frequency` would enable finer correlation between CPU state transitions and power usage. Additionally, very short-lived processes may terminate and be removed from the LRU map before the next tick is collected, leading to a slight underrepresentation of transient workloads.

1.17 GPU Collector Integration

1.17.1 Introduction and Motivation

Accelerators are increasingly responsible for the energy footprint of modern compute workloads. To attribute this consumption to containerized applications with high temporal accuracy, Tycho must incorporate GPU telemetry into the same unified timing model used for RAPL, eBPF, and Redfish domains (§ 1.20). Achieving this integration is challenging: GPU drivers do not expose continuous measurements but publish telemetry at discrete, hardware-dependent intervals. If these intervals are not respected, sampling quickly suffers from aliasing, redundant reads, and temporal drift across subsystems, as well as imprecise timing.

NVIDIA's telemetry interfaces further complicate accurate measurement. The widely used `nvmlDeviceGetPowerUsage` function reports a *one-second trailing average*[12], not the instantaneous power required for sub-second energy attribution. High-frequency power samples are available only through specialised field APIs. Cumulative energy counters (when present) provide authoritative publish boundaries, but they are absent on many devices, including consumer GPUs and MIG configurations. Process-level telemetry is even more restrictive: NVML aggregates utilisation over caller-specified wall-clock windows and provides no information about the device's internal publish cadence.

Because of these structural limitations, fixed polling intervals or naïve periodic sampling are fundamentally insufficient. Accurate attribution requires that Tycho (i) infer the GPU's implicit publish cadence, (ii) align its sampling with this cadence, and (iii) integrate both device- and process-level telemetry into the global measurement timeline without violating the strict monotonic ordering enforced by Tycho's multi-domain ring buffer (§ 1.21.1).

This work introduces two contributions that address these challenges:

- **A phase-aware sampling mechanism** that infers the GPU's hidden publish rhythm and adaptively concentrates polling around predicted update edges. This transforms GPU sampling from periodic polling into a timing-aligned, event-driven process.

- **A unified integration of GPU telemetry** into Tycho's global timebase, producing exactly one `GpuTick` per confirmed hardware update, with timestamps that are directly comparable to all other energy domains.

Together, these mechanisms provide temporally precise, low-latency GPU measurements while respecting the variability and constraints of NVIDIA's telemetry ecosystem. This elevates the GPU subsystem to a first-class energy domain in Tycho and enables accurate container-level attribution in heterogeneous accelerator environments.

1.17.2 Architectural Overview

The GPU collector is organised as a layered subsystem that integrates vendor telemetry, adaptive timing, and unified buffering into a coherent measurement pipeline. Its structure reflects Tycho's core design principles: strict adherence to a monotonic timebase, decoupling of heterogeneous sampling frequencies, and event-driven integration into the platform-wide timing and buffering infrastructure (§ 1.20, § 1.21.1).

At the lowest layer, the collector interfaces with NVIDIA accelerators through a backend abstraction compatible with both `NVML` and `DCGM` ("*Data Center GPU Manager*"). This abstraction handles device enumeration, capability probing, MIG topology inspection, and access to device and process telemetry. The collector does not assume uniform backend capabilities: cumulative energy counters, instantaneous power fields, and process-level utilisation may or may not be available depending on hardware generation and configuration.

Above this backend, the collector exposes two measurement paths:

- **Device path.** Retrieves power, utilisation, frequency, thermal, and memory metrics for all devices and MIG instances. These values describe the instantaneous operational state of the accelerator.
- **Process path.** Aggregates per-process utilisation over a backend-defined wall-clock window. This enables multi-tenant attribution but is inherently retrospective and independent of the device's internal publish cadence.

Both paths feed into a shared sampling layer governed by Tycho's timing engine. The device path is triggered by a *phase-aware scheduler* that aligns its polling activity with the driver's implicit publish cadence. The process path is invoked only when a device update is detected, ensuring temporal alignment between instantaneous device measurements and aggregated process data.

The final integration step mirrors all other Tycho subsystems: each confirmed hardware update is converted into a `GpuTick` structure containing device and (optionally) process snapshots, together with a strictly ordered monotonic timestamp. This tick is emitted into Tycho's multi-domain ring buffer, where it becomes part of the unified energy timeline used for correlation and attribution across eBPF, RAPL, and platform power domains.

Figure 1.1 (later in this section) provides a conceptual overview of this pipeline, illustrating the interaction between the backend interface, the phase-aware sampler, and Tycho's global collection engine.

1.17.3 Phase-Aware Sampling: Conceptual Overview

GPU drivers publish power and utilisation metrics at discrete, device-internal intervals. These updates occur neither continuously nor synchronously with the sampling frequencies required by Tycho's timing engine. Because the driver does not expose its publish cadence directly, a naïve fixed-interval polling strategy risks both *aliasing* (missing updates) and *redundancy* (repeatedly reading identical values). Either effect would distort the temporal alignment of GPU measurements with the rest of Tycho's energy domains.

To avoid this, the GPU collector introduces a *phase-aware sampling* mechanism that infers the driver's implicit publish cadence from observations. The sampler tracks two quantities: an estimated publish period and the phase offset between the device's update rhythm and Tycho's monotonic timebase. By predicting the next likely update moment, the sampler can modulate its polling intensity accordingly:

- **Base mode:** low-frequency polling maintains coarse alignment and detects long-term drift in the publish cadence.
- **Burst mode:** when the current time approaches a predicted update edge, the sampler briefly increases its polling frequency to minimise the latency between the hardware update and Tycho's observation of it.

This adaptive strategy ensures that Tycho reads the device only when a fresh publish is likely to be available. Freshness is determined by comparing each snapshot to the most recent confirmed update, preferably via cumulative energy counters when present, or otherwise via power deltas exceeding a configurable threshold. Only when a new publish is detected does the sampler emit an event.

The resulting behaviour is simple but powerful:

> *Each hardware update produces exactly one* `GpuTick`*, and no tick is emitted unless the device has genuinely updated.*

This one-to-one correspondence is critical for integrating GPU measurements into Tycho's unified energy timeline. It guarantees temporal fidelity, eliminates redundant samples, and ensures that GPU metrics are directly comparable with other measurements obtained under the same timing and buffering semantics.

The next subsection formalises this behaviour by presenting the timing model used to estimate publish periods, track phase offsets, and define the burst window around predicted update edges.

1.17.4 Phase-Aware Timing Model

The timing model enables Tycho to infer the GPU driver's implicit publish cadence and to align sampling with the actual update moments of the hardware. It maintains two quantities derived from confirmed device updates: an estimate of the *publish period* and a *phase offset* relative to Tycho's monotonic clock. This subsection presents the model in a unified mathematical form.

Let $t_{\text{obs},k}$ denote the monotonic timestamp of the k-th confirmed hardware update. From these observations, the sampler derives the period estimate $\hat{T}_k$ and phase estimate $\hat{\phi}_k$.

Period Estimation. Each new inter-update interval

$$\Delta t_k = t_{\text{obs},k} - t_{\text{obs},k-1}$$

provides a direct sample of the device's publish period. To remain robust to jitter caused by DVFS, thermal transitions, or backend noise, the sampler applies an exponential moving average (EMA):

$$\hat{T}_k = (1 - \alpha_T)\,\hat{T}_{k-1} + \alpha_T\,\Delta t_k,$$

with $\alpha_T \in (0, 1)$ controlling the smoothing strength. The resulting estimate is clamped to a stable range derived from Tycho's engine cadence, ensuring predictable behaviour across different GPUs.

Phase Tracking. Given a current period estimate, the expected time of the k-th update is

$$\hat{t}_k = t_{\text{obs},k-1} + \hat{\phi}_{k-1} + \hat{T}_k.$$

The deviation

$$\delta_k = t_{\text{obs},k} - \hat{t}_k$$

represents the phase error. The sampler updates its phase estimate through a second EMA:

$$\hat{\phi}_k = (\hat{\phi}_{k-1} + \alpha_\phi\,\delta_k) \bmod \hat{T}_k,$$

where α_ϕ is a small adaptation constant. This ensures smooth convergence toward the device's true publish rhythm.

Edge Prediction. At an arbitrary time t_{now}, the predicted next update edge is

$$t_{\text{next}} = t_{\text{obs},k} + n \cdot \hat{T}_{k+1} + \hat{\phi}_{k+1},$$

where n is the smallest non-negative integer such that $t_{\text{next}} \geq t_{\text{now}}$. This prediction determines where sampling effort should be concentrated.

Burst Window. To avoid continuous high-frequency polling, the sampler restricts hyperpolling to a narrow window of half-width w around t_{next}:

$$\text{mode}(t_{\text{now}}) = \begin{cases} \text{burst}, & |t_{\text{now}} - t_{\text{next}}| \leq w, \\ \text{base}, & \text{otherwise.} \end{cases}$$

The width w is expressed as a fraction of the calibrated engine cadence, ensuring proportional behaviour across platforms.

Summary. The phase-aware model enables Tycho to infer the GPU's implicit publish cadence solely from observed updates and to align sampling with the device's true update edges. By combining smooth period estimation, adaptive phase correction, and narrow burst windows around predicted publishes, the sampler detects new hardware updates with low latency and emits exactly one `GpuTick` per publish.

Figure 1.1 visualises the behaviour of the phase-aware sampling model introduced above. The top lane represents the hardware's implicit publish sequence; the middle lane shows Tycho's adaptive polling pattern during both calibration and the phase-locked regime; and the bottom lane shows the resulting GPU ticks, demonstrating the one-to-one mapping between fresh device updates and emitted `GpuTick` events.

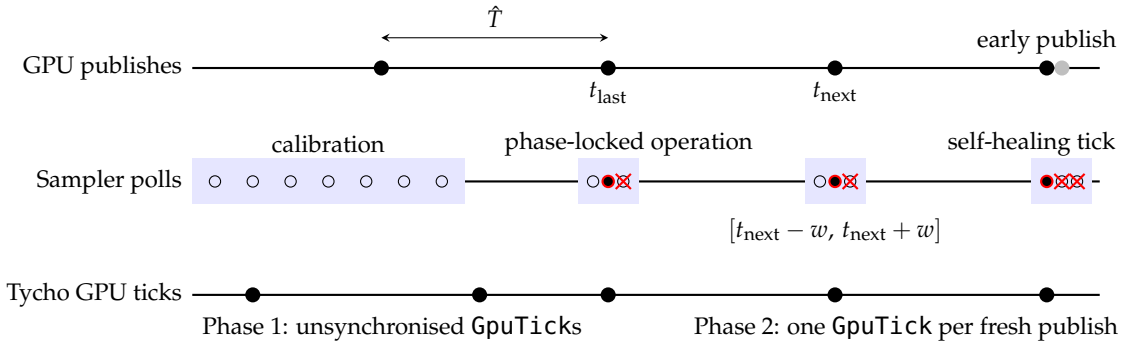


FIGURE 1.1: Phase-aware GPU polling timeline

1.17.5 Event Lifecycle

The GPU collector converts each confirmed hardware update into a monotonically-timestamped `GpuTick` that integrates into Tycho's multi-domain energy timeline. The lifecycle consists of five stages: polling, device acquisition, optional process acquisition, freshness detection, and tick emission.

1. Poll Initiation. Polling is triggered solely by the phase-aware scheduler (§ 1.17.3, § 1.17.4). Base-mode polls track long-term cadence drift; burst-mode polls densely probe the vicinity of predicted update edges. Each poll receives a monotonic timestamp t_{now} that anchors the resulting event.

2. Device Snapshot Acquisition. A poll retrieves device-level telemetry for all GPUs and MIG instances, capturing power, utilisation, clocks, thermals, and memory state. All values reflect the device's instantaneous condition at t_{now} and form a consistent cross-device snapshot of the accelerator subsystem.

3. Optional Process Snapshot Acquisition. If available, process-level telemetry is sampled over a backend-defined wall-clock window (§ 1.17.6). Although retrospective, these samples are associated with the same monotonic timestamp as the device snapshot, ensuring that device and process data remain correlated without temporal ambiguity.

4. Freshness Determination. The collector compares the new device snapshot with the most recent confirmed update. Cumulative energy counters, when available, serve as the authoritative freshness signal; otherwise Tycho uses a power-delta threshold to avoid counting noise as updates. Only fresh snapshots update the period and phase estimators and proceed to the next stage.

5. Tick Emission. A fresh observation is converted into a `GpuTick` containing device and (optional) process snapshots and the timestamp t_{now}. The tick is then delivered to Tycho's multi-domain ring buffer (§ 1.21.1). If no fresh update is detected, the poll produces no tick, ensuring that the GPU timeline faithfully reflects the hardware's publish cadence.

Summary. The event lifecycle ensures that GPU telemetry is sampled only when meaningful, timestamped consistently with Tycho's timebase, and integrated without blocking or duplication. Each hardware update generates exactly one `GpuTick`, providing a precise, causally ordered input to Tycho's cross-domain energy attribution pipeline.

1.17.6 Per-Process Telemetry Window

Device-level metrics describe the instantaneous state of each GPU, but many applications require attributing accelerator activity to individual processes or containers. NVIDIA's interfaces provide such information only in the form of *aggregated utilisation over a caller-specified time window*. Correctly selecting and interpreting this window is essential for obtaining meaningful per-process data and for aligning process-level records with the device-level timeline maintained by Tycho.

Wall-Clock Semantics. Unlike device publishes, which occur on the GPU's internal cadence, NVIDIA's per-process APIs integrate utilisation over a duration supplied by the caller. These interfaces expect a *wall-clock* interval, expressed in milliseconds, rather than a duration derived from Tycho's monotonic timebase. The distinction is crucial: Tycho's monotonic clock operates on an internal quantum chosen to support high-resolution scheduling (§ 1.20), but this quantum has no defined relationship to real elapsed time. Using monotonic differences directly would produce windows that are several orders of magnitude too short, yielding incomplete utilisation samples.

For this reason, Tycho maintains a separate wall-clock origin for each GPU or MIG instance. Whenever process telemetry is requested, the duration since the last successful query is computed using wall-clock time, ensuring that the backend receives a true real-time interval.

Window Derivation. For each owner (physical GPU or MIG instance), Tycho records the timestamp $t^{(i)}_{\text{last}}$ of the most recent successful process query. When a new query occurs at time $t^{(i)}_{\text{now}}$, the raw duration

$$\Delta t^{(i)}_{\text{raw}} = t^{(i)}_{\text{now}} - t^{(i)}_{\text{last}}$$

is transformed according to backend expectations:

1. *Clamping.* The duration is restricted to a safe range $\Delta t_{\min} \leq \Delta t^{(i)} \leq \Delta t_{\max}$ to avoid zero-length or excessively long sampling windows.

2. *Millisecond granularity.* NVIDIA's process APIs accept durations in whole milliseconds. Tycho therefore rounds the clamped value up to the next full millisecond to prevent systematic underestimation of utilisation.

After a successful query, the wall-clock origin is updated to $t^{(i)}_{\text{last}} \leftarrow t^{(i)}_{\text{now}}$, establishing continuity across successive sampling windows.

Temporal Alignment with Device Updates. Even though per-process telemetry describes accumulated activity rather than a snapshot, Tycho ensures that all process samples remain aligned with the device timeline. Each process record is associated

with the device-level timestamp of the poll that triggered the query. If a device has never produced a fresh update, the collector uses the timestamp of the most recent device tick as the initial origin for its process window. This guarantees that device and process metrics are linked to the same global temporal reference and can be fused without interpolation.

Backend Variability and Robustness. Process-level support varies widely across NVIDIA hardware and software stacks. DCGM-capable systems typically expose high-quality, high-resolution utilisation data, whereas NVML-only systems (particularly consumer GPUs) may provide limited or noisy information. Tycho's design accommodates these differences gracefully: when a process query fails, the wall-clock origin is still advanced to prevent tight retry loops, and device-level sampling proceeds unaffected. This ensures stable behaviour even in mixed configurations where only a subset of devices expose meaningful per-process telemetry.

Summary. By separating wall-clock process windows from monotonic device timestamps and carefully aligning both within Tycho's timing architecture, the GPU collector provides process-level telemetry that is semantically correct, temporally consistent, and robust to backend limitations. This separation of concerns is essential for accurate multi-tenant attribution in heterogeneous accelerator environments.

1.17.7 Collected Metrics

The GPU collector reports two complementary categories of telemetry that together describe both the instantaneous state of each accelerator and the distribution of GPU activity across processes. All metrics are incorporated into a unified `GpuTick` structure and timestamped under Tycho's monotonic timebase, ensuring direct comparability with other domains.

Device-Level Metrics. Device and MIG-level metrics capture the operational state of the accelerator at the moment Tycho detects a fresh hardware update. These values include power, utilisation, memory usage, thermal data, and clock frequencies, along with backend-specific fields such as instantaneous power samples or cumulative energy counters. Cumulative energy, when available, is used as the authoritative indicator of publish boundaries and therefore plays a central role in the timing model and freshness detection. Due to a tendency of NVIDIA GPUs to still produce (invalid) values when queried for cumulative energy, the actual availability of correct cumulative energy-metrics is verified during the initial calibration.

Process-Level Metrics. Process-level metrics describe the aggregated utilisation of individual processes over the backend-defined wall-clock window (§ 1.17.6). They enable multi-tenant attribution by associating GPU activity with specific applications, containers, or pods. Because these values represent accumulated work rather than an instantaneous snapshot, they are paired with the device-level timestamp of the triggering poll, ensuring temporal consistency within Tycho's unified timeline.

Tables 1.2 and 1.3 summarise the metrics collected at both levels.

Metric	Unit	Description
Utilisation metrics		
`SMUtilPct`	%	Streaming multiprocessor (SM) utilisation.
`MemUtilPct`	%	Memory controller utilisation.
`EncUtilPct`	%	Hardware video encoder utilisation.
`DecUtilPct`	%	Hardware video decoder utilisation.
Energy and thermal metrics		
`PowerMilliW`	mW	Instantaneous power via NVML/DCGM (1s average).
`InstantPowerMilliW`	mW	High-frequency instantaneous power from NVIDIA field APIs.
`CumEnergyMilliJ`	mJ	Cumulative energy counter (preferred freshness signal).
`TempC`	°C	GPU temperature.
Memory and frequency metrics		
`MemUsedBytes`	bytes	Allocated framebuffer memory.
`MemTotalBytes`	bytes	Total framebuffer memory.
`SMClockMHz`	MHz	SM clock frequency.
`MemClockMHz`	MHz	Memory clock frequency.
Topology and metadata		
`DeviceIndex`	–	Numeric device identifier.
`UUID`	–	Stable device UUID.
`PCIBusID`	–	PCI bus identifier.
`IsMIG`	–	Indicates a MIG instance.
`MIGParentID`	–	Parent device index for MIG instances.
`Backend`	–	Backend type (`NVML` or `DCGM`).

TABLE 1.2: Device- and MIG-level metrics collected by the GPU subsystem.

Metric	Unit	Description
`Pid`	–	Process identifier.
`ComputeUtil`	%	Per-process SM utilisation aggregated over the query window.
`MemUtil`	%	Per-process memory controller utilisation.
`EncUtil`	%	Per-process encoder utilisation.
`DecUtil`	%	Per-process decoder utilisation.
`GpuIndex`	–	Device or MIG instance to which the sample belongs.
`GpuUUID`	–	Corresponding device UUID.
`TimeStampUS`	µs	Backend timestamp associated with the utilisation record.
MIG metadata (when applicable)		
`GpuInstanceID`	–	MIG GPU instance identifier.
`ComputeInstanceID`	–	MIG compute-instance identifier.

TABLE 1.3: Process-level metrics collected over a backend-defined time window.

1.17.8 Configuration Parameters

The GPU collector exposes only a minimal set of configuration parameters. In contrast to traditional monitoring systems that require hand-tuned polling intervals, Tycho derives the parameters of the phase-aware sampler directly from the engine cadence calibrated during system startup (§ 1.20). This ensures that GPU sampling

inherits the same temporal consistency as all other energy domains and remains robust across heterogeneous hardware.

The configuration governs three tightly coupled aspects of the sampling mechanism:

- **Cadence bounds.** The initial estimate of the GPU publish period, as well as its minimum and maximum permissible values, are expressed as simple fractions of the engine cadence. This constrains the period estimator to a stable range without relying on device-specific heuristics.
- **Polling intervals.** Both base-mode and burst-mode polling frequencies are derived from fixed ratios of the engine cadence. As a result, Tycho polls aggressively only when a publish is predicted without requiring manual tuning.
- **Burst-window width.** The half-width of the burst window around t_{next} is likewise tied to the engine cadence. This determines how narrowly the sampler focuses its hyperpolling effort around predicted publish edges.

Because all parameters scale with the calibrated cadence, the sampler adapts automatically to different GPU generations, backend behaviours, and platform timing characteristics. No user-facing configuration is required; temporal correctness follows directly from Tycho's system-wide timing model.

1.17.9 Robustness and Limitations

The GPU collector is designed to operate reliably across heterogeneous hardware, backend capabilities, and driver behaviours. Its phase-aware sampling, decoupled event queue, and unified timebase ensure that GPU telemetry integrates cleanly with Tycho's multi-domain measurement framework. Nevertheless, several structural constraints in NVIDIA's telemetry ecosystem define the practical limits of what can be inferred and with what temporal precision.

Backend Variability. The capabilities of `NVML` and `DCGM` differ significantly across GPU generations and product classes. Datacenter GPUs typically expose cumulative energy counters, high-frequency instant power fields, and stable process-level utilisation, while consumer GPUs often lack cumulative energy and provide only coarse utilisation metrics. Tycho handles these differences gracefully (sampling continues even when certain fields are missing) but the quality of the resulting attribution reflects the capabilities of the underlying hardware.

Power Measurement Limitations. The widely used `nvmlDeviceGetPowerUsage` call provides a *one-second trailing average*, which is unsuitable as a high-frequency power signal. Tycho therefore relies on instantaneous power fields (e.g. field 186) when available, and uses cumulative energy counters as the authoritative freshness indicator. On devices lacking both instantaneous fields and cumulative energy, power-based freshness detection becomes less precise, increasing uncertainty in the inferred publish cadence.

Process Attribution Constraints. Process-level utilisation is inherently aggregated over a wall-clock window, since NVIDIA provides no access to per-process instantaneous state. This retrospective design imposes two limitations: (i) spikes shorter

than the sampling window may be attenuated, and (ii) per-process values cannot be aligned to the exact moment of a device publish. Tycho addresses this by using the device-level timestamp to anchor all process records, but the granularity of attribution ultimately depends on backend resolution.

Cadence Inference and Jitter. Because the driver does not expose its publish cadence, Tycho must infer it indirectly. Under conditions of high load, thermal transitions, or DVFS-induced jitter, publish intervals may vary, introducing uncertainty into edge prediction. Tycho's EMA-based estimators maintain stability under such variability, but prediction accuracy is inherently bounded by the noisiness of the underlying telemetry.

Mixed and MIG Configurations. Systems combining MIG and non-MIG devices, or devices with partial telemetry support, may expose inconsistent field availability across accelerators. Cumulative energy counters may exist for some instances but not others; process information may be available only at the parent-device level. Tycho handles these cases through per-device fallbacks and independent cadence models, but the precision of multi-GPU attribution varies with the fidelity of each device's telemetry.

Vendor support scope. The current GPU collector supports only NVIDIA-based accelerators, following the design of Kepler. While this excludes other vendors, it is justifiable: according to market research[13], NVIDIA captured approximately 93% of the server GPU revenue in 2024. Given this dominant share, focusing on NVIDIA hardware is acceptable for the majority of data-centre GPU deployments.

1.18 Redfish Collector Integration

The Redfish collector retrieves node-level power data from the server's Baseboard Management Controller (BMC) via the Redfish API. As an out-of-band source, it complements in-band interfaces such as RAPL by providing an external, hardware-validated view of total system power. Tycho integrates this telemetry into its synchronized measurement framework, ensuring consistent timing and comparability across collectors.

1.18.1 Overview and Objectives

Redfish power metrics are vendor-defined and updated asynchronously, with variable latency and precision. The Tycho implementation therefore focuses on reliability, timing control, and consistent timestamping. All polling, freshness tracking, and temporal alignment are managed centrally by Tycho, allowing Redfish samples to be merged with other data sources for later workload-level energy attribution.

1.18.2 Baseline in Kepler

In Kepler, the Redfish implementation provided a minimal wrapper around the BMC's `/redfish/v1/Chassis/*/Power` endpoint. Its sole purpose was to retrieve aggregated chassis power at a fixed interval and expose it through the node-level energy interface used by the power model. The default polling frequency was

set to 60 seconds, adequate for coarse monitoring but too infrequent for detailed analysis.

At such long intervals, issues like repeated values or timing drift were largely masked by the coarse sampling period. However, the design offered no mechanisms to detect new versus stale data, to associate samples with BMC timestamps, or to align readings precisely with other metrics. The internal background ticker operated independently of other Kepler collectors, providing no unified notion of time or freshness. Kepler's Redfish integration was therefore sufficient for low-resolution system energy reporting, but not designed for higher measurement intervals or fine-grained temporal correlation.

1.18.3 Refactoring and Tycho Extensions

1.18.3.1 Timing Ownership and Polling Control

Tycho removes Kepler's internal ticker and delegates all Redfish polling to its centralized timing engine. To account for the unpredictable nature of BMC update cycles, Tycho introduces an optional adaptive mode governed by `TYCHO_REDFISH_POLL_AUTOTUNE`. When enabled, the collector dynamically infers a suitable polling interval from observed publication gaps, learning the effective refresh frequency of the specific Redfish implementation. When disabled, Tycho performs fixed-interval polling strictly at the user-defined cadence, preserving deterministic operation.

By externalizing timing control, the collector decouples sampling from Redfish's internal pacing, enabling reproducible experiments and consistent temporal correlation with other measurement sources.

1.18.3.2 Header-Based Newness and Sequence Tracking

When polled at higher frequencies, Redfish endpoints often repeat identical payloads until the BMC updates its internal sensors. To avoid redundant samples, Tycho introduces a lightweight newness detection mechanism combining HTTP headers and value comparison.

Each response is inspected for the `ETag` and `Date` headers. If an `ETag` differs from the previously stored value, or if the `Date` timestamp is newer, the sample is treated as fresh. If no header change is observed, Tycho falls back to value-based detection by comparing the reported power against the previous reading. A monotonically increasing `seq` counter is maintained per chassis to mark every distinct update, allowing downstream components to identify repeated or skipped readings unambiguously.

This design provides consistent differentiation between new and stale measurements without requiring vendor-specific heuristics. It also ensures that timestamp alignment and freshness analysis remain reliable even when Redfish responses arrive irregularly or contain repeated values.

1.18.3.3 Heartbeat Mechanism and Freshness Metric

Because Redfish publication intervals can vary considerably between BMC implementations, Tycho introduces a heartbeat mechanism to ensure continuous sample availability. If no new data are received within a configurable timeout, defined by

TYCHO_REDFISH_HEARTBEAT_MAX_GAP_MS, the collector emits a heartbeat sample that reuses the last known power value. This prevents temporal gaps in the time series and maintains a consistent data flow for later energy integration.

Each emitted sample also carries a *freshness* metric, representing the time difference between the Redfish-reported Date header (if present) and the local collection timestamp. This value quantifies the staleness of a reading and allows the analysis layer to account for delayed or buffered updates. In practice, freshness remains below one second on well-behaved BMCs but can increase significantly under heavy load or poor firmware timing.

Together, the heartbeat and freshness metric allow Tycho to stabilize asynchronous Redfish data streams and provide temporal confidence estimates for each sample.

1.18.3.4 Fixed vs Auto Polling Mode

Tycho supports two complementary Redfish polling strategies, selectable via TYCHO_REDFISH_POLL_AUTOTUNE. In *fixed mode* (false), polling occurs strictly at the user-defined cadence TYCHO_REDFISH_POLL_MS. This mode guarantees deterministic timing and is suited for controlled experiments where Redfish irregularities are tolerable or where timing synchronization with other collectors is critical.

When *auto mode* (true) is enabled, the collector dynamically adjusts its internal expectations to match the observed publication rhythm of the BMC. It derives the median inter-arrival time of new Redfish samples and adapts the expected heartbeat gap accordingly. This allows the collector to align its emission behavior with the actual update frequency of the hardware, minimizing redundant polls and improving temporal coherence between samples.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

THIS SECTION NEEDS TO BE UPDATED UPON CALIBRATION PACKAGE COMPLETION
A future calibration module will further refine POLL_MS and related delay parameters based on startup profiling, but this mechanism remains outside the collector itself. Within Tycho, the auto mode provides a self-stabilizing behavior that balances responsiveness with measurement overhead, while the fixed mode ensures reproducibility for benchmark-oriented studies. XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

1.18.4 Collected Metrics

The Redfish collector retrieves instantaneous power readings from the BMC for each physical chassis. Unlike software-based collectors, it provides direct hardware telemetry at node level and does not expose component-level breakdowns. All readings are timestamped, aligned to Tycho's global monotonic clock, and supplemented with freshness and sequence metadata to support temporal correlation. Table 1.4 lists the collected fields.

These metrics provide a coherent node-level view of power consumption with explicit temporal context, forming the hardware baseline for higher-level attribution in later stages of the Tycho pipeline.

Metric	Unit	Description
Primary power metrics		
`PowerWatts`	W	Instantaneous chassis power draw reported by the BMC via `/redfish/v1/Chassis/*/Power`.
`EnergyMilliJ`	mJ	Integrated node energy derived from consecutive power samples (computed downstream).
Temporal and identity metadata		
`ChassisID`	-	Identifier of the chassis or enclosure corresponding to the Redfish endpoint.
`Seq`	-	Incremental counter marking each new reading as determined by header or value changes.
`SourceTime`	s	Original BMC timestamp parsed from the HTTP `Date` header, if available.
`CollectorTime`	s	Local collection time according to Tycho's monotonic clock.
`FreshnessMs`	ms	Time difference between `SourceTime` and `CollectorTime`, indicating sample latency.
Operational context		
`Heartbeat`	flag	Marks a repeated emission when no new BMC data were available within the configured heartbeat interval.
`PollMode`	enum	Indicates whether fixed or auto polling mode was active during sampling.

TABLE 1.4: Metrics collected by the Redfish collector.

1.18.5 Integration and Data Flow

The Redfish collector operates as a passive data source within Tycho's unified collection framework. It queries the BMC through the Redfish API, extracts instantaneous chassis power, and writes each result into a synchronized ring buffer shared with the central engine. Each record carries both system- and collection-time metadata, enabling later temporal alignment during analysis.

This integration layer is deliberately lightweight: the collector's responsibility ends once valid samples are obtained and buffered. All subsequent processing is handled by Tycho's analysis modules. This separation keeps the collector simple, minimizes coupling to higher layers, and isolates potential BMC irregularities from the rest of the system.

1.18.6 Accuracy and Robustness Improvements

Tycho introduces several measures to improve the precision and reliability of Redfish telemetry compared to Kepler. Header-based newness detection ensures that only genuinely updated readings are processed, reducing redundant samples caused by repeated BMC responses. Each reading carries a freshness metric that quantifies its temporal distance from the BMC's internal timestamp, providing explicit visibility into data latency. A lightweight heartbeat mechanism compensates for occasional gaps or stalls in BMC reporting, maintaining continuity in the power time

series without fabricating new information.

These measures collectively enhance stability across heterogeneous Redfish implementations and ensure that all retained samples are both valid and chronologically consistent.

1.18.7 Limitations

Despite its improved design, the Redfish collector remains constrained by the capabilities and responsiveness of the underlying BMC. Sampling frequency is typically limited to one or two seconds, and the precision of reported timestamps varies widely across vendors. No component-level breakdown is available (only total chassis power), restricting fine-grained attribution to software-based collectors such as RAPL or eBPF.

1.19 Configuration Management

1.19.1 Overview and Role in the Architecture

Tycho adopts a simple, centralized configuration layer that is initialized during exporter startup and made globally accessible through typed structures. This layer defines all runtime parameters controlling timing, collection, and analysis behaviour. It serves as the interface between user-defined settings and the internal scheduling and buffering logic described in § 1.20.

The configuration is loaded once at startup, combining defaults, environment variables, and optional overrides passed through Helm or local flags. Its purpose is not to support dynamic reconfiguration, but to provide deterministic, reproducible operation across (experimental) runs. No backward compatibility with previous Kepler versions is maintained.

1.19.2 Configuration Sources

Configuration values can be provided in three ways: first, through a `values.yml` file during Helm installation, second, as command-line flags for local or debugging builds, and third, via predefined environment variables that act as defaults.

During startup, Tycho sequentially evaluates these sources in fixed order— defaults are loaded first, then environment variables, followed by any user-supplied overrides. The resulting configuration is stored in memory and printed once for verification. After initialization, all components reference the same in-memory configuration, ensuring consistent behaviour across collectors and analysis modules.

1.19.3 Implementation and Environment Variables

The configuration implementation in Tycho closely follows the approach used in Kepler v0.9.0. Each configuration key is mapped to an environment variable, which is resolved at startup through dedicated lookup functions. If no variable is set, the corresponding default value is applied. This mechanism enables flexible configuration without external dependencies or complex parsing logic. All variables are read once during initialization, after which they are cached in typed configuration structures. This guarantees consistent operation even if environment variables change

later, since Tycho is not designed for live reconfiguration. The configuration layer is invoked before the collectors and timing engine are instantiated, ensuring that parameters such as polling intervals, buffer sizes, or analysis triggers are available to all components from the first cycle onward.

1.19.3.1 Validation and Normalization at Startup

During initialization, Tycho validates all user inputs and normalizes them to a consistent, safe configuration. First, basic bounds are enforced: the global timebase quantum must be positive, non-negative values are required for all periods and delays, and missing essentials fall back to minimal defaults. Trigger coherence is then checked. If `redfish` is selected while the Redfish collector is disabled, Tycho switches to the timer trigger and ensures a valid interval. Unknown triggers default to `timer`.

All periods and delays are aligned to the global quantum so that scheduling, buffering, and analysis operate on a common time grid. The analysis wait `DelayAfterMs` is raised if needed to cover the longest enabled per-source delay. Buffer sizing is derived from the slowest effective acquisition path (poll period plus delay) and the analysis wait, with a small safety margin. If Redfish is enabled, its heartbeat requirement is included to guarantee coverage. Sanity checks also ensure plausible Redfish cadence and warn if no collectors are enabled. Non-fatal environment hints (for example the RAPL powercap path) are reported at low verbosity.

The result is a single, internally consistent configuration snapshot. Adjustments are announced once at startup to aid reproducibility while avoiding log noise.

1.19.4 Evolution in Newer Kepler Versions

Subsequent Kepler releases (v0.10.0 and later) have replaced the environment-variable system with a unified configuration interface based on CLI flags and YAML files. This modernized approach simplifies configuration management and aligns better with Kubernetes conventions, providing clearer defaults and validation at startup.

Tycho intentionally retains the v0.9.0 model to maintain structural continuity with its experimental foundation. Since configuration handling is not a research focus, adopting the newer scheme would add complexity without scientific benefit. Nevertheless, the newer Kepler design confirms that Tycho's configuration logic can be migrated with minimal effort if long-term maintainability becomes a requirement.

1.19.5 Available Parameters

All parameters are read at startup and remain constant throughout execution. The following table 1.5 summarizes the user-facing configuration variables with their default values and functional scope. Internal or experimental parameters are omitted for clarity.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
CHECK THIS TABLE BEFORE HANDIN, THIS NEEDS CLEANUP
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Variable	Default	Description
Collector enable flags		
TYCHO_COLLECTOR_ENABLE_BPF	true	Enables eBPF-based process metric collection.
TYCHO_COLLECTOR_ENABLE_RAPL	true	Enables RAPL energy counter collection.
TYCHO_COLLECTOR_ENABLE_GPU	true	Enables GPU power telemetry collection.
TYCHO_COLLECTOR_ENABLE_REDFISH	true	Enables Redfish-based BMC power collection.
Timing and delays		
TYCHO_TIMEBASE_QUANTUM_MS	1	Base system quantum (ms) defining the global monotonic time grid.
TYCHO_RAPL_POLL_MS	50	RAPL polling interval (ms).
TYCHO_GPU_POLL_MS	200	GPU telemetry polling interval (ms).
TYCHO_REDFISH_POLL_MS	1000	Redfish polling interval (ms); should be below BMC publish cadence.
TYCHO_RAPL_DELAY_MS	0	Expected delay between workload change and RAPL visibility (ms).
TYCHO_GPU_DELAY_MS	200	Expected delay between workload change and GPU visibility (ms).
TYCHO_REDFISH_DELAY_MS	0	Expected delay between workload change and Redfish visibility (ms).
TYCHO_REDFISH_HEARTBEAT_MAX_GAP_MS	3000	Maximum tolerated gap between consecutive Redfish samples (ms).
Autotuning controls		
TYCHO_RAPL_POLL_AUTOTUNE	true	Enables automatic calibration of RAPL polling interval.
TYCHO_RAPL_DELAY_AUTOTUNE	true	Enables automatic calibration of RAPL delay.
TYCHO_GPU_POLL_AUTOTUNE	true	Enables automatic calibration of GPU polling interval.
TYCHO_GPU_DELAY_AUTOTUNE	true	Enables automatic calibration of GPU delay.
TYCHO_REDFISH_POLL_AUTOTUNE	true	Enables automatic calibration of Redfish polling interval.
TYCHO_REDFISH_DELAY_AUTOTUNE	true	Enables automatic calibration of Redfish delay.
Analysis parameters		
TYCHO_ANALYSIS_TRIGGER	"timer"	Defines analysis trigger: `redfish` or `timer`.
TYCHO_ANALYSIS_EVERY_SEC	15	Interval for timer-based analysis (s).
TYCHO_ANALYSIS_DETECT_LONGEST_DELAY	false	Enables detection of the longest observed metric delay.

TABLE 1.5: User-facing configuration variables available in Tycho.

1.20 Timing Engine

1.20.1 Overview and Motivation

Tycho introduces a dedicated timing engine that replaces the synchronous update loop used in Kepler with an event-driven, per-metric scheduling layer. While the conceptual motivation for this change was discussed in § **??**, its practical purpose is straightforward: to decouple the collection frequencies of heterogeneous telemetry sources and to establish a common temporal reference for subsequent analysis.

Each collector in Tycho (e.g., RAPL, eBPF, GPU, Redfish) operates under its own polling interval and is triggered by an aligned ticker maintained by the timing engine. All tickers share a single epoch (base timestamp) and are aligned to a configurable time quantum, ensuring deterministic phase relationships and bounded drift across all metrics. This architecture allows high-frequency sources to capture fine-grained temporal variation while preserving coherence with slower metrics.

The timing engine thus provides the temporal backbone of Tycho: it defines *when* each collector produces samples and ensures that all samples can later be correlated on a unified, monotonic timeline. Collected samples are pushed immediately into per-metric ring buffers, described in § 1.21, which retain recent histories for downstream integration and attribution.

1.20.2 Architecture and Design

The timing engine is implemented in the `engine.Manager` module. It acts as a lightweight scheduler that governs the execution of all metric collectors through independent, phase-aligned tickers. During initialization, each collector registers its callback function, polling interval, and enable flag with the manager. Once started, the manager creates one aligned ticker per enabled registration and launches each collector in a dedicated goroutine. All tickers share a single epoch, captured at startup, to guarantee deterministic alignment across collectors.

This design contrasts sharply with the global ticker used in Kepler, where a single update loop refreshed all metrics at a fixed interval. In Tycho, each ticker operates at its own cadence, determined by the configured polling period of the respective collector. For instance, RAPL may poll every 50 ms, GPU metrics every 200 ms, and Redfish telemetry every second, yet all remain phase-aligned through the shared epoch.

To maintain temporal consistency, the timing engine relies on the `clock` package, which defines both the aligned ticker and a monotonic timeline abstraction. The aligned ticker computes the initial delay to the next multiple of the polling period and then emits ticks at strictly periodic intervals. Each emitted epoch is converted into Tycho's internal time representation using the `Mono` clock, which maps wall-clock time to discrete quantum indices. The quantum defines the global temporal resolution (default: 1 ms) and guarantees strictly non-decreasing tick values, even under concurrency or system jitter.

The engine imposes minimal constraints on collector behavior: callbacks are expected to perform non-blocking work, typically pushing samples into the respective ring buffer, and to return immediately. This ensures low scheduling jitter and prevents slow collectors from influencing others. Lifecycle control is context-driven: when the execution context is cancelled, all ticker goroutines stop gracefully, and the manager waits for their completion before shutdown.

1.20.3 Synchronization and Collector Integration

All collectors in Tycho are synchronized through a shared temporal reference established at engine startup. The `Manager` captures a single epoch and provides it to every aligned ticker, ensuring that all collectors operate on the same epoch even if their polling intervals differ by several orders of magnitude. As a result, each collector's tick sequence can be expressed as a deterministic multiple of the global epoch, allowing later correlation between independently sampled metrics without interpolation artefacts.

Collectors register themselves before the timing engine is started. Each registration includes the collector's name, polling period, enable flag, and a `collect()` callback that executes whenever the corresponding ticker emits a tick. This callback receives both the current execution context and the aligned epoch, which is immediately converted into Tycho's internal monotonic time representation via the `Mono.From()` function. The collector then packages its raw measurements into a typed sample and pushes it to its corresponding ring buffer.

Because all collectors share the same monotonic clock and quantization step, the resulting sample streams can be merged and compared without further time normalization. Fast sources, such as RAPL or eBPF, provide dense sequences of measurements at fine granularity, while slower sources such as Redfish or GPU telemetry produce sparser but phase-aligned data points. This synchronization model eliminates the implicit coupling between sources that existed in Kepler and replaces it with a deterministic, time-driven coordination layer suitable for high-frequency, heterogeneous metrics.

1.20.4 Lifecycle and Configuration

The timing engine is initialized during Tycho's startup phase, after the metric collectors and buffer managers have been constructed. Before activation, each collector registers its collection parameters with the `Manager`, including polling intervals, enable flags, and callback references. Once registration is complete, the engine locks its configuration and starts the aligned tickers. Further modifications are prevented to guarantee a stable scheduling environment during runtime.

At startup, all timing parameters are validated and normalized. Invalid or negative values are rejected or normalized to safe defaults, and the global quantum is verified to be strictly positive. Polling intervals and buffer windows are cross-checked to ensure consistency across collectors, and derived values such as buffer sizes are recomputed from the validated configuration. This guarantees deterministic timing behavior even under partial or malformed configuration files.

The configuration layer also provides flexible control over measurement cadence. Polling periods for individual collectors can be adjusted independently, allowing users to balance temporal precision against system overhead. The default parameters represent a high-frequency but safe baseline: 50 ms for RAPL, 50 ms for eBPF, 200 ms for GPU, and 1 s for Redfish telemetry. All tickers are aligned to the global epoch defined by the monotonic clock, ensuring that these differences in cadence do not lead to drift over time.

Engine termination is context-driven: cancellation of the parent context signals all tickers to stop, after which the manager waits for all goroutines to complete. This unified shutdown mechanism ensures a clean and deterministic teardown sequence without leaving residual workers or buffers in undefined states.

1.20.5 Discussion and Limitations

The timing engine establishes the foundation for Tycho's decoupled and fine-grained metric collection. By aligning all collectors to a shared epoch while allowing individual polling intervals, it eliminates the rigid synchronization that limited Kepler's temporal accuracy. This design provides a lightweight yet deterministic coordination layer, enabling heterogeneous telemetry sources to contribute time-consistent samples at their native cadence.

The engine's strengths lie in its simplicity and extensibility. Each collector operates independently, governed by its own aligned ticker, while context-driven lifecycle control ensures deterministic startup and shutdown. Because callbacks perform minimal, non-blocking work, jitter remains bounded even at high polling frequencies. This structure scales naturally with the number of collectors and provides a separation between timing logic, collection routines, and subsequent analysis stages.

Nevertheless, several practical limitations remain. The current implementation assumes a stable system clock and does not compensate for jitter introduced by the Go runtime or external scheduling delays. Collectors are expected to execute quickly; long-running or blocking operations may distort effective sampling intervals. Moreover, the engine's alignment is restricted to a single node and does not extend to multi-host synchronization, which would require external clock coordination. At

very high sampling rates, the cumulative scheduling overhead may also become non-negligible on resource-constrained systems.

Despite these constraints, the timing engine represents a decisive architectural improvement over Kepler's fixed-interval model. It provides the temporal backbone for Tycho's data collection pipeline and enables accurate, high-resolution correlation across diverse telemetry sources. The following section, § 1.21, describes how these samples are buffered and retained for subsequent analysis, completing the temporal layer that underpins Tycho's measurement and attribution framework.

1.21 Ring Buffer Implementation

1.21.1 Overview

Tycho employs a per-metric ring buffer to store recent collection ticks produced by the individual collectors. Each collector owns a dedicated buffer that maintains a fixed number of entries, replacing the oldest values once full. This approach provides predictable memory usage and allows fast, allocation-free access to recent measurement histories. All ticks are stored in chronological order and include a monotonic epoch, ensuring consistent temporal alignment with the timing engine. The buffers are primarily used as transient storage for downstream analysis, enabling energy and utilization data to be correlated across metrics without incurring synchronization overhead.

1.21.2 Data Model and Sample Types

Each ring buffer is strongly typed and holds a single metric-specific tick structure. These tick types encapsulate all data collected during one polling interval and embed the `SampleMeta` structure, which records Tycho's monotonic epoch. Depending on the metric, a tick may contain simple scalar values (e.g., total node power) or collections of per-entity deltas (e.g., per-process counters, per-GPU readings, or per-domain energy data). For example, a `RaplTick` stores per-socket energy deltas across all domains, while a `BpfTick` aggregates process-level counters and hardware event deltas observed during that tick. This typed approach simplifies access and ensures that all metric data (regardless of complexity) can be correlated on a uniform temporal axis defined by the timing engine.

1.21.3 Dynamic Sizing and Spare Capacity

The capacity of each ring buffer is determined dynamically at startup from the configured buffer window and the polling interval of the corresponding collector. This calculation is performed by the `SizeForWindow()` function, which estimates the number of ticks required to represent the desired time window and adds a small margin of spare capacity to tolerate irregular sampling or short bursts of delayed polls. As a result, each buffer maintains a stable temporal horizon while avoiding premature overwrites during transient load variations. If configuration changes occur, buffers can be resized at runtime, preserving the most recent entries to ensure data continuity across reinitializations.

1.21.4 Thread Safety and Integration

Each ring buffer can be wrapped in a synchronized variant to ensure safe concurrent access between collectors and analysis routines. The synchronized type, `Sync[T]`, extends the basic ring with a read-write mutex, allowing simultaneous readers while protecting write operations during tick insertion. In practice, collectors append new ticks concurrently to their respective synchronized buffers, while downstream components such as the analysis engine or exporters read snapshots asynchronously. A central `Manager` maintains references to all buffers, handling creation, resizing, and typed access. This design provides deterministic retention and thread safety without introducing locking overhead into the collectors themselves, keeping the critical path lightweight and predictable.

Appendix A

Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches

Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

SPECIALIZATION PROJECT 2

Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
JULY 31, 2025

Re-edited on
January 31, 2026

Study program:
Computer Science, M.Sc.

Imprint

Project: Specialization Project 2
Title: Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches
Author: Caspar Wackerle
Date: July 31, 2025
Keywords: process-level energy consumption, 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

The growing energy demands of data centers have positioned energy efficiency as a critical concern in modern cloud computing. As containerization becomes the dominant approach for deploying scalable workloads, understanding the energy consumption of individual containers gains strategic relevance. However, accurately attributing energy usage to containerized workloads remains a complex and largely unsolved challenge due to hardware abstraction, shared resource utilization, and limited telemetry visibility.

This thesis investigates the theoretical foundations, methodological challenges, and existing approaches to container-level energy consumption measurement. Emphasizing bare-metal Kubernetes environments, the study systematically explores system-level energy measurement techniques, the complexities of attributing node-level energy to individual containers, and the limitations of current measurement tools. Rather than developing a new estimation tool, this work provides a structured analysis of existing solutions, highlighting methodological gaps, validation challenges, and critical design considerations for future research and tool development.

The findings offer a consolidated understanding of the technical factors influencing container energy attribution and outline practical recommendations for advancing energy transparency in containerized cloud infrastructures.

Chapter 1

Introduction

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. Data centers accounted for around 1.5% (approximately 415 TWh) of the world's electricity consumption in 2024 and are set to more than double by 2030[2]. This figure slightly exceeds Japan's current electricity consumption.

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, as efficiency per unit of compute continues to improve more slowly[6]. As containerized workloads form a significant and growing fraction of data center operations, understanding their energy impact is of increasing relevance.

1.1.1 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 introduces a critical complication: accurately understanding and attributing their

energy consumption. Despite their operational advantages, containers obscure energy usage due to their shared-resource architecture, making transparent monitoring and assessment of their true energy efficiency significantly more difficult.

1.1.2 Thesis Context and Motivation

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 intended to explore the theoretical and methodological aspects of container energy consumption measurements in detail.

Furthermore, 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 energy measurement in computer systems at the hardware, firmware, and software levels. This thesis builds upon these foundations, focusing specifically on the problem of container-level energy consumption measurement.

1.1.3 Use of AI Tools

During the writing of this thesis, *ChatGPT*[14] (Version 4o, OpenAI, 2025) was used as an auxiliary tool to enhance efficiency in documentation and technical writing. Specifically, it assisted in:

- Assisting in LaTeX syntax corrections and document formatting.
- Improving clarity and structure in selected technical explanations.
- Supporting minor code analysis and debugging tasks.

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

1.1.4 Container Energy Consumption Measurement Challenges

Containerized environments introduce fundamental challenges to energy consumption measurement. Unlike virtual machines, containers share the host operating system kernel and underlying hardware resources. This shared architecture obscures the direct relationship between a specific container and the physical energy consumed, making isolated measurement infeasible.

While modern processors expose hardware-level telemetry via interfaces such as Intel's Running Average Power Limit (RAPL), these provide only node-level or component-level insights. In addition, such telemetry is typically inaccessible from within containers, particularly in multi-tenant or cloud-hosted environments. Public cloud providers often aggregate or abstract energy-related data, further limiting observability.

Attribution of energy consumption to containers is complicated by concurrent resource sharing. CPU cores, memory, network interfaces, and storage systems serve multiple containers simultaneously. Available resource utilization metrics, such as CPU time, memory usage, or performance counters, provide indirect signals that could be used for energy attribution, but don't directly attribute it.

Various tools attempt to model container-level power consumption by correlating resource utilization with node-level energy telemetry. However, these models are often simplistic, opaque, or specific to particular hardware setups, and lack systematic validation.

Collectively, these factors result in a complex, multi-layered measurement problem: translating node-level energy consumption into accurate, container-level usage estimates within modern cloud infrastructures.

1.1.5 Scope and Research Questions

This thesis investigates the theoretical and practical landscape of container energy consumption measurement, focusing on the attribution of energy usage within bare-metal Kubernetes environments. Instead of proposing a novel measurement tool or developing a new energy estimation model, the thesis aims to systematically analyze the problem space and existing solutions.

The objective is to identify the methodological, technical, and practical factors that influence accurate container-level energy measurement. The study evaluates how existing tools address these challenges and highlights unresolved issues that hinder reliable and standardized energy attribution in containerized systems.

To guide this exploration, the following research questions are posed:

- **RQ1:** What are the fundamental challenges that prevent accurate measurement of container-level energy consumption?
- **RQ2:** Which methods, metrics, and models currently support container energy consumption estimation, and how do they address the attribution problem?
- **RQ3:** How do existing tools implement container-level energy consumption estimation, and what are the limitations of their approaches?

Rather than explicitly answering these questions in isolation, the thesis addresses them throughout its analysis: These questions structure the thesis' analytical approach. A detailed overview of the thesis structure is provided in the following section.

1.1.6 Terminology: Power and Energy

In this thesis, the physical units power (measured in watts) and energy (measured in joules) are used interchangeably where appropriate. While these units describe distinct physical quantities (energy representing the total amount of work performed, and power representing the rate of energy usage over time), the time interval in question is generally known or defined in all relevant contexts, rendering conversion between them trivial.

As a result, discussions of container-level energy consumption, power usage, and energy attribution may reference either energy or power depending on context, without introducing ambiguity. Where necessary, the specific unit used is stated explicitly.

1.1.7 Contribution and Structure of the Thesis

This thesis contributes a structured, theory-focused exploration of container-level energy consumption measurement. Rather than presenting a novel tool or proposing a specific implementation concept, it synthesizes existing methods, models, and challenges relevant to this field. Its primary contribution lies in analyzing how energy consumption can be measured and attributed to individual containers in bare-metal Kubernetes environments, highlighting limitations, and identifying open challenges for future work.

The thesis is structured as follows:

- **Chapter 2** introduces the fundamentals of server energy consumption, including hardware-level telemetry, component-level energy behaviors, and system power management mechanisms.
- **Chapter 3** analyzes the attribution problem of mapping node-level energy consumption to individual container workloads, identifying core challenges and influencing factors.
- **Chapter 4** surveys existing tools and approaches that attempt to estimate container-level energy consumption, assessing their methodologies and limitations.
- **Chapter 5** synthesizes the findings, identifies open questions, and outlines recommendations for future research and tool development.

Both this thesis and the preceding implementation-focused project are publicly available in the PowerStack[1] repository on GitHub. While this thesis does not come with additional code, the repository contains Ansible playbooks for automated deployment, Kubernetes configurations, monitoring stack setups, and benchmarking scripts from the preceding thesis.

Chapter 2

State of the Art and Related Research

2.1 Energy Consumption Measurement and Efficiency on Data Center Level

Energy consumption and efficiency at the data center level have been well-studied to the point where various literature reviews have been published[15, 16]. Much of this research focuses on data center infrastructure (cooling and power), and with good reason, as infrastructure is responsible for a large share of total energy consumption. While a wide range of coarse-, medium-, and fine-grained metrics for data center energy consumption exist, most operators have concentrated on improving coarse-grained metrics, especially *Power Utilization Effectiveness (PUE)*, through infrastructure improvements. This has resulted in a PUE of 1.1 or lower in some cases[5]. Meanwhile, server energy efficiency has improved substantially, particularly for partial load and idle power[17]. This has allowed data center operators to improve energy efficiency simply by installing more efficient cooling and power systems and servers. Fine-grained metrics, such as server component utilization rates or speeds, 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 Server Level

As a result of the energy efficiency improvements of both data center infrastructure and server hardware mentioned in the previous section, attention has shifted towards evaluating actual server load energy efficiency. Efficiency gains at this level compound into further improvements at the data center level. The resource-sharing methods of modern cloud computing(and especially the use of containers)create significant opportunities for server workload optimization for energy efficiency, which in turn requires power consumption measurements for evaluation. In the context of containers on multi-core processors, measuring the energy consumption of the entire server is insufficient, as it does not allow attribution of consumed energy to specific containers or processes. While component-level power measurements provide finer granularity and could theoretically be modeled to reflect container energy consumption, they drastically raise complexity for several reasons:

- Component-level energy consumption measurement without external tools is

far from straightforward. While some components provide estimation models (e.g. Intel RAPL or *Nvidia Management Library* (NVML)), others can only be estimated using performance metrics. This invariably leads to large measurement uncertainties, especially due to hardware differences between generations and manufacturers.

- Attributing measured or estimated energy consumption to individual containers is itself a non-trivial problem: it requires fine-grained time synchronization between energy consumption and container resource usage due to the fast-switching nature of most server components during multitasking.

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

- Any server-level power models used to estimate the relationship between individual component energy consumption suffer from the variety of server configurations, due to specialization such as storage-, GPU-, or memory-optimized servers.

In a systematic review of cloud server power models, Lin et al.[18] categorize power collection methods into four 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 acquisition system	Specialized systems	Specified models of machines	Difficult	Machine or component-level	High
Based on software monitoring	Built-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 related to the problem domains listed above, focusing on different measurement approaches: direct hardware measurements, in-band measurement techniques, and model-based estimation. The sections are organized by measurement approach, foregoing organization by server component. For this reason, § 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 Acquisition

Instrument-based data acquisition produces the highest data credibility at 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 of simply measuring electric power at convenient hardware locations using dedicated equipment can, of course, be extended to provide additional granularity. For example, Desrocher et al.[19] custom-created a DIMM extender fitted with Hall-sensor resistors and a Linux measurement utility to measure the power consumed by a DIMM memory module at a 1kHz sampling rate, using a *WattsAppPro?* power meter and a *Measurement Computing USB.1208FS-Plus* data acquisition board.

This highlights a fundamental truth of instrument-based data collection: while it is possible to implement a measuring solution that provides high-granularity and high-sampling rate power data, this requires immense effort, as such solutions are not available off-the-shelf. Unsurprisingly, 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 in data center servers due to its poor scalability and prohibitive costs. Hence, the primary role of instrument-based power data acquisition is as a benchmarking and validation tool for research and development.

2.3.2 Dedicated Acquisition Systems

2.3.2.1 BMC Devices, IPMI and Redfish

Some manufacturers have developed specialized power data acquisition systems for their own server products. The baseboard management controller (BMC) is a typical dedicated acquisition system usually integrated with the motherboard, typically as part of the Intelligent Platform Management Interface (IPMI)[18]. It can be connected to the system bus, sensors, and a number of components to provide power and temperature 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 integration between system software and underlying hardware. BMC devices on modern servers often offer IPMI or Redfish interfaces. While these interfaces use the same physical sensors, their implementations differ significantly, with Redfish generally offering 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.[20] found the accuracy of IPMI power data to be relatively low 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 idle power. They concluded that IPMI power data was still useful when a longer averaging window was used and the initial data points were discounted. In a later study, they suggested combining the measurements of IPMI

and Intel RAPL (which they found to underestimate power consumption) for a reasonable approximation of true measurements[21]. Kavanagh's findings have been cited in various studies, often to argue against using IPMI for power measurement. When used, it is sometimes chosen simply because it was the "simplest power metric to read"[22] in the context of entire data centers.

Redfish is a modern out-of-band management system, first released in 2015 explicitly to replace IPMI[23]. It uses a RESTful API and JSON data format, making queries easier to perform via code. In 2019, Wang et al.[24] directly compared IPMI and Redfish power data to readings from 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 attributed to the improved latency.

In conclusion, BMC power data acquired over Redfish provides a simple and comparatively easy way to measure system power based on various physical system sensors. Its greatest strength 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 highly dynamic 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[20].

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 [25]. 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 has been 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 that provides a way to monitor and control the energy consumption of various components within their processor package[26]. An adaptation of RAPL for AMD processors uses largely the same mechanisms and the same interface[27], although it provides less information than Intel's RAPL, offering no DRAM energy consumption[28]. Unfortunately, RAPL lacks detailed low-level implementation documentation, and the exact methodology of the RAPL calculations remains unknown[29].

Intel RAPL has been used extensively in research to measure energy consumption[30] despite some objections regarding its accuracy, which will be discussed in § 2.4.2.2 and § 2.4.2.3. The general consensus is that RAPL is *good enough* for most scientific work in the field of server energy consumption and efficiency. As Raffin et al.[31] point out, it is mostly used *like a black box without deep knowledge of its behavior*, resulting in implementation mistakes. For this reason, § 2.4.2.1 presents an overview of the RAPL fundamentals.

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 Architectures Software Developer's Manual[32, Section 16.10] and the works of Raffin et al. [31] (2024) and Schöne et al. [33] (2024).

Running Average Power Limit (RAPL) is a power management interface in Intel CPUs. Apart from power limiting and thermal management, it also allows measuring the energy consumed by various components (or *domains*). These domains include individual CPU cores, integrated graphics (in non-server CPUs), and DRAM, as well as the *package*, referring to the whole CPU die. While it initially used models to estimate energy use[34], it now uses physical measurements. The processor is divided into different power domains or "planes", representing specific components,

as 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 do not show the *PP1 uncore* domain (typically used for integrated GPUs), and client-grade processors do not 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 focuses on the RAPL power domains available to server-grade processors.

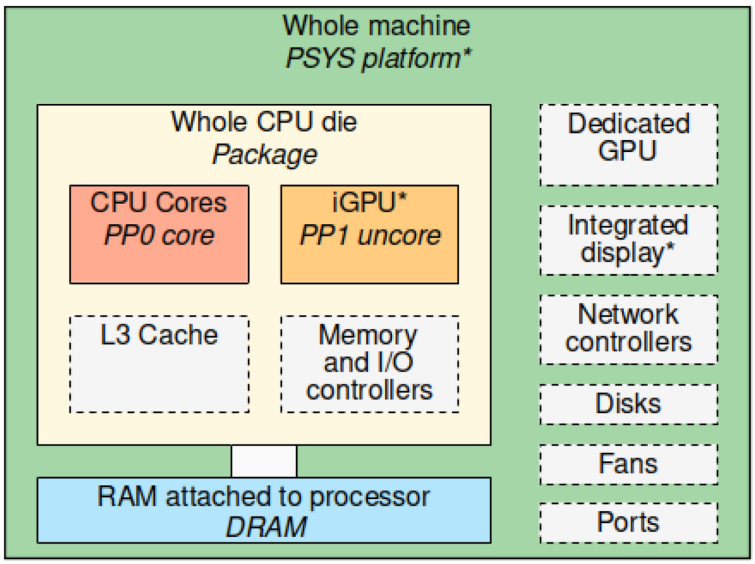


FIGURE 2.1: Hierarchy of possible RAPL domains and their corresponding hardware components. Domain names are in italic, and grayed items do not form a domain on their own. Items with an asterisk are not present on servers[31].

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 power meters are required, nor 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.[31] outline their individual features and tradeoffs, summarized 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.

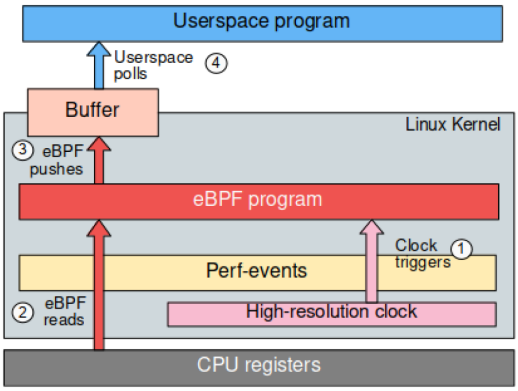
- The **Model-Specific Register (MSR)** interface provides low-level access to RAPL energy counters but is complex and hardware-dependent. Developers must manually determine register offsets and unit conversions based on processor model and vendor documentation. This method lacks safeguards, requires

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 (as 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[31]

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 to prevent high measurement overhead caused by reduced time spent in low-power states. Under heavy load, a high frequency can be used to capture more information.
- `perf-events` is the overall recommended measurement method, providing good efficiency, latency, and overflow protection. Powercap is less efficient but provides a simpler sysfs API.
- Even though `perf-events` and the eBPF measurement method appear to be the most energy-efficient, they are not recommended due to their complexity. For the same reason, the MSR method is not recommended, 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% (compared to <1% on standalone systems)[29].

2.4.2.2 RAPL Validation

Since its inception, RAPL has been the subject of various validation studies, with the general consensus that its accuracy could be considered "good enough"[31]. Notable works include Hackenberg et al., who in 2013 found RAPL accurate but missing timestamps[35], and in 2015 noted a major improvement to RAPL accuracy after Intel switched from a modeling approach to actual measurements for their Haswell architecture[34]. Desrochers et al. concluded in a 2016 RAPL DRAM validation study[19] that DRAM power measurement was reasonably accurate, especially on server-grade CPUs. They also found measurement quality to drop when measuring an idling system. Later, Alt et al.[36] 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 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 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 heterogeneity can improve performance and energy efficiency, it also increases the complexity of scheduling decisions and power-saving mechanisms, adding to the already complex architecture, which features per-core Dynamic Voltage and Frequency Scaling (DVFS), idle states, and power limiting/thermal protection.

Schöne et al.[33] 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 consumer-grade Intel Core i9-12900K processor measured at the base frequency of 0.8GHz.

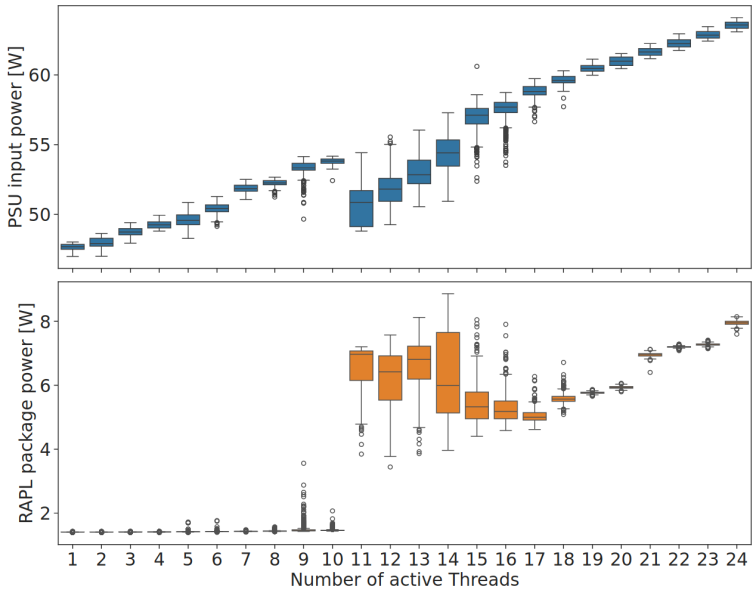


FIGURE 2.3: RAPL and reference power consumption sampled at 100 ms / 50 ms intervals, respectively. Double precision matrix multiplication kernel at 0.8GHz running for 60s each at increasing number of active threads[33].

2.4.2.3 RAPL Limitations and Issues

Several limitations of RAPL have been identified 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[31, 37]. 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 (61μJ 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*[32]. This is solvable with a simple correction, provided that the measurement intervals 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:

- **DRAM accuracy:** DRAM accuracy can only reliably be used for the Haswell architecture[19, 36, 37] and may still exhibit a constant power offset (attributed to the voltage regulator power loss of the memory system).

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

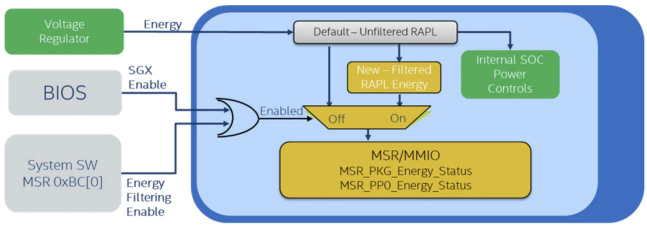
- **Unpredictable timings:** While Intel documentation states that the RAPL time unit is 0.976ms, the actual intervals may vary. This is an issue since the measurements do not include timestamps, making precise measurements difficult[37]. Several coping mechanisms have been used to mitigate this, notably *busypolling* (polling the counter for updates, significantly compromising overhead in terms of time and energy[38]), *supersampling* (lowering the sampling interval, increasing overhead and occasionally creating duplicates that need to be filtered[37]), or *high-frequency sampling* (lowering the sampling rate when the resulting data is still sufficient[39]). Another solution is to use a *low sampling frequency* to smooth out the relative error due to spikes, with the only drawback of a loss of temporal precision. At sampling rates slower than 50Hz, the relative error is less than 0.5%[29].

- **Non-atomic register updates:** RAPL register updates are non-atomic[37], 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.

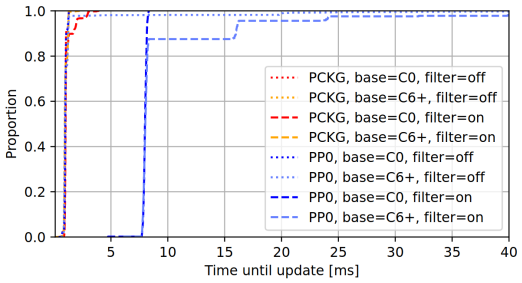
- **Lower idle power accuracy:** When measuring an idling server, RAPL tends to be less accurate[19, 33].

- **Side-channel attacks:** While the update rate of RAPL is usually 1ms, it can get as low as 50μs for the PP0 domain (processor cores) on desktop processors[33]. This can be exploited to retrieve processed data in a side-channel attack (coined "Platypus")[33, 40].

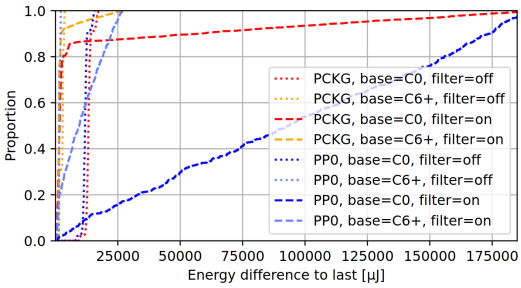
  To mitigate this issue while retaining RAPL functionality, Intel implements a filtering technique via the `ENERGY_FILTERING_ENABLE`[41, Table 2-2] entry or when *Software Guard Extension (SGX)* is activated in the BIOS. This filter adds random noise to the reported values (visualized in figure 2.4a). 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 of granularity resulting from its activation. In a 2022 article, Tamara[42] found a surprisingly 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[43]



(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 with 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 significantly influences measurements for the PP0 domain and measurably affects PCKG measurements.

FIGURE 2.4: Observable loss of granularity caused by the activation of `ENERGY_FILTERING_ENABLE`[33]

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 caused inaccuracies in developed measurement tools, but corrections for 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[44]. Kubernetes now includes mechanisms for GPU provisioning, enabling containerized workloads to leverage GPU acceleration[45].

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[46]. This increase is largely attributed to ML workloads and real-time processing tasks that benefit from the parallel processing capabilities of GPUs[47]. 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[48, 49], either through 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[45].

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[50]. 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[51]. 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[52]. 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[50].

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[53]. They concluded 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 offers 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 hundred milliseconds on some devices. Additionally, a slower rise (with linear growth) was observed, taking over a second to reach 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 remained 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[53]. 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

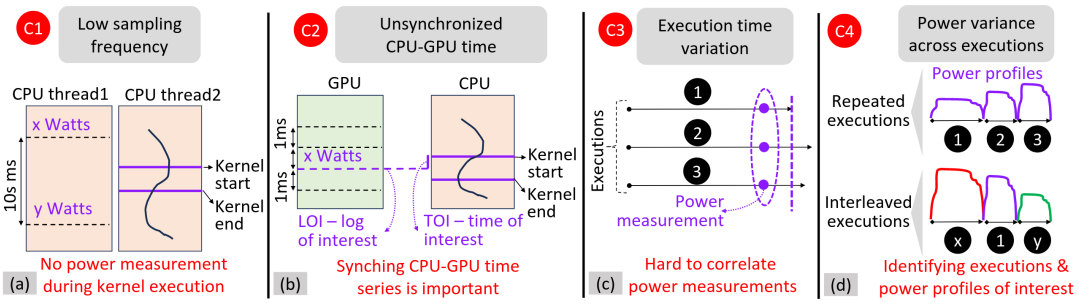
While most research has used NVIDIA-NVML to measure GPU power consumption, some research has focused on alternative measurement tools, usually to address issues similar to those stated by Yang et al. in the previous section. Specifically,

the following three tools were proposed to provide higher sampling rates to enable finer-grained power analysis.

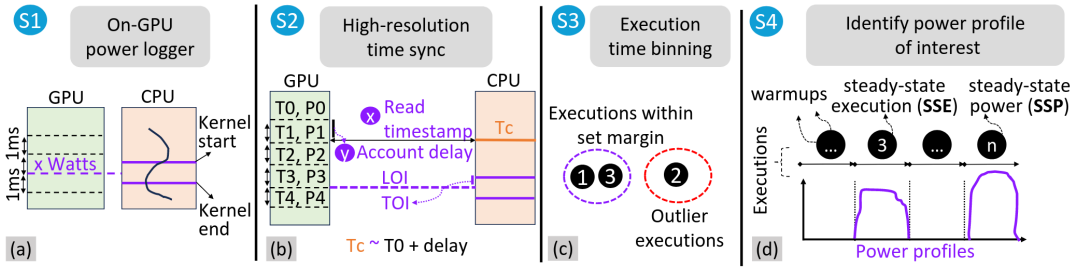
AccelWattch In 2021, Pan et al. proposed *AccelWattch*[54], a configurable GPU power model that provides both a higher accuracy cycle-level power model and a way to measure constant and static power, utilizing any pure-software performance model, NVIDIA-NVML, or a combination of the two. Notably, their 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. proposed *FinGraV*[55] (abbreviated from **Fin**e-**Gra**in **V**isibility), a fine-grained power measurement tool capable of sub-millisecond power profiling for GPU executions on an AMD MI300X GPU. They identified the following 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, result in fluctuating power consumption, challenging consistent profiling.



(A) Challenges in fine-grain GPU power analysis



(B) FinGraV strategies to address challenges

FIGURE 2.5: FinGraV GPU power measurement challenges and strategies[55]

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 benchmarks reveals several critical observations: Kernel executions differ significantly between initial runs and steady-state, with deviations of 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*[56] 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 studies have investigated the power consumption of storage devices. In 2008, Hylick et al.[57] investigated real-time HDD energy consumption and found significant differences in power consumption between standby, idle, and active power states. Cho et al.[58] proposed various energy estimation models for SSDs after measuring and comparing the energy consumption of different models. The most notable model-based energy consumption estimation mechanisms are presented in § 2.5.4.

In contrast to CPU or GPU components, storage devices (HDD, SSD, or NVMe drives) cannot make use of physical power sensors. While a BMC-measurement-based solution would technically be feasible, real-world implementation is impractical: while a BMC might be able to measure the power supply to a storage device, it typically is not exposed through IPMI or Redfish. Such measurements would further be complicated 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 § 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`[59] 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`[60] 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, throughput, and other indicators. Often, HDD speed is exposed. Notably, *SMART* metrics are typically more focused 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 Interface Card (NIC) are almost always connected via PCIe. As such, many cards support device power states[61] as specified by the PCIe specifications. Notably, not all NICs support all (or any) power states. These device states allow 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 is 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 be reliably estimated (especially beyond just NICs), device power states cannot be used to reliably estimate device power consumption. An attempt at the estimation of NIC power consumption is covered in § 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 utilization and server power, a great number of models use CPU metrics as the only indicator of server power. Fan et al.[62] proposed a linear interpolation between idle power and full power, which

they further refined 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 introducing more complex non-linear models, such as Hsu and Poole[63], 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 throughout related research) has historically been a popular suggestion[64]. 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[65].

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 as an assembly of its components, as Song et al.[66] propose:

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

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 devices.

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[67]. 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[18], 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.[62] propose a linear interpolation between idle power and full power based on CPU utilization. Basmadjian et al.[68] 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.[69] proposing a polynomial model as a univariate function of CPU utilization. Other models include individual CPU components[70, 71] 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 existence of a model capable of accurately estimating 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 have been proposed in the 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[72] or memory accesses[73], the number of cache misses[74], memory state[68], 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 focusing 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-dependent power estimation[75]. 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[76]

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[76]

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[77]

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[77]

TABLE 2.3: Power consumption for various storage device types.

Apart from simple estimations like those shown in tables 2.3a to 2.3d, a few works have focused on the energy consumption of individual storage devices. In 2015, Cho et al. developed Energysim[58], an SSD energy modeling framework advancing 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 using 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 be generalized to other storage devices where these are unknown.

In 2014, Li and Long[78] presented 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 developed 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 device's idle, standby, and busy power consumption. In the context of this 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.[79] proposed a number of models based on generalized stochastic Petri nets (GSPN) for performance and energy consumption evaluation of individual and hybrid (HDD + SSD) storage systems. GSPN is a suitable formalism for storage system design as, unlike queueing network models, synchronization, resource sharing, and conflicts are naturally represented. Also, the 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 explicitly incorporates power consumption per operation (e.g. random-read-4KB on SSD).

The following notation is adopted:

- $E\{\#p\}$ 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 probability 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 at a time is carried out. Concerning SSDs-NVMe, *pResource* assumes the number of threads 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 fell 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 initial 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 significant 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 extensive research has analyzed the power consumption of network equipment like switches, routers, or gateways, NICs (especially non-wireless NICs) have not received as much attention. While several methods exist that modern NICs use to save power (e.g. PCIe Link power states and D-states, *Active State Power Management (ASPM)*, or *Energy Efficient Ethernet (EEE)*), there are no 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.[80] measured and compared the power consumption of six 10 Gbps and four multiport 1 Gbps NICs at a fine-grained level. While they do not provide a method to estimate NIC energy consumption, they noted significant variation in power consumption between different NICs. Unfortunately, it cannot be ruled out that some results are cherry-picked: Solarflare NICs tend to dominate the introduced metrics, and a communications spokesperson is prominently credited with contact information. Regardless, some findings are consistent across manufacturers and align with other literature sources[81]. While these findings cannot directly contribute to a potential NIC power consumption estimation approach, they are relevant to understanding underlying mechanisms and assessing the relative importance of the NIC compared to other server components.

- **Idle Power**
  - The measured NICs showed power consumption between 5–20W.
  - Link connection status had little effect on idle energy consumption.
  - Physical media influenced power consumption: CX4 models had the lowest power consumption due to the simple design of the CX4 interconnect.

This was followed by fiber models. Finally, Base-T models consumed significantly more power due to the signal processing components in the card.

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

In 2012, Basmadjian et al.[82] modeled a NIC by separating NIC power consumption into idle mode and dynamic mode (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 is 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 is 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, it shows that NIC power consumption is assumed to rise linearly with channel utilization.

Arjona Aroca et al.[83] modeled NIC efficiency based on their previous measurements. They found that NIC efficiencies for both sending and receiving are almost linear with the transfer rate and deduced a linear dependency on the network throughput.

In 2016, De Maio et al.[84] proposed a network energy consumption model for node-to-node transfers to estimate the total energy consumption required for virtual machine migration. Unfortunately, their model does not specifically handle NIC energy consumption, opting to model the entire node's energy consumption instead.

Another approach is presented by Dargie and Wen[85], who used stochastic modeling to examine the relationship between the utilization of a NIC and its power consumption, expressing these quantities as random variables or processes. They used curve fitting to determine the relationship between utilization and measured energy consumption of their specific NIC, after creating a dataset using a SPECpower benchmark. They assumed a uniformly distributed bandwidth utilization in the interval [0,125] MBps. Interestingly, their model showed 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.[86] in 2024, analyzing per-application energy consumption. The authors noted that NIC idle power consumption may contribute up to 90% of the total NIC energy consumption. They proposed 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 bandwidth of both the uplink and downlink of the NIC, and BW_{used} is the used bandwidth of links (uplinks, downlinks, or both). The authors combined 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 if idle and maximum NIC power consumption are known.

In contrast to the formulae presented by Basmadjian and Arjona Aroca, these 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 formula currently available. A generalizable formula for estimating overall NIC power consumption is unlikely to exist due to the vast variety of NICs and the significant 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 § 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.[82], 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 provides information at the server level, the power consumption of any of the n PSUs is given by the following formula, 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 delivered[82]:

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

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 to the square 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 telemetry 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.[18] 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, which is 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 the 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 granularity mismatch issues. Consequently, direct one-to-one mapping of energy consumption to workloads 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 4** will examine how selected tools implement these ideas in practice. While some implementation aspects will be referenced for illustration, this chapter focuses 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 § 3.2.2. § 3.2.3 examines and compares different utilization tracking mechanisms in Linux and Kubernetes. As 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 § 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 § 3.2.5. Finally, § 3.3 discusses the different philosophies of various attribution models to 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 [87–92]. 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, allowing 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 aforementioned methods. While user-friendly, these usually offer 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 approximately 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 no direct equivalent in **cgroups-v2**, support was added in `iptables` to allow BPF filters that hook on cgroup v2 pathnames to control 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. [93], 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[94], 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.

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.

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

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 metrics introduce several challenges that need to be addressed by a power attribution methodology. Some of these represent natural trade-offs that an architect needs to be aware of, while others pose issues that simply cannot be circumvented without major drawbacks 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[95].

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[94].

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[96]. 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 trade-offs 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 § 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 sub-second 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 or 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, its 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[97]. 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 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

Approaches and Tools for Container Energy Measurement

4.1 Introduction

Accurately measuring and attributing energy consumption in containerized environments has become a central challenge in sustainable cloud computing. As container orchestration platforms like Kubernetes grow in adoption, the need for energy observability at finer granularities (down to the container or even process level) becomes increasingly critical. This requirement stems from a range of applications, including cost optimization, carbon accounting, energy-aware scheduling, and performance tuning.

A number of tools and frameworks have emerged in recent years to address this problem. Some focus on the system or server level, exposing power metrics via standardized interfaces or external instrumentation. Others adopt telemetry-based estimation approaches that infer energy usage from resource utilization statistics. More recently, several tools have begun to target container-level energy attribution specifically, often by integrating with Kubernetes and leveraging technologies such as eBPF, RAPL, and cgroups.

This chapter surveys the landscape of existing tools, organized into three categories: system-level monitoring solutions, telemetry-based estimation frameworks, and container-focused energy attribution tools. Particular attention is given to tools that support Kubernetes environments, as these are directly relevant to the goals of this thesis. Each tool is analyzed based on its architecture, metric sources, modeling approach, and practical limitations. Later sections discuss the emerging tool KubeWatt and present a comparative synthesis of strengths and weaknesses across the reviewed solutions.

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 remains an important aspect. Scientific works and tools in this domain generally do not 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[21] 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[98] 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[99]. 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[100] 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[101] (PowerAPI, HWPC, SmartWatts)

PowerAPI[102] 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 § 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**[103]: 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*[104] library, enabling accurate runtime learning of workload-specific power signatures. SmartWatts is further detailed in § 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)[105] 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

While system-level monitoring and telemetry-based estimation provide valuable insights into overall server energy consumption, they fall short when it comes to attributing energy use to individual containers. In multi-tenant or microservice-based environments, such granularity is essential for accurate accountability, optimization, and scheduling decisions.

This section focuses on tools specifically designed to address this challenge by providing energy attribution at the level of containers or processes within a containerized environment. These tools typically integrate with container runtimes and Kubernetes, leveraging sources such as hardware counters, control groups, and performance monitoring frameworks to estimate or infer energy consumption.

The following subsections analyze three prominent tools (Kepler, Scaphandre, and SmartWatts), each with a distinct architectural approach. A fourth tool, KubeWatt, is discussed separately as a derivative implementation developed in response to identified limitations in Kepler.

4.3.1 Kepler

4.3.1.1 Overview and Goals

Kepler (*Kubernetes-based Efficient Power Level Exporter*)[106] 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[107].

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.

It should be noted that shortly before the completion of this thesis, version 0.10.0 of Kepler was released. This version constitutes a major architectural rewrite of the project, intended to address structural limitations of earlier versions. However, the analysis in this chapter focuses on Kepler versions 0.9.x and earlier, which remain the most widely deployed at the time of writing. § 4.3.1.8 briefly summarizes the key changes introduced in the new release.

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.

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

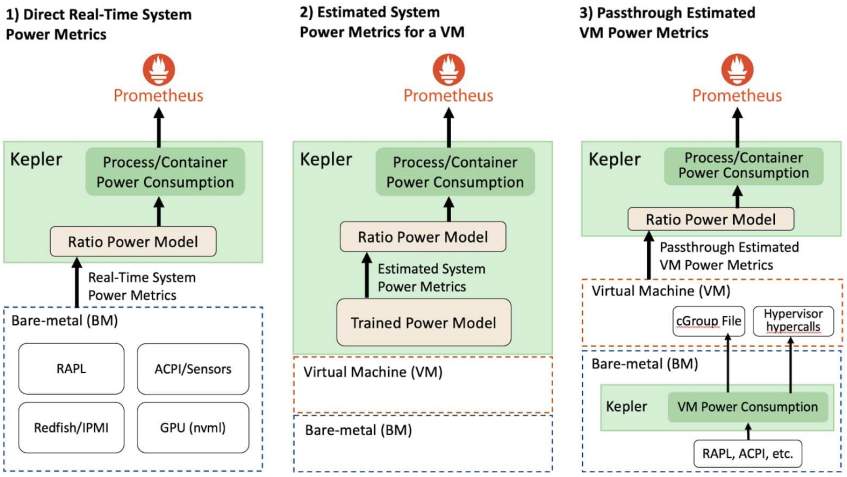


FIGURE 4.1: Kepler deployment models: direct power measurement on bare-metal, estimation on VMs, and the proposed passthrough model (currently not implemented)[108]

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 shown in figure 4.2.

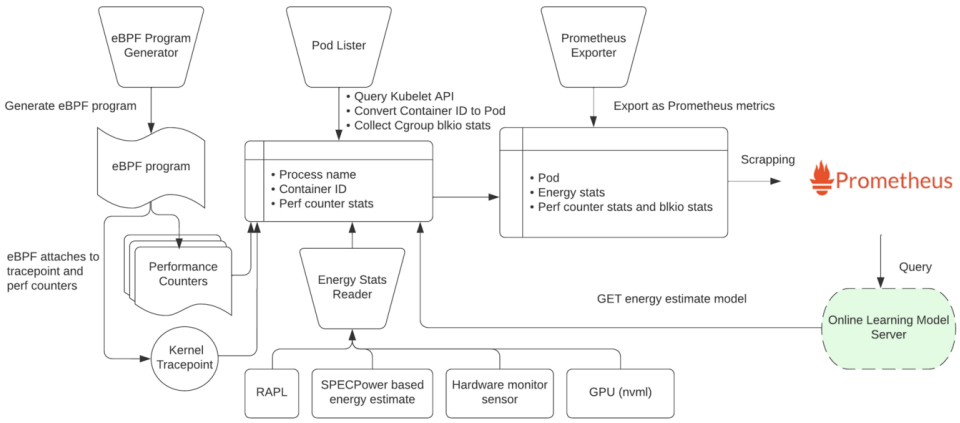


FIGURE 4.2: Simplified architecture of the Kepler monitoring agent and exporter components[108]

Resource 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. The 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 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 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` and `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, 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 like `nvmlDevice.GetProcessUtilization` to return process statistics, including Streaming Multiprocessor utilization (`SmUtil`), memory utilization (`MemUtil`), and optional encoding/decoding activity. Additionally, Kepler collects GPU energy consumption using NVML's `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`. 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 of 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 attribution. 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[109], 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 core functionality, this integration facilitates operational awareness and debugging.

4.3.1.3 Attribution Model and Output

Kepler Configurability and Measurement Interval Kepler updates its input metrics at a configurable interval (default: `SamplePeriodSec` = 3 seconds), with the exception of Redfish-based measurements, which have a separate, user-defined interval (default: 15 seconds). Although utilization and RAPL energy metrics theoretically support more fine-grained sampling, Kepler performs attribution solely based on aggregated values within each configured sampling window. While this reduces system overhead, it may miss short-lived energy fluctuations. Kepler's attribution framework is modular by design, allowing easy integration of new power sources through standardized interfaces. Depending on the availability of input metrics, Kepler uses either direct power readings or model-based estimation. In the absence of hardware telemetry (i.e., RAPL), Kepler can apply regression models trained on a reference system where accurate power readings are available. For this thesis, only scenarios where all relevant metrics are available (as discussed in § 4.3.1.2) are considered.

Division of Power into Idle and Dynamic Components Kepler conceptually divides total power consumption into *idle power* and *dynamic power*. Their sum constitutes the total (often named *absolute*) measured energy on both process and node levels. Dynamic power is correlated with resource utilization, while idle power represents the static baseline consumption that persists regardless of system activity.

This distinction is critical, as Kepler distributes these two components differently among processes [110]. Idle power attribution follows the Greenhouse Gas Protocol guidelines [111], which recommend splitting idle power across containers based on their size (i.e., resource requests relative to the total). Dynamic power, by contrast, is attributed proportionally to observed resource usage.

Node-Level Energy Attribution On the node level, Kepler directly uses RAPL energy readings from the `pkg`, `core`, `uncore`, and `dram` domains for all available sockets, reporting energy in millijoules. GPU energy is measured per device via NVML. Platform-level energy is retrieved via the configured source, typically Redfish. Kepler iterates through all exposed chassis members and internally converts Redfish power metrics (in watts) to milliwatts, which are then multiplied by `SamplePeriodSec` to produce energy values in millijoules. These can then be treated like standard metrics.
After collecting node-level energy metrics, Kepler updates the tracked *minimum idle power* if a new lower value is observed. The corresponding resource usage is also recorded. This tracking applies to all RAPL domains and GPU devices. Node-level dynamic energy is computed as the difference between total and idle energy. Additionally, Kepler calculates the residual category *Other* as platform power minus the sum of `pkg`, `dram`, and GPU power, separately for idle and dynamic metrics. This implicitly attributes all remaining energy to unspecified components (e.g. network interfaces, fans, or storage devices).

Process-Level Energy Attribution Process energy is attributed based on resource utilization metrics and node-level energy values. Using the most recent measurements of component, GPU (if enabled), and platform power, Kepler calculates each process's energy consumption. Idle energy is simply divided evenly among all processes. This is in contrast to Kepler's documentation, which states that idle energy should be divided by requested container size (in accordance with the GHG protocol [111]). Since container-level metadata is aggregated from per-process statistics, implementing size-based idle attribution at the process level is non-trivial. Nonetheless, this feature is advertised but not implemented, as noted by a `TODO` comment in the codebase.

Dynamic energy is attributed proportionally using the formula:

$$E_{process_{dyn}} = \left\lceil E_{component_{dyn}} \cdot \frac{U_{process}}{U_{node}} \right\rceil \tag{4.1}$$

where U_{process} is the process's resource usage, and U_{node} is the total component resource usage indicated by RAPL. If no usage is recorded, energy is divided evenly among processes. The specific usage metrics used in the default ratio model are:

- **CPU**: based on `CPUInstructions` (fallback: `CPUTime`)
- **DRAM**: based on `CacheMisses` (fallback: `CPUTime`)
- **GPU**: based on `GPUComputeUtilization` (via NVML)
- **Other** and **Platform**: intended to use a configurable default metric

However, the default usage metric for `Uncore` and `Platform` domains is not set in the configuration, defaulting to an empty string. As a result, these metrics are not attributed based on usage and fall back to equal distribution. It is unclear whether this is a conscious design decision or an oversight.

The following formula 4.2 summarizes the complete energy attribution model used by Kepler at the process level. It combines all relevant energy domains: dynamic and idle energy across measured components (e.g. package, core, DRAM, uncore, GPU), as well as residual platform energy not accounted for by specific components. Each part of the equation reflects a distinct step in the attribution logic previously discussed, merging them into a single expression for total per-process energy consumption.

$$
\begin{aligned}
E_{\text{process}} = & \sum_{\text{comp} \in \{\text{core,dram,uncore,gpu}\}} \left(\left\lceil E^{\text{dyn}}_{\text{comp}} \cdot \frac{U_{\text{process, comp}}}{U_{\text{node, comp}}} \right\rceil + \left\lceil \frac{E^{\text{idle}}_{\text{comp}}}{N_{\text{processes}}} \right\rceil \right) \\
& + \left\lceil \frac{E^{\text{dyn}}_{\text{platform}} - \left(E^{\text{dyn}}_{\text{pkg}} + E^{\text{dyn}}_{\text{dram}} \right)}{N_{\text{processes}}} \right\rceil + \left\lceil \frac{E^{\text{idle}}_{\text{platform}} - \left(E^{\text{idle}}_{\text{pkg}} + E^{\text{idle}}_{\text{dram}} \right)}{N_{\text{processes}}} \right\rceil
\end{aligned} \tag{4.2}
$$

4.3.1.4 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.

This timing issue primarily affects metrics that are reported as cumulative deltas (e.g. energy in millijoules). For Redfish-based metrics, which are provided in watts and converted internally by Kepler into energy values by multiplying with the update interval, synchronization is less problematic. The conversion to energy is performed over a well-defined window, which inherently matches the Prometheus export interval, thus avoiding the same class of timing issues.

4.3.1.5 Validation and Research Context

Kepler has been adopted in various academic and industrial settings as a tool for estimating energy consumption at the container and pod level. In most cases, it is treated as a black-box exporter, with little investigation into the reliability or internal consistency of its reported metrics. While its integration with Kubernetes and Prometheus makes it convenient to use, its internal attribution mechanisms and modeling assumptions are rarely scrutinized in published work. Andringa[112] investigates Kepler, but lacks the necessary depth.

A notable exception is the study by Pijnacker et al. [113, 114], which constitutes the first dedicated empirical evaluation of Kepler's behavior and attribution accuracy.

Their work critically examines whether Kepler's per-container energy metrics correspond to expected utilization and measured system-level power under controlled conditions.

To this end, the authors design a testbed using a Dell PowerEdge R640 with dual Intel Xeon CPUs, operating a single-node Kubernetes cluster. Redfish/iDRAC power readings, validated using a wall power plug, are used as a high-confidence ground truth reference. Container CPU metrics and metadata are collected using Prometheus and cAdvisor. Kepler is deployed in version 0.7.2, configured to use RAPL for component-level power metrics and Redfish for platform-level readings.

The validation experiment employs a repeated CPU stress workload (`stress-ng -cpu 32`), combined with a set of idle containers left in the Completed state to test the attribution of idle power. Additional tests involve dynamic workload changes, including the deletion of idle pods during active load, to analyze Kepler's behavior during transitions in cluster state. These experiments reveal attribution inconsistencies across both idle and dynamic workloads.

Importantly, the authors isolate the source of Kepler's inaccuracy as its attribution logic, rather than its raw power estimation. While node-level energy estimates can match ground-truth readings closely, the logic used to assign portions of that energy to specific containers often fails to reflect actual utilization patterns. Inconsistent handling of kernel and system processes, as well as temporary spikes in attribution caused by metric timing mismatches, point to deeper architectural shortcomings. These and other limitations are summarized in the next subsection.

4.3.1.6 Limitations and Open Issues

The results of the evaluation by Pijnacker et al. reveal several systemic issues in Kepler's container-level energy attribution. In parallel, this thesis identifies additional design and implementation problems through source code analysis and direct experimentation.

Validation results from Pijnacker et al. [114]:

- **Incorrect idle power attribution:** Kepler assigns idle power equally to all containers, including pods in the Completed state. This leads to energy being attributed to containers that are no longer active, skewing total container-level metrics.
- **Latency mismatch artifacts:** A mismatch between fast-updating utilization metrics (e.g. CPU usage) and slower power metrics (e.g. Redfish at 60s intervals) causes transient attribution spikes. This effect is most pronounced during workload transitions.
- **Inconsistent attribution to system processes:** When idle containers are removed or when attribution is unclear, Kepler sometimes redirects energy consumption to generic "system processes", even when this does not match observed CPU activity. This fallback mechanism introduces further attribution noise.

- **Unstable behavior during dynamic cluster state changes:** In deletion experiments, the removal of idle pods triggered inconsistent redistribution of both idle and dynamic power, including unexplained reductions in power attributed to active workloads.

- **Observability gaps for Completed pods:** Once a pod transitions to the Completed state, it may no longer be visible to cAdvisor or other telemetry tools. This can lead to orphaned power assignments in Kepler, which continues to attribute energy to containers that are no longer observable.

Additional issues identified in this work While the attribution issues demonstrated by Pijnacker et al. are the most critical and empirically validated shortcomings of Kepler, a detailed inspection of the source code conducted as part of this thesis (see § 4.3.1.3) confirms these problems and reveals several additional implementation and documentation-related concerns. Although these issues are less severe in their impact, they further highlight the current limitations of Kepler as a mature and reliable observability tool.

- **Incomplete codebase:** Kepler's source code contains over 1000 unresolved `TODO` markers, some located in critical modules such as the ratio-based power model. These incomplete sections may affect reliability or create hard-to-debug behavior in real deployments.

- **Discrepancy between documentation and implementation:** While Kepler's documentation describes proportional idle power distribution based on container size or usage, the actual implementation uses equal distribution across all containers, regardless of their activity or footprint.

- **Outdated documentation and diagrams:** Both public-facing and internal documentation, including UML diagrams and developer notes, are out of date. This lack of alignment between the codebase and its documentation hinders reproducibility and extension.

- **Inadequate measurement intervals for heterogeneous workloads:** Although Kepler collects advanced eBPF-based metrics at high frequency, RAPL-based measurements are only sampled every three seconds. For workloads with heterogeneous container behavior or short-lived processes, this can drastically impact accuracy. Higher sampling intervals for RAPL-based data would allow more fine-grained attribution.

Despite these limitations, it is important to emphasize that **Kepler remains the most advanced and integrated open-source implementation for process-level energy monitoring in Kubernetes environments to date**. Its combination of kernel-level instrumentation, Prometheus integration, and extensible modeling capabilities makes it a uniquely valuable reference point and starting foundation for future research and tool development.

4.3.1.7 KubeWatt (Derived from Kepler)

As a direct response to the shortcomings identified in Kepler's attribution model, Pijnacker developed *KubeWatt*, a proof-of-concept alternative designed for improved container-level energy observability in Kubernetes. While Kepler's modular design

and eBPF-based telemetry collection provide flexibility, it suffers from attribution artifacts, particularly related to static power division, temporal misalignments, and system process handling. KubeWatt explicitly addresses these issues by adopting a simpler, more transparent approach centered around external power readings and CPU-based attribution.

Separation of Static and Dynamic Power A core improvement in KubeWatt is its strict separation between static and dynamic power. Unlike Kepler, which distributes idle power among all containers, including non-running ones, KubeWatt isolates static power entirely. Static power, including control plane activity and system overhead, is measured upfront using one of two initialization modes. It is then excluded from per-container attribution. Dynamic power is attributed solely to active containers based on their proportional CPU utilization. This separation eliminates attribution artifacts such as idle containers appearing to consume power.

Initialization Modes for Static Power Estimation KubeWatt introduces two initialization modes to measure or estimate static power: *base initialization* and *bootstrap initialization*. The base mode measures node power in an empty cluster to establish an accurate baseline, while the bootstrap mode estimates static power during active workloads by fitting a third-degree polynomial regression model to observed CPU and power values. The initialization result is stored and reused, based on the assumption that static power remains stable over time.

Simplified Attribution Model For container-level attribution, KubeWatt modifies the hierarchical power-mapping model originally proposed in [112]. It attributes dynamic node power to containers proportionally to their CPU usage, normalized over all active containers. (While the specific metric used for *CPU usage* is a central feature of Kepler, KubeWatt does address this specifically.) This avoids using the full node CPU metric, which includes kernel and system overhead already covered by static power. The attribution model is transparent, avoids container misclassification, and is robust against changes in container count.

Resilience Against Kepler's Pitfalls In contrast to Kepler, KubeWatt shows robustness under dynamic workload changes, such as container creation or deletion. It avoids misattributing power to terminated or inactive pods and mitigates power leakage into "system_processes" labels. Additionally, KubeWatt explicitly ignores control plane pods in the dynamic attribution, instead incorporating their idle load into the static baseline.

External Power Source Integration Unlike Kepler, which may rely on internal sensors or hybrid models, KubeWatt is built around a single, clearly defined power source: the Redfish API via iDRAC. Power data is abstracted via a modular interface, allowing extensibility to other external sources. This design choice avoids mixing sensor domains and contributes to more interpretable results.

Improved Evaluation Outcomes Empirical evaluation confirms that KubeWatt reduces RMSE in total node power estimation compared to Kepler, achieves accurate attribution even under stressor workloads, and avoids over-reporting container power. Notably, the estimator mode maintains high accuracy as long as container CPU utilization is stable and consistent with Prometheus scrape intervals.

Overall, KubeWatt represents a targeted refinement of Kepler's architectural and attribution approach, favoring interpretability and correctness over feature completeness. While limited in scope (e.g. no support for GPU or memory attribution), it successfully mitigates several of the critical limitations identified in Kepler.

4.3.1.8 Re-Release of Kepler Version 0.10.0

During the final stages of this work, Kepler version 0.10.0 was released as a significant architectural rewrite. While maintaining its fundamental role as a Prometheus exporter for container-level energy metrics, the new release introduces considerable changes to both its internal implementation and energy attribution methodology. This section provides an overview of the revised architecture and critically analyzes the implications of these changes in the context of container-level energy observability.

Implementation Changes Kepler v0.10.0 represents a ground-up redesign focusing on modularity, thread-safety, and accessibility. Mutex locks are used extensively throughout the codebase to ensure safe concurrent metric collection. Unlike previous versions, Kepler now requires only read-only access to `/proc` and `/sys` filesystems, eliminating the need for elevated privileges (`CAP_SYSADMIN`, `CAP_BPF`). This is marketed as a significant improvement in deployability and security.

Container detection still relies on traditional cgroup path analysis, with per-process cgroups parsed via regex matching to identify container runtimes and IDs. However, unlike prior versions, container-level resource metrics are no longer derived by aggregating per-process metrics. Instead, Kepler v0.10.0 treats containers as first-class workloads: CPU usage statistics for containers are read directly from cgroups, without requiring process-level aggregation.

Node-level CPU activity is determined via the Linux `/proc/stat` interface, using the formula:

$$\text{CPUActiveTime} = \text{TotalTime} - (\text{IdleTime} + \text{IOWaitTime}) \tag{4.3}$$

A node-level usage ratio is computed as:

$$\text{CPUUsageRatio} = \frac{\text{CPUActiveTime}}{\text{TotalTime}} \tag{4.4}$$

This represents a significantly simplified utilization model compared to prior versions, which relied on more granular hardware performance counters (e.g. instructions retired, cache misses). While the new architecture is once again designed in a modular way, this specific ratio model is not, perhaps suggesting that no user-selectable utilization metric is aimed for, as there was in the previous version.

At the power measurement layer, Kepler continues to rely on RAPL readings, now referred to as "energy zones". In addition to standard `package`, `core`, `uncore`, and `dram` zones, Kepler v0.10.0 dynamically detects the `psys` domain, expanding applicability to non-server-grade processors. Multi-socket systems are supported via automatic aggregation of identically named RAPL zones, with counter wraparounds

correctly handled through delta calculations based on maximum energy values.

Finally, pod lifecycle awareness has seen notable improvements. Kepler now tracks ephemeral and init containers explicitly, indexing container IDs via Kubernetes' native API server. This mitigates previous observability gaps for short-lived and Completed pods, which were previously problematic due to reliance on cAdvisor and cgroup detection alone.

Power Attribution Model The energy attribution model has undergone a substantial simplification. Kepler v0.10.0 attributes energy solely based on CPU time. The node's total RAPL-reported active energy is distributed proportionally to workload CPU activity:

- **Node-level attribution:** At each interval, total energy deltas are computed from RAPL readings. Active and idle energy components are calculated using the node's CPU usage ratio:

$$E_{\text{active}} = \Delta E \cdot \text{CPUUsageRatio} \tag{4.5}$$

$$E_{\text{idle}} = \Delta E - E_{\text{active}} \tag{4.6}$$

- **Process-, container-, and pod-level attribution:** Processes, containers, and pods are treated as independent workloads. Each receives a fraction of the node's active energy based on its CPU time share:

$$E_{\text{workload}} = E_{\text{node, active}} \cdot \frac{\text{CPUTimeDelta}_{\text{workload}}}{\text{CPUTimeDelta}_{\text{node}}} \tag{4.7}$$

It is important to emphasize that container metrics are no longer derived from per-process metrics. Instead, both processes and containers draw energy directly from node-level active energy based on their own independently tracked CPU usage statistics. This parallel accounting structure avoids process-level aggregation.

Critical Analysis While Kepler v0.10.0 improves accessibility, security, and operational simplicity, these enhancements come at the cost of reduced modeling fidelity. CPU time has replaced more nuanced metrics such as CPU instructions and cache misses for workload attribution. The new model assumes linear energy scaling with CPU activity, disregarding workload heterogeneity and hardware-level power behaviors such as frequency scaling or turbo boost. By not attributing idle power to workloads, Kepler focuses on actionable energy consumption linked to active resource use. While this limits full node-level energy accountability at workload granularity, it avoids inaccuracies introduced by arbitrary idle power distribution.

Additionally, Kepler's model assumes that CPU time directly correlates to power consumption, overlooking variations introduced by CPU frequency scaling, idle states, instruction-level differences, and workload-specific resource behaviors (e.g. memory-bound tasks). These factors can significantly distort power attribution, especially in heterogeneous or bursty workloads.

rom a research perspective, the simplification represents a methodological compromise: Kepler prioritizes operational deployability and stability over modeling granularity. While suitable for general monitoring, its current attribution model is less suited to detailed workload-specific energy analysis.

In the author's view, Kepler's redesign trades attribution accuracy for simplicity and usability. While this architectural shift benefits practical deployment, it diminishes the tool's value for research-focused energy observability. Finer-grained temporal attribution and more diverse utilization metrics would have been desirable developments.

4.3.2 Scaphandre

4.3.2.1 Overview and Goals

Scaphandre[10] 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 (and sockets) by matching directories with 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 [115].

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, which are discussed in § 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. 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()`:

$$E_{\text{proc}}(t) = \frac{E_{\text{RAPL}}(t) \cdot \text{CPU}_{\text{proc}}(t)}{\sum_i \text{CPU}_i(t)} \tag{4.8}$$

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 § 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[95], 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[95], 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[31] was unable to push Scaphandre's 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*[103] 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 Python. 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 powermeter (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 for static power isolation, event selection, cgroups, and event monitoring. This separation allows for heterogeneous cluster nodes.

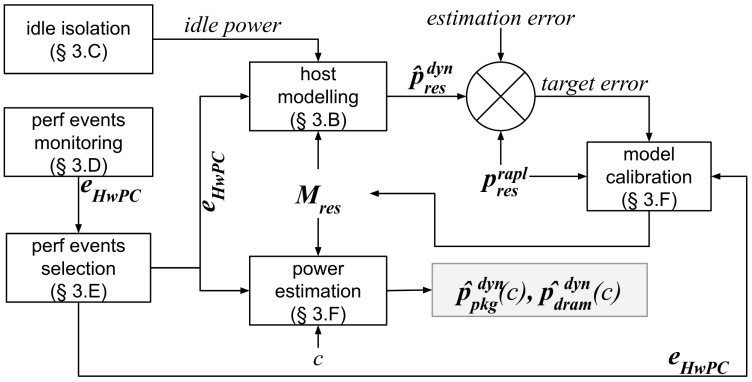


FIGURE 4.3: SmartWatts Architecture

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:

$$p_{res}^{rapl} = p_{res}^{static} + p_{res}^{dynamic} \tag{4.9}$$

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

$$p_{res}^{static} := median_{res} - 1.5 \cdot IRQ_{res} \tag{4.10}$$

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 HWPC:

$$\exists f \in F, \hat{p}_{res}^{dyn} = M_{res}^{f} \cdot E_{res}^{f} \tag{4.11}$$

where E_{res}^{f} denotes all *events*. The model M_{res}^{f} is built 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 constrained 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 applied to determine worthy candidates.

Container power consumption is estimated by applying the inferred power model M^f_{res} at the scale of the container's events $E^f_{res}(c)$, as seen in formula 4.12. In formula 4.13, the intercept i is distributed proportionally to the dynamic part of the consumption of c.

$$\exists f \in F, \forall c \in C, \hat{p}^{dyn}_{res}(c) = M^f_{res} \cdot E^f_{res}(c) \tag{4.12}$$

$$\forall c \in C, \tilde{p}^{dyn}_{res}(c) = \hat{p}^{dyn}_{res}(c) - i \cdot \left(1 - \frac{\hat{p}^{dyn}_{res}(c) - i}{\hat{p}^{dyn}_{res} - i}\right) \tag{4.13}$$

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

$$\forall c \in C, \left\lceil \tilde{p}^{dyn}_{res}(c) \right\rceil = \frac{p^{dyn}_{res} \cdot \tilde{p}^{dyn}_{res}(c)}{\hat{p}^{dyn}_{res}} \tag{4.14}$$

This approach also allows calculating a confidence interval of the power consumption of containers by scaling down the observed global error:

$$\forall c \in C, \epsilon_{res}(c) = \frac{\tilde{p}^{dyn}_{res}(c)}{\hat{p}^{dyn}_{res}} \cdot \epsilon_{res} \tag{4.15}$$

In order to 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 number of samples required before attempting to learn a power model (default: 10)
- Size of the history window used to retain samples (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 focuses on model accuracy when compared to RAPL values, rather than values obtained from an external source of power data. The SmartWatts validation evaluated the quality of power estimation under sequential and parallel workloads, the accuracy and stability of power models, and the overhead of the *sensor* component. Standard benchmarks like Stress-NG and NAS Parallel Benchmarks were used.

While no advanced statistical analysis was conducted, the validation shows that, for an error threshold of 5 Watts for CPU and 1 Watt for DRAM, power consumption can be reliably estimated with errors below 3 Watts and 0.5 Watts, respectively. The only case where the error exceeds the threshold is at the CPU idle frequency. Model stability is shown to improve significantly when lower recalibration frequencies are used. SmartWatts successfully reuses a given power model for up to 594 estimations, depending on frequency. Monitoring overhead is observed to be 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 negligible.

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 excels 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 provide 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.

Chapter 5

Conclusion and Future Work

5.1 Summary of Findings

This thesis confirms that attributing energy consumption to containerized workloads remains an inherently complex and uncertain task. While existing tools can provide detailed measurements, none can achieve exact results: fundamentally, every method introduces estimation errors due to limitations in data sources and methodological assumptions. Measurement uncertainty is unavoidable and varies depending on the chosen metrics and approach.

The analyzed tools demonstrate significantly different design philosophies, largely reflecting their intended audiences and operational environments. For example, Kepler's earlier versions (v0.9.0) prioritized measurement granularity and resource-level accuracy, relying on eBPF instrumentation and hardware counters to achieve fine-grained attribution. In contrast, Kepler's latest version (v0.10.0) adopts a simplified, CPU-time-based attribution model that sacrifices accuracy in favor of deployability, security, and operational stability, suggesting a strategic shift.

The intended audience of a measurement tool is a critical factor influencing its architecture. Researchers and infrastructure engineers working on system optimization may demand maximal granularity and accuracy, accepting additional overhead and complexity. Developers, in contrast, often seek only high-level visibility into container-level energy consumption and may prefer black-box tools that abstract away implementation details. Cluster operators balance energy awareness with priorities such as security, availability, and maintainability; conditions that typically preclude privileged monitoring techniques like eBPF.

Across all tools, Intel's RAPL interface emerges as the most reliable and promising source of real-time energy telemetry for CPU and DRAM domains. Its millisecond-level update frequency theoretically enables highly granular measurements. However, most tools (including Kepler) fail to exploit this potential. Multi-second sampling intervals remain standard, reducing the fidelity of captured workload dynamics and limiting attribution accuracy, especially for heterogeneous workloads characterized by short-lived or bursty processes. Granular temporal analysis is thus identified as a key requirement for accurate energy attribution in complex environments. Fine-grained tracking better captures workload diversity and transient behavior but introduces computational overhead. Tools attempt to balance these competing concerns, but no clear consensus or universally optimal strategy emerges.

Another important trade-off lies between estimation and accuracy. While perfect energy attribution is unattainable, approximate models provide valuable insights even at lower precision. This thesis argues that estimation remains worthwhile, especially in multi-tenant and dynamic environments where detailed hardware telemetry is unavailable or incomplete.

Notably, all examined tools focus exclusively on CPU, RAM, and GPU energy consumption. No attempt is made to estimate power consumption of other system components such as storage devices or network interfaces. Extending measurement capabilities beyond the core compute elements presents a significant research opportunity. While such estimations are inherently less precise, even approximate visibility could enhance observability and inform optimization efforts.

In conclusion, energy attribution in containerized systems is a balancing act between detail and practicality. Existing tools demonstrate that both high-accuracy and operational simplicity are valid design goals, serving different user groups and system environments. RAPL remains central to accurate measurement, but practical deployment constraints often limit its effective use. Achieving reliable and actionable power monitoring requires careful architectural decisions, guided by the specific needs of the intended audience and the realities of the deployment environment.

5.2 Critical Reflection

5.2.1 Methodological Reflection

This thesis adopts a purely analytical approach, centered on literature review, code analysis, and architectural evaluation of existing container-level energy attribution tools, as well as potential sources of information. This methodological choice aligns with the intended scope of the project: as a VT2-level research work, the thesis was designed to provide a theoretical foundation for subsequent practical development.

While the thesis proposes concrete recommendations for future tool design (see § 5.3), it deliberately refrains from developing a full tool architecture or implementation. Instead, the findings are intended to inform such efforts in future research, particularly within the scope of the author's upcoming master's thesis.

While empirical validation through experimental benchmarking was deliberately omitted to prioritize tool coverage and architectural analysis, this remains a methodological limitation. Validation insights from existing literature were integrated where available.

The focus on Linux and Kubernetes environments further narrows the applicability of the findings. Alternative infrastructures, proprietary tools or telemetry sources (e.g. OEM-specific BMC energy reporting interfaces) were not explored in depth. This reflects both the author's area of expertise and the practical relevance of Kubernetes in modern cloud environments but limits generalizability.

Despite these limitations, the thesis's core strengths lie in its detailed inspection of source metics, tool architectures and source code, combined with a practical understanding of Kubernetes-based deployments. This approach allowed the identification of undocumented behaviors, implementation inconsistencies, and unaddressed design trade-offs in tools such as Kepler, Scaphandre, and SmartWatts.

In summary, while empirical validation and cross-environment generalization are lacking, the thesis successfully fulfills its role as a theoretical exploration of container energy attribution, providing a solid analytical foundation for future tool development and evaluation.

5.2.2 Tool Adoption in Real-World Systems

A critical reflection of container-level energy attribution tools reveals significant barriers to their adoption in production environments. Chief among these are security concerns and the need for privileged access. Tools that rely on kernel-level instrumentation, such as eBPF or `perf_event_open`, often require elevated permissions, introducing potential security risks. For most cluster operators, energy monitoring remains a secondary concern compared to reliability, availability, and security. This limits the practical deployment of high-precision tools in real-world systems.

Operational simplicity frequently outweighs measurement accuracy. Tools with complex configurations, hardware-specific dependencies, or non-standard export interfaces are typically avoided, even if they promise higher measurement fidelity. In practice, both industry users and academic researchers often treat energy monitoring tools as black boxes. This is evident from the widespread use of tools like Kepler and Scaphandre without detailed validation or configuration tuning. While these tools offer configurability, leveraging it requires substantial technical understanding, which many users lack or are unwilling to invest.

As a result, most energy monitoring tools prioritize usability over accuracy. Prometheus integration, simple deployment (e.g. via Helm charts), and minimal security concerns are often deemed more important than methodological rigor. No tool analyzed in this thesis explicitly targets researchers seeking maximal measurement accuracy; instead, tools implicitly address operators who require straightforward observability solutions.

Finally, tool design often fails to recognize the fundamentally different needs of developers, operators, and researchers. Currently, no single tool effectively caters to all these audiences. This segmentation of user requirements, combined with practical deployment constraints, helps explain why most existing tools settle for operational simplicity at the expense of measurement accuracy.

5.2.3 Transparency, Trust, and Black-Box Measurement

A fundamental challenge in container-level energy attribution is the reliance on inherently opaque measurement interfaces. Critical telemetry sources such as Intel RAPL, NVIDIA NVML, and platform-level BMC sensors provide essential power and energy data, yet their internal operation and measurement scopes remain poorly documented. For instance, both Scaphandre and Kepler developers acknowledge uncertainty regarding the exact coverage of RAPL domains, most notably `PKG` and `PSys`. This lack of clarity complicates both tool implementation and the interpretation of reported energy metrics.

Black-box measurement interfaces hinder the development of accurate and explainable energy monitoring tools. When the underlying telemetry mechanisms are closed, tool developers are forced to make assumptions which propagate into the attribution models and directly impact reported results. Without visibility into how energy

counters are computed or which hardware components are included, users cannot fully trust the reported energy consumption data, nor can they debug unexpected results.

To address this, future energy monitoring frameworks should:

- Clearly document all assumptions related to telemetry sources and attribution models.
- Provide visibility into the source and scope of every reported metric.
- Encourage open standards for energy telemetry, advocating for greater transparency in RAPL, NVML, and similar interfaces.

5.2.4 Energy Attribution Philosophies

The analysis confirms that energy attribution models (container-centric, shared-cost, and residual modeling) reflect fundamentally different perspectives. What is considered a "fair" distribution depends on the user's priorities: developers, operators, or researchers will each favor different approaches.

Current tools often make implicit attribution decisions, especially regarding idle power and system processes, without clear documentation. This obscures how reported metrics should be interpreted.

To improve transparency and usability, future tools should:

- Clearly document their attribution model.
- Where feasible, allow users to choose between attribution strategies.
- Make residual power explicit rather than hidden in shared costs.

5.3 Recommendations for Future Tool Development

5.3.1 Towards Maximum-Accuracy Measurement Tools

Future container-level monitoring tools should prioritize temporal resolution, metric flexibility, and modular attribution models to maximize measurement accuracy. Based on the findings of this thesis, the following design principles are recommended:

High-Resolution Hardware Metrics RAPL-based measurements should support sub-second sampling intervals configurable by the user. Intervals as low as 50 milliseconds (close to RAPL's practical resolution limit) would enable significantly finer-grained power attribution, especially in heterogeneous or bursty workloads. Critically, power readings should be attributed directly upon collection, avoiding fixed aggregation cycles that dilute temporal precision and introduce attribution inaccuracies.

Decoupled Metric Handling Metric collection loops should differentiate between high-frequency (e.g. RAPL, eBPF) and low-frequency (e.g. Redfish, IPMI) sources.

Separating high-priority, high-frequency metrics from slower telemetry sources minimizes performance overhead and maximizes the utility of each metric type.

Multi-Metric Integration Tools should support combining diverse telemetry sources, such as RAPL, Redfish, ACPI, and BMC, in a coherent manner. Coarse-grained metrics (e.g. Redfish node-level power) can be fused with fine-grained metrics (e.g. RAPL domain-level power) to interpolate or validate measurements. However, care must be taken to preserve the distinction between direct measurements and model-based estimations when combining such sources.

User-Configurable Estimation Modules Modular estimation frameworks should be employed for subsystems lacking direct telemetry, such as storage devices or network interfaces. Default models can provide reasonable estimates based on automatic device detection (e.g. storage device type, or link speed for network interfaces). However, advanced users should be able to override idle, maximum, and typical power values to refine model accuracy.

Selectable or configurable Attribution Models Energy attribution should support multiple modeling approaches, ideally selectable at runtime. Examples include container-centric models, shared-cost models, or hybrid methods. Importantly, idle and system-level energy consumption should be accounted for explicitly, not implicitly merged into container totals, improving transparency and accuracy.

Self-Calibration Support Tools should offer automated or semi-automated calibration methods, such as idle power calibration or workload-based calibration inspired by Kavanagh et al. [21]. Where possible, standardized interfaces for integrating external measurement devices should be considered, enabling users to validate or refine energy models via external power meters.

Standards-Based Implementation Wherever feasible, tools should adhere to standardized system interfaces such as the Linux `powercap` framework for RAPL access, avoiding proprietary solutions. This facilitates long-term maintainability and eases deployment across heterogeneous environments.

In summary, a high-accuracy monitoring tool must prioritize both technical rigor and architectural flexibility. Drawing from design strengths observed in Kepler, KubeWatt, Scaphandre, and SmartWatts, future tools should offer high-resolution measurements, modular estimation, and transparent energy attribution as core design objectives.

5.3.2 Addressing Missing Domains: Disk, Network, and Others

No current container-level energy monitoring tool provides direct measurements or estimations for storage devices or network interfaces. Nevertheless, for a comprehensive understanding of node-level energy consumption, these components should not be neglected.

Modular Estimation Frameworks Future tools should include optional, modular estimation models for disks and network interface controllers (NICs). Such models could leverage existing system metrics such as I/O request counts, throughput

rates, or link speeds as input signals. For example, storage energy estimation could differentiate between SSDs and HDDs based on device identification, using I/O operations as a proxy for activity levels. Similarly, NIC models could base estimations on transmitted and received data volumes or link activity states.

Inherent Accuracy Limitations These estimations will inevitably remain less accurate than direct telemetry from hardware sensors. However, including such models can enhance the completeness of node-level energy consumption analysis, particularly in environments where disks and NICs constitute non-trivial portions of total power draw.

Residual Energy Utilization In cases where total node power is known (e.g. via Redfish or BMC sensors) and major contributors like CPU and memory are directly measured, residual energy (the unaccounted portion) could be partially attributed to storage and networking components. However, reliance on residual energy must be approached cautiously, as it risks compounding measurement and attribution errors.

Configurability Estimation models should remain user-configurable. While default values enable ease of use for casual users, advanced users should be able to fine-tune idle power, maximum power, and activity-to-power correlation parameters to improve estimation accuracy.

In summary, although disk and network power estimations are inherently imprecise, including them in a modular and configurable manner would significantly enhance the practical value of container-level energy monitoring tools.

5.3.3 Balancing Accuracy and Overhead

The pursuit of maximum measurement accuracy inevitably increases monitoring overhead. Future tools should address this trade-off by offering distinct operational modes, allowing users to select between accuracy and resource efficiency based on their specific needs.

'Precision' Mode In this mode, all available telemetry sources and fine-grained attribution models should be enabled. High-frequency sampling intervals, detailed container-level breakdowns, and optional estimations for secondary components (e.g. disks, NICs) provide maximal measurement detail. This mode is intended for research, validation, or auditing scenarios where energy transparency is prioritized over runtime performance.

'Lightweight' Mode Conversely, a lightweight mode should disable high-frequency probes, omit low-relevance subsystems, and focus on core power consumers such as CPU and memory. Sampling intervals can be relaxed, and coarse-grained metrics prioritized. This configuration is suitable for production environments where minimizing monitoring overhead is critical.

Mode Selection Not all environments or users require maximum accuracy. By providing predefined operational modes, tools can adapt to a wide range of use cases

without forcing users to manually configure every parameter. However, manual overrides should remain possible for expert users seeking fine control.

In summary, supporting both 'precision' and 'lightweight' modes allows monitoring tools to serve diverse operational contexts without compromising on flexibility or usability.

5.3.4 Supporting Multiple User Roles and Needs

Container-level energy monitoring tools must accommodate a diverse range of users, each with distinct goals and expectations. A future-proof tool should address these needs through architectural modularity, clear defaults, and extensive documentation.

Developers Developers typically seek simple, per-container energy consumption totals to guide software optimization. Their focus is on understanding the energy impact of specific applications or containers, without concern for the idle power waste or baseline energy consumption of the broader system. Minimal setup complexity and straightforward metric outputs are priorities for this user group.

Operators Infrastructure operators require system-wide energy observability, not limited to individual containers. Their goals include identifying idle energy waste, optimizing resource utilization, and avoiding performance degradation due to throttling or resource contention. Operators value transparency, stability, and actionable insights across the entire infrastructure stack.

Researchers Researchers demand the highest levels of accuracy, configurability, and architectural transparency. They require detailed documentation of attribution models, known limitations, and telemetry sources, alongside access to raw metrics and calibration options. Flexibility and reproducibility are critical requirements for this audience.

Serving All Audiences To accommodate these varied needs, monitoring tools should adopt a modular architecture with:

- Sensible, production-ready defaults for black-box usability.
- Optional advanced configuration layers for expert users.
- Clear and comprehensive documentation describing methods, assumptions, and limitations.

Recognizing that many users will treat the tool as a black box, default configurations must produce reasonable, usable results without requiring manual tuning. However, advanced users should retain the ability to inspect, customize, and extend the tool's behavior.

In summary, serving developers, operators, and researchers simultaneously requires balancing simplicity, flexibility, and transparency within the tool's design.

5.3.5 Energy Metrics for Virtualized Environments

Energy attribution within virtualized environments, particularly for Kubernetes clusters running inside virtual machines (VMs), remains a challenge. Existing monitoring tools primarily target bare-metal deployments, leaving a significant gap in energy observability for cloud-based and virtualized infrastructures.

Existing Approaches Two conceptual approaches have been explored:

- **Scaphandre's QEMU Passthrough:** Implements a basic export mechanism by writing host-side energy metrics to a virtual file system accessible by guest VMs running an identical instance of Scaphandre. This approach is functional but but limited, since input and output metrics must correlate to serve identical instances.
- **Kepler's Hypercall Concept:** Proposes using hypercalls as a mechanism for host-to-guest metric transfer. However, this concept remains unimplemented.

Future Directions Future monitoring tools should prioritize the development of standardized telemetry export mechanisms to enable accurate energy monitoring within VMs. Potential approaches include:

- Hypervisor-supported hypercalls specifically designed for energy metrics.
- Virtio-based APIs to expose host-side telemetry directly to guest VMs.
- Enhanced QEMU or container runtime interfaces capable of exporting power data.

Relevance Given the prevalence of virtualized infrastructure in modern cloud environments, particularly for managed Kubernetes platforms, solving this problem would significantly broaden the applicability and adoption of energy observability solutions.

In summary, enabling reliable host-to-VM energy metrics passthrough represents a critical development priority for future container-level energy monitoring tools.

5.3.6 Standardization and Hardware Vendor Transparency

Measurement accuracy in container-level energy monitoring is fundamentally limited by the availability and transparency of hardware telemetry. Addressing these constraints requires both industry-wide standardization efforts and increased openness from hardware vendors.

Hardware Vendor Responsibility Vendors should provide native power telemetry for additional system components, such as network interface cards (NICs), storage devices, and peripheral subsystems. These metrics should be accessible via standardized, open interfaces to facilitate direct measurement and reduce reliance on model-based estimations.

Expanding Telemetry Standards Existing interfaces like Redfish, which currently expose node-level power data, could be extended to include per-component power

reporting. Similarly, the adoption of open, vendor-neutral standards for exposing telemetry at the subsystem level would significantly enhance energy observability.

RAPL Transparency The lack of public documentation regarding Intel's Running Average Power Limit (RAPL) interface remains a barrier to fully understanding and validating the reported power domains. Vendors should disclose the internal structure and calculation methods of such telemetry systems to resolve current 'black-box' concerns identified by tool developers, including Kepler and Scaphandre contributors.

In summary, improving measurement precision at the workload level depends not only on software design but also on cooperation from hardware manufacturers and industry standards bodies. Transparent and standardized telemetry interfaces represent a critical enabler for future progress in energy observability.

5.4 Broader Research and Industry Opportunities

The advancement of energy observability in containerized environments extends beyond tool development. Broader collaboration across industry stakeholders and research communities is essential to drive progress.

Role of Standards Organizations Organizations such as the Cloud Native Computing Foundation (CNCF) and Kubernetes Special Interest Groups (SIGs) could play a central role in promoting standard APIs for energy metric collection and dissemination within cloud-native environments. Establishing best practices and reference implementations would encourage adoption and consistency across tools.

Hardware Manufacturer Involvement Hardware vendors are positioned to directly influence the quality and availability of energy telemetry data. By exposing accurate and accessible power metrics across all major system components, manufacturers can enable precise energy measurement without reliance on coarse or estimated models. Collaboration with open-source communities could further support development of vendor-agnostic solutions.

Sustainable Cloud Computing Energy observability should be recognized as a foundational component of sustainable cloud computing. Accurate energy measurement at the workload level enables informed optimization decisions, supports regulatory compliance, and advances corporate sustainability goals. As such, energy transparency should be integrated into both research agendas and industry roadmaps for cloud infrastructure development.

In summary, advancing energy observability requires coordinated efforts spanning tool developers, standards bodies, hardware vendors, and sustainability-focused initiatives.

5.5 Closing Remarks

Energy consumption measurement at the container or workload level remains a complex and evolving challenge. This thesis has identified key architectural and

methodological features that future monitoring tools should incorporate to enhance measurement accuracy, architectural flexibility, and practical usability.

However, progress in this field is constrained not only by technical trade-offs (such as balancing accuracy against monitoring overhead) but also by external limitations. Chief among these are the lack of transparent hardware telemetry, incomplete standardization of power reporting interfaces, and the absence of established methodologies for energy attribution in virtualized environments.

The recommendations presented in this chapter are intended to guide both tool developers and researchers. By integrating high-resolution hardware metrics, modular estimation frameworks, user-configurable attribution models, and standardized interfaces, future tools can advance the state of energy observability in containerized infrastructures.

Ultimately, energy-aware computing practices will become increasingly relevant as the industry shifts towards sustainable cloud operations. Improved energy transparency at workload granularity represents a critical foundation for enabling these efforts.

Appendix B

Powerstack

Implementation of an energy monitoring environment in Kubernetes

Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

SPECIALIZATION PROJECT 1

Powerstack: Implementation of an energy monitoring environment in Kubernetes

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
JANUARY 31, 2025

Re-edited on
January 31, 2026

Study program:
Computer Science, M.Sc.

Imprint

Project: Specialization Project 1
Title: Powerstack: Implementation of an energy monitoring environment in Kubernetes
Author: Caspar Wackerle
Date: January 31, 2025
Keywords: process-level energy consumption, 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

Energy efficiency in cloud computing has become a critical concern as data centers consume an increasing share of global electricity. This thesis investigates energy consumption at the container and node level in Kubernetes-based infrastructures, using KEPLER (Kubernetes-based Efficient Power Level Exporter) to monitor and analyze power consumption in a controlled test environment.

A bare-metal Kubernetes cluster was deployed on three identical servers, configured using K3s for lightweight orchestration and managed through Ansible for full automation. The entire system was designed to be fast to deploy, highly reproducible, and adaptable to different hardware environments. Configurations were centralized for easy reusability in future projects, ensuring that modifications could be made with minimal effort. Prometheus and Grafana were integrated to collect and visualize KEPLER's real-time energy consumption metrics. A series of controlled benchmarking experiments were conducted to stress CPU, memory, disk I/O, and network I/O, assessing KEPLER's accuracy in reporting power usage under varying workloads.

The results indicate that KEPLER credibly tracks workload-induced power variations at the CPU package level, though inconsistencies arise in non-CPU power domains. High idle power consumption was observed at the node level, suggesting that infrastructure energy efficiency must account for static consumption beyond dynamic workloads.

This thesis provides a foundation for further research into energy-efficient Kubernetes environments, including improving KEPLER's accuracy, extending workload profiling, and exploring automation-driven energy optimization strategies. The modular and automated deployment architecture ensures that the findings and methodologies can be readily adapted for use in other energy-related cloud research projects.

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

Chapter 1

Introduction and Context

1.1 Significance of Energy Efficiency in Cloud Computing

Cloud computing has reshaped how computing resources are provisioned and consumed, offering efficiency gains through large-scale resource sharing. While economic benefits (such as reduced operational costs and improved scalability) are widely recognized, energy efficiency and environmental impact are equally important considerations. By enabling higher utilization of shared infrastructure, cloud computing can reduce energy waste and support global sustainability objectives.

The rapid growth of cloud adoption has transformed it into a central component of global IT infrastructure. Hyperscalers such as Amazon Web Services, Google Cloud, and Microsoft Azure are now major consumers of electrical power, drawing increasing attention from policymakers and environmental organizations. Although cloud providers have made significant investments in renewable energy procurement, the use of green power alone does not guarantee efficient energy utilization at workload level.

Technological advancements have continuously improved the energy efficiency of data centers, with modern facilities achieving Power Usage Effectiveness (PUE) values close to 1. However, PUE reflects facility-level efficiency and does not capture how effectively energy is used by individual workloads. Even at a theoretical PUE of 1, substantial waste may occur if computational resources are underutilized. This underscores the need to examine workload-level energy efficiency.

Containers, as a lightweight virtualization technology, improve resource density and often deliver better energy efficiency than traditional virtual machines (VMs). However, they also introduce new challenges for accurate energy measurement. Granular monitoring becomes more complex in containerized environments (especially in Kubernetes) due to dynamic resource allocation, scheduling, and autoscaling mechanisms.

Despite the importance of this topic, research on energy efficiency in cloud-native systems remains limited. While data center operations have been extensively optimized and green-coding practices are increasingly emphasized, the energy efficiency of Kubernetes clusters is comparatively underexplored. Addressing this gap is essential for aligning economic and environmental goals.

1.2 The Need for Energy-Efficient Kubernetes Clusters

Kubernetes has become the de facto standard for container orchestration, managing large-scale, distributed workloads. Its sophisticated features (dynamic scheduling, autoscaling, and distributed resource management) enable high performance and operational flexibility. However, these same features complicate energy measurement and optimization.

In many deployments, Kubernetes clusters run on virtual machines to simplify infrastructure management, adding an additional abstraction layer and making energy attribution more difficult. Achieving energy-efficient Kubernetes clusters therefore requires robust measurement techniques that can translate raw energy data into actionable insights. Such measurements form the basis for systematic optimization efforts.

Given the growing energy footprint of cloud infrastructures and the limited research on energy efficiency in Kubernetes, this remains a pressing and timely research area. By addressing this gap, the present work contributes to the broader objective of sustainable cloud computing.

1.3 Objectives and Scope of this Thesis

1.3.1 Context

This thesis is part of the Master's program in Computer Science at the Zurich University of Applied Sciences (ZHAW) and represents the first of two specialization projects. The current project (VT1) focuses on the practical implementation of a test environment for energy-efficiency research in Kubernetes clusters. The subsequent project (VT2) will examine theoretical foundations and methodological approaches to measuring and improving energy efficiency in such environments.

The work builds upon previous projects on performance optimization and energy measurement. EVA1 addressed topics such as operating-system tooling, statistical foundations, and eBPF, while EVA2 explored energy measurement across hardware, firmware, and software layers. These foundations inform the present thesis but will not be revisited in detail.

1.3.2 Scope

This thesis focuses on the practical implementation of a test environment, excluding detailed theoretical analysis and extensive literature review. The primary goal is to document the creation of a reliable and reproducible environment that supports future research on energy efficiency in Kubernetes clusters.

The work builds upon prior activities carried out in EVA1 and EVA2, leveraging existing knowledge while extending the research focus. Fundamental concepts from these earlier projects are assumed as background knowledge and are therefore not revisited in detail.

The EVA1 presentations introduced essential principles related to Linux performance monitoring, system optimization, and a conceptual understanding of eBPF. While these concepts play an important role in this research, they are not re-explained here,

as the emphasis of this thesis lies on their practical application within an energy-efficient Kubernetes cluster.

Similarly, the EVA2 presentations established foundational knowledge for understanding energy-consumption measurement across hardware, firmware, and software layers. Topics such as CPU power states (ACPI, C-states, P-states), Intel CPU performance-scaling drivers, and Intel RAPL for power monitoring and control are considered essential prerequisites but are not explicitly covered again in this thesis.

By building on these existing foundations, this thesis narrows its scope to the investigation of energy efficiency at the Kubernetes cluster level, incorporating relevant techniques from prior research where appropriate.

1.3.3 Objectives

The main objective is to design and implement a test environment that enables:

- Analysis of key parameters affecting energy efficiency in Kubernetes clusters.
- Reliable and consistent experimentation.
- Reproducibility and automation in deployment and configuration.

The resulting environment will form the basis for subsequent research projects.

1.3.3.1 Parameters for Analysis

The project aims to reuse established tools and components where feasible. The parameters to be analyzed include:

- CPU utilization and energy consumption.
- Memory usage and its impact on power draw.
- Disk I/O and storage-related power consumption.

Additional parameters may be incorporated following further evaluation.

1.3.3.2 Data Integrity and Persistence

Ensuring data integrity and persistence is critical for reliable analysis. Key requirements include:

- Persistent storage that survives system shutdown.
- A unified data store accessible by all nodes.
- Data retention across Kubernetes cluster reinstallations.
- The ability to power down unused worker nodes without data loss.

1.3.3.3 Reproducibility and Automation

Reproducibility and automation are additional goals aimed at improving research efficiency. Benefits include:

- Simplified recovery from misconfiguration through rapid redeployment.
- Reduced troubleshooting time.
- Improved stack cleanliness by eliminating residual configurations.

1.3.3.4 Security

Security, while not a primary focus, will be addressed by implementing basic best practices. Key measures include:

- Use of encrypted passwords.
- Adherence to standard Kubernetes security best practices.
- Minimization of potential vulnerabilities through careful configuration.

1.3.4 Use of AI Tools

During the writing of this thesis, *ChatGPT*[14] (Version 4, OpenAI, 2024) was used as an auxiliary tool to improve efficiency in documentation and technical writing. Specifically, it assisted in:

- Structuring and improving documentation clarity.
- Refining descriptions of code and technical implementations.
- Beautifying and formatting smaller code snippets.
- Assisting in LaTeX syntax corrections and debugging.

All AI-generated text was critically reviewed, edited, and adapted to 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.

1.3.5 Project Repository

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

Chapter 2

Architecture and Design

2.1 Overview of the Test Environment

The test environment consists of a Kubernetes cluster deployed on three bare-metal servers located in a university datacenter. The servers are identical in their hardware specifications and are connected through both a private network and the university network. This setup enables complete remote management and ensures direct communication between the servers for Kubernetes workloads. A detailed description of the hardware and network topology is provided below. Figure 2.1 illustrates the overall architecture and network configuration.

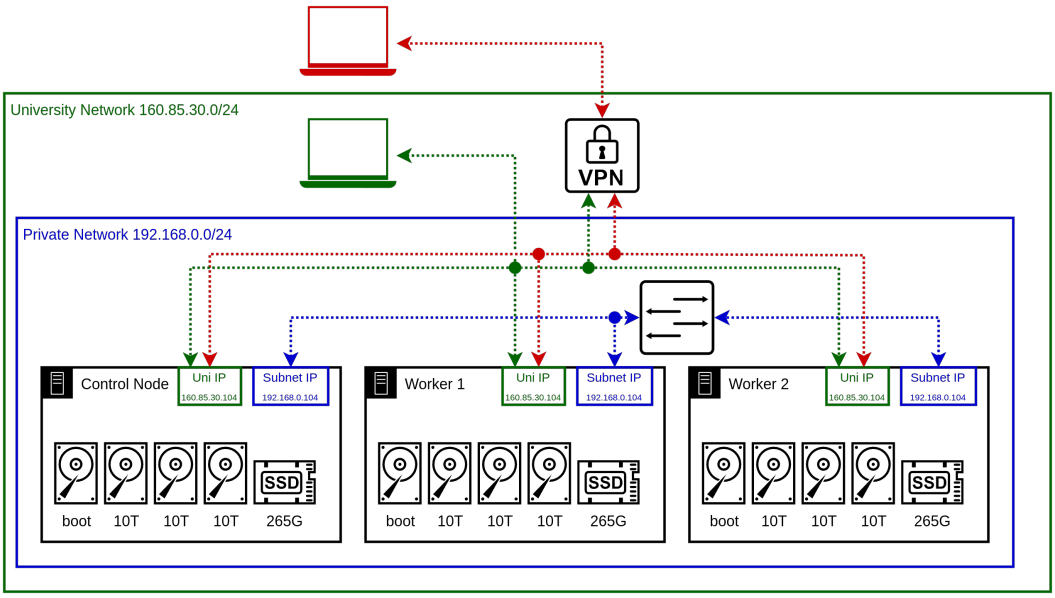


FIGURE 2.1: Physical Infrastructure Diagram

2.1.1 Hardware and Network

2.1.1.1 Bare-Metal Servers

The cluster is built using three identical Lenovo ThinkSystem SR530 servers, each equipped with the following hardware:

- CPU: 1 × Intel(R) Xeon(R) Bronze 3104 @ 1.70 GHz, 6 cores.

- Memory: 4 × 16 GB DDR4 DIMMs (total 64 GB RAM).
- Storage:
  - 2 × 32 GB M.2 SATA SSD (operating-system boot drive).
  - 1 × 240 GB 6 Gbps SATA 2.5″ SSD (persistent storage).
  - 3 × 10 TB 7200 RPM 12 Gbps SAS 3.5″ HDD (bulk storage).
- Power: Dual redundant power supplies.
- Cooling: 4 of 6 possible fans installed.
- Firmware:
  - BMC Version: 8.88 (Build ID: CDI3A4A)
  - UEFI Version: 3.42 (Build ID: TEE180J)
  - LXPM Version: 2.08 (Build ID: PDL142H)

Each server includes a Lenovo XClarity Controller (BMC) for remote management. Access via the BMC IP provides out-of-band management and monitoring capabilities.

2.1.1.2 Network Topology

The servers are connected using two distinct networks:

- **Private Network:** Each server has a private IP address (192.168.0.104–192.168.0.106), enabling direct, high-speed communication between nodes. This reduces load on the university network and improves performance for intra-cluster Kubernetes workloads.
- **University Network:** Public-facing IP addresses (160.85.30.104–160.85.30.106) allow access from within the university network, with external access available via VPN.

Note: Detailed switch and gateway configurations are maintained by the university IT department and are beyond the scope of this document.

2.2 Key Technologies

2.2.1 Ubuntu

Ubuntu was selected as the operating system primarily due to the author's familiarity with it. Additionally, it was already installed on the servers when they were received, which reduced setup time and complexity. While other Linux distributions are optimized specifically for Kubernetes, using a well-known distribution ensured a smoother initial configuration process.

2.2.2 Bare-Metal K3s

Deploying Kubernetes directly on bare-metal servers (without using a hypervisor or virtual machines) was a fundamental design decision to ensure direct access to hardware-level data. This enables Kubernetes components and monitoring tools to interact with the underlying hardware more effectively, an essential requirement for accurate energy-consumption measurements.

K3s was selected for several reasons:

- It is lightweight and therefore suitable for lower-powered servers while potentially reducing overall energy consumption.
- Despite its minimal footprint, it remains fully compatible with upstream Kubernetes, allowing standard resources and configurations to be used without modification.
- K3s includes optimizations for ARM-based systems, making it a convenient choice for homelab environments and flexible future deployments.
- The author had prior experience with K3s and Rancher, enabling a faster and more reliable setup.

2.2.3 Ansible, Helm, `kubectl`

For automation and deployment, Ansible and Helm were chosen. Helm and `kubectl` are standard tools for managing Kubernetes applications and resources, offering broad community support and extensive documentation.

Ansible was selected for its flexibility and its simplicity when managing server configurations across multiple nodes. Its agentless approach (requiring only SSH and Python on the target machines) makes it particularly suited for managing bare-metal servers.

2.2.4 Kube-Prometheus Stack

The Kube-Prometheus stack was chosen because it is the de facto standard for monitoring in Kubernetes environments. The project is mature, feature-rich, and integrates seamlessly with Kubernetes components. Installation and configuration via Helm are straightforward, and the wide availability of community resources simplifies troubleshooting.

2.2.4.1 Prometheus

Prometheus was selected as the primary monitoring tool due to its strong integration with Kubernetes. While it provides extensive capabilities, it also introduces overhead and is not well suited for sub-second or low-second intervals, as typical scrape intervals are longer. For use cases focused on container orchestration, where lifetimes tend to be longer, this limitation is acceptable.

2.2.4.2 Grafana

Grafana was included for its ability to provide intuitive, customizable visualizations of metrics collected by Prometheus. It enables efficient interpretation of complex data through dashboards and visual representations, making it a valuable component of the monitoring stack.

2.2.4.3 AlertManager

AlertManager is bundled with the Kube-Prometheus stack and is used to route and manage alerts generated by Prometheus. Although AlertManager was not utilized in this project, its presence is beneficial for potential future extensions, particularly in production-like scenarios involving alerting and incident response.

2.2.5 Kepler

2.2.5.1 Purpose of Kepler

KEPLER[106], the *Kubernetes-based Efficient Power Level Exporter*, is a project focused on measuring energy consumption in Kubernetes environments. It provides detailed power-consumption metrics at the process, container, and pod levels, addressing an increasingly important need for energy-efficient cloud computing.

With cloud providers and enterprises facing growing pressure to improve energy efficiency, Kepler offers a practical solution. By enabling detailed, near-real-time measurement of power usage, it bridges the gap between high-level infrastructure metrics and workload-specific energy data. This ability to attribute energy consumption to individual components makes Kepler a valuable tool for advancing energy-efficient Kubernetes clusters.

2.2.5.2 Limitations of Kepler

Despite its potential, Kepler exhibits several limitations in the context of this project:

- **Active development:** Kepler is still under active development, meaning that features and APIs may change frequently. Documentation is limited, and community support for troubleshooting is still evolving.
- **Complexity:** Kepler is a large system with a complex architecture. Adapting it beyond basic configuration requires a deep understanding of its internal structure, making custom changes or enhancements challenging without substantial expertise.

Although Kepler is not without drawbacks, it remains the most promising available solution for measuring energy consumption in Kubernetes environments. Consequently, this thesis places significant emphasis on evaluating Kepler's capabilities and identifying potential areas for improvement.

2.3 Architecture and Design

2.3.1 Kubernetes Cluster Design

The Kubernetes cluster is deployed on three bare-metal servers running Ubuntu. One server is designated as the control-plane node, while the remaining two serve as worker nodes. For simplicity and in line with the project scope, no high-availability (HA) configuration is used. The servers communicate via their internal IP addresses, ensuring direct node-to-node communication without traversing external networks.

All Kubernetes control-plane components such as the API server, controller manager, and scheduler run exclusively on the control-plane node, while workloads are distributed across all nodes. Figure 2.1 provides an overview of the system architecture, including major components and data flow.

2.3.2 Persistent Storage

Persistent storage is provided using the spare SSD installed in the control-plane server. A partition on the SSD is created, formatted with the BTRFS filesystem, and mounted as the primary storage location. The control-plane node hosts an NFS server, and the worker nodes mount the corresponding NFS share to access the shared storage.

This centralized approach was chosen because the control-plane node is guaranteed to remain powered on throughout all experiments, making distributed storage solutions such as CEPH unnecessary for this project.

Within the NFS share, separate directories are allocated for Prometheus and Grafana data. Persistent volumes (PVs) and persistent volume claims (PVCs) are created for each service. The size of the PVs is configurable at installation time, providing flexibility for future storage requirements.

2.3.3 Monitoring Architecture

The monitoring stack is deployed using the `kube-prometheus-stack` Helm chart. This stack bundles Prometheus, Grafana, and AlertManager, providing a complete monitoring and visualization solution for Kubernetes environments. Prometheus scrapes metrics from Kepler and key Kubernetes endpoints (such as the kubelet API) at configurable intervals. Grafana connects to Prometheus to visualize these metrics through customizable dashboards.

2.3.4 Metrics Collection and Storage

Kepler generates energy-related metrics by collecting data from several sources:

- **Hardware-level metrics:** Using `eBPF` and kernel tracepoints to gather low-level data such as CPU cycles and cache misses.
- **Power-related metrics:** Obtained through RAPL (Running Average Power Limit) and IPMI (Intelligent Platform Management Interface) to measure CPU and platform-level energy consumption.

- **Container-level metrics:** Retrieved via the Kubernetes kubelet API, which exposes cgroup resource usage for running containers and pods.

Kepler aggregates these data sources, computes power-consumption metrics, and exposes them in a Prometheus-compatible format. Prometheus scrapes the metrics and stores them as time-series data on persistent storage, enabling detailed analysis of resource-usage patterns over time.

Chapter 2.4 provides a brief overview of the Kepler architecture, with a focus on metrics collection and generation. Figure 2.2 illustrates the information flow across the monitoring stack.

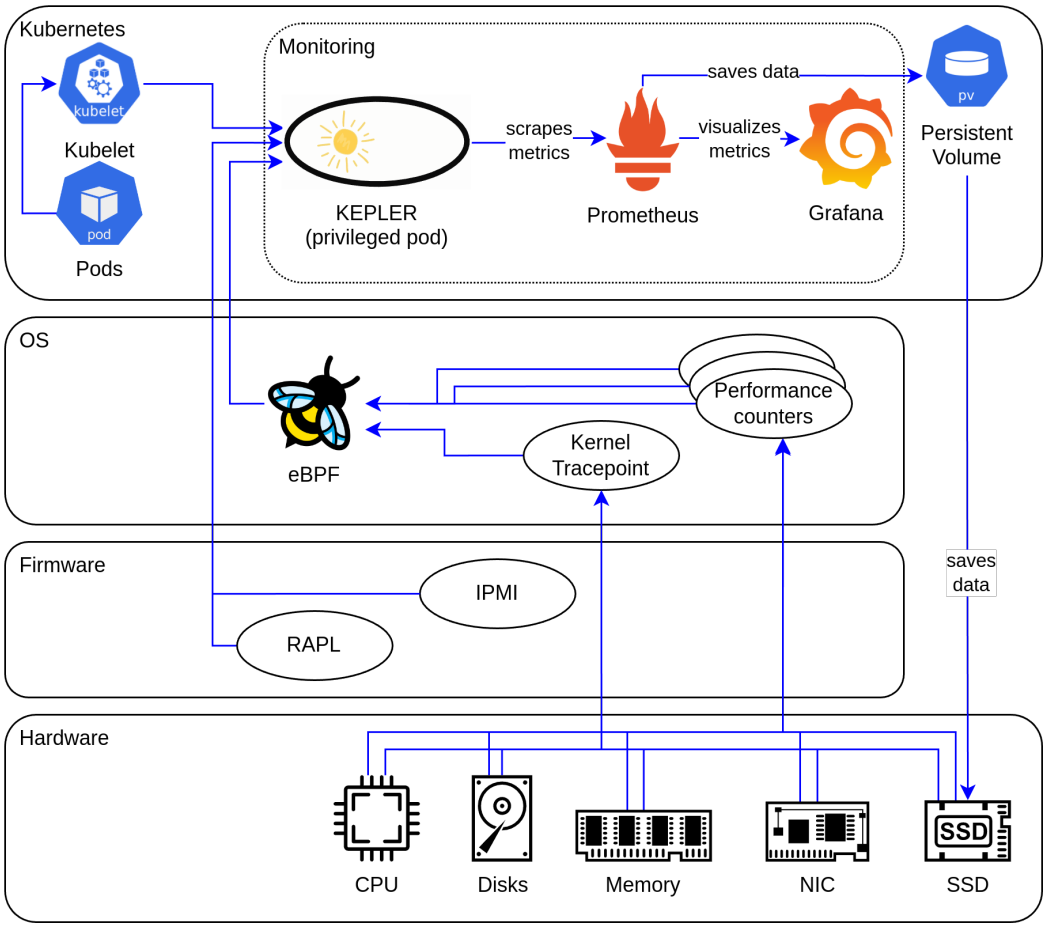


FIGURE 2.2: Monitoring data flow diagram of the entire stack

Given the substantial research and practical implementation already accomplished by the Kepler project, it was selected as a core component for this work.

2.3.5 Repository Structure

The repository for this project includes all aspects of the Kubernetes-based energy-efficiency test environment, from deployment automation to documentation. Due to the reliance on external projects, a hybrid dependency-management strategy was adopted.

2.3.5.1 Submodules for External Repositories

Several external projects that undergo frequent updates were forked and included as Git submodules. This approach allows easy customization and configuration while preserving the ability to synchronize with upstream repositories. Freezing submodules at specific commits ensures stability and avoids unexpected changes from upstream updates.

2.3.5.2 Direct Deployment from External Repositories

For external projects requiring minimal customization, direct deployment from their upstream repositories was chosen. This simplifies repository maintenance and ensures that stable, vetted versions are used.

2.3.5.3 Structure Overview

The repository is organized to maintain clarity and a clear separation of concerns:

- **ansible/:** Ansible playbooks and roles for automated deployment.
- **helm/:** Custom or external Helm charts managed through Ansible.
- **scripts/:** Bash scripts for executing Ansible playbooks.
- **config/:** Centralized configuration files and the Ansible vault.
- **docs/:** Documentation files describing setup and usage.
- **thesis/:** All thesis-related files, written in LaTeX.

2.3.6 Automation Architecture

Automation was a key focus in this project to ensure reproducibility, consistency, and ease of deployment. The automation architecture is primarily based on Ansible, with Helm commands embedded into Ansible playbooks for Kubernetes-specific deployments.

2.3.6.1 Ansible and Helm Integration

Ansible was used to automate the setup of the base environment, including system-level configurations and Kubernetes deployments. All Helm installations, such as the `kube-prometheus-stack`, were wrapped in Ansible playbooks. This approach provided a unified automation framework in which both system configurations and Kubernetes resources could be managed together. It also ensured clear version control and consistent logging of all deployment steps.

2.3.6.2 Execution Scripts

Custom Bash scripts were created to execute the Ansible playbooks. In addition to convenience, these scripts ensured:

- the correct execution context and configuration for invoking playbooks,
- automatic log creation, simplifying troubleshooting and auditing.

2.3.6.3 Centralized Configuration

All configuration values (such as IP addresses, storage paths, and deployment options) were centralized in a single configuration file. This design simplifies re-deployment on different hardware by requiring changes only in one location. When necessary, Jinja templates were used within Ansible to dynamically adapt configurations based on this central file.

2.3.6.4 Security

Sensitive information such as passwords and API keys was encrypted using an Ansible Vault. This allowed confidential data to be securely stored within the repository without compromising security during deployment.

2.4 Kepler Architecture and Metrics Collection

Because Kepler is a central component of this project, it is important to understand its architecture and how it collects metrics. This section provides a brief overview of Kepler's major components and data-collection approach. For more detailed information, the official Kepler documentation[**KeplerDocumentation**] should be consulted.

2.4.1 Kepler Components

2.4.1.1 Kepler Exporter

The core component of Kepler is the *Exporter*, which runs as a privileged DaemonSet pod on each Kubernetes node. The exporter interacts directly with the hardware and kernel to collect energy-consumption and resource-utilization metrics. It estimates power usage at the process, container, and pod levels and exposes the collected metrics in a Prometheus-compatible format.

A ServiceMonitor is also deployed, enabling Prometheus to scrape metrics from Kepler's exporter endpoints.

2.4.1.2 Kepler Model Server

Although the Kepler Model Server is not used in this project, its purpose is noteworthy. The model server provides power-estimation models at various granularities (node, pod, or component level). It may also include an online trainer that updates these models dynamically at runtime.

2.4.2 Kepler Data Collection

2.4.2.1 Process and Container Data

Kepler employs `eBPF` to collect detailed CPU event data. `eBPF` programs run in a privileged kernel context, allowing efficient, low-overhead monitoring of kernel-level events. Specifically, Kepler hooks into the `finish_task_switch` kernel function, which handles context switching, to collect process-level data. The following Perf counters are recorded:

- `PERF_COUNT_HW_CPU_CYCLES`

- `PERF_COUNT_HW_REF_CPU_CYCLES`
- `PERF_COUNT_HW_INSTRUCTIONS`
- `PERF_COUNT_HW_CACHE_MISSES`

By maintaining a BPF hash keyed by process IDs, cgroup IDs, CPU IDs, and timestamps, Kepler correlates collected events to individual processes and containers. This data forms the foundation for deriving energy-consumption estimates. The BPF hash structure is shown in Table 2.1.

TABLE 2.1: Hardware CPU events monitored by Kepler

Key	Value	Description
pid	pid	Process ID
	cgroupid	Process cgroupID
	process_run_time	Total time a process occupies the CPU (calculated each time the process leaves the CPU on a context switch)
	cpu_cycles	Total CPU cycles consumed by the process
	cpu_instr	Total CPU instructions consumed by the process
	cache_miss	Total cache misses by the process
	page_cache_hit	Total page-cache hits
	vec_nr	Total number of soft-IRQ handles (max 10)
	comm	Process name (max length 16)

2.4.2.2 CPU Power Data

Kepler leverages Intel RAPL (Running Average Power Limit) to monitor energy consumption across CPU domains such as cores, DRAM, and integrated GPUs. RAPL provides real-time power data with fine granularity, allowing Kepler to accurately measure CPU-related energy usage. The supported domains include:

- **Package (PKG):** Total energy consumption of the socket, including cores, caches, and memory controllers.
- **Power Plane 0 (PP0):** Energy consumption of CPU cores.
- **Power Plane 1 (PP1):** Energy consumption of integrated GPUs, if present.
- **DRAM:** Energy consumption of memory attached to the CPU.

To access RAPL data, Kepler uses the following methods (in order of preference):

1. **RAPL sysfs:** Direct access through the Linux power-capping framework at `/sys`. This requires root access and is the method used in this project.
2. **RAPL MSR:** Access via Model-Specific Registers, providing detailed energy readings.
3. **xgene-hwmon kernel driver:** Used on specific ARM architectures.

2.4.2.3 Platform Power Information

Kepler can also collect platform-level power data, representing the total power usage of the node. This is achieved through:

- **ACPI (Advanced Configuration and Power Interface):** Provides access to system-level power information.
- **IPMI (Intelligent Platform Management Interface):** Exposes power data via the Baseboard Management Controller (BMC).

2.4.3 Kepler Power Model

Kepler uses two complementary power-modeling approaches. If total node power is known, Kepler applies a ratio-based model to derive finer-grained power figures for individual components at the node and container levels. If detailed hardware-level readings are unavailable (for example, in virtualized environments) Kepler estimates power consumption from system-utilization metrics using a pretrained model (currently based on an Intel Xeon E5-2667 v3 processor). Because this model is processor-specific, it is inherently flawed when applied to other architectures. Increasing the number of available models is therefore a long-term goal of the project.

In previous experiments conducted by the author, Kepler was deployed on a Kubernetes cluster with virtualized nodes in an OpenStack environment. With no hardware-level power data available, Kepler attempted to estimate power consumption solely from system metrics. The resulting estimates were inconsistent and unreliable, underscoring the importance of accurate hardware data for meaningful energy analysis.

2.4.4 Metrics Produced by Kepler

Kepler collects and exports a wide range of metrics related to energy consumption and resource utilization.

2.4.4.1 Container-level Metrics

At container level, Kepler estimates total energy consumption in joules. Energy usage is broken down into the following components: cores, DRAM, uncore (such as last-level cache and memory controllers), total CPU package, GPU, and other. Additional resource-utilization metrics include total CPU time, cycles, instructions, and cache misses. Several IRQ-related metrics are also provided, such as the number of transmitted and received network packets and the number of block I/O operations.

2.4.4.2 Node-level Metrics

At node level, Kepler again estimates total energy consumption in joules. Energy estimates are provided for the whole node as well as the Core, DRAM, Uncore, CPU package, GPU, Platform, and Other categories. Kepler also exposes node-specific metadata (such as CPU architecture), aggregated metrics used by the Kepler model server, and Intel QAT utilization.

Chapter 3

Implementation

This chapter describes the implementation and configuration of the various components used in this project. All automation scripts are designed to be idempotent and are executed using shell scripts located in the `Powerstack/scripts` directory. In general, configuration is performed via the central configuration file (`Powerstack/configs/inventory.yml`), unless stated otherwise. Sensitive information is stored in the Ansible Vault file (`Powerstack/configs/vault.yml`).

3.1 K3s Installation

This section outlines the steps involved in setting up a Kubernetes cluster using K3s on bare-metal servers. Installation was automated using an Ansible playbook forked from the official `k3s-io/k3s-ansible`[116] repository, with customizations for internal IP-based communication.

3.1.1 Preparing the Nodes

Before running the Ansible playbook, the following prerequisites must be met on all servers:

- **Operating system:** Ubuntu 22.04 (kernel version 5.15.0).
- **Passwordless SSH:** A user with sudo privileges must have passwordless SSH access to each server.
- **Networking:** Each server must have both an internal IP (for cluster traffic) and an external IP (for VPN or external management).
- **Local Ansible control node setup:**
  - **ansible-community** 9.2.0 (version 8.0+ required).
  - **Python** 3.12.3 and **Jinja** 3.1.2.
  - **kubectl** 1.31.3.

3.1.2 K3s Installation with Ansible

The playbook supports x64, arm64, and armhf architectures. For this project, it was tested only on x64.

3.1.2.1 Configuration Details

- Internal and external IP addresses for all servers must be defined.
- One server must be designated as the control-plane node.
- Default values such as `ansible_user`, `ansible_port`, and the `k3s_version` may be adjusted if necessary.

3.1.2.2 Kubectl Configuration

- The playbook automatically installs and configures `kubectl` on the Ansible control node by copying the Kubernetes config file from the control-plane node.
- The user must rename the copied file from `config-new` to `config` and select the PowerStack context using:
  `kubectl config use-context powerstack`

3.2 NFS Installation and Setup

3.2.1 NFS Installation with Ansible

The NFS server and clients were fully automated via an Ansible playbook. Before beginning the automated setup, the following manual step must be completed:

- **Disk selection:** A disk must be selected on the control-plane node for persistent storage. This disk will be reformatted and all existing data will be lost.

The Ansible playbook performs the following actions:

- **Disk preparation:** The selected disk is partitioned (if required) and formatted with a single Btrfs partition occupying the full disk. The partition is mounted at `/mnt/data`, and an entry is added to `/etc/fstab` for persistence across reboots.
- **NFS server setup:** The `nfs-kernel-server` package is installed and configured on the control-plane node. The directory `/mnt/data` is exported as an NFS share for the worker nodes.
- **NFS client setup:** On each worker node, the `nfs-common` package is installed. The NFS share is mounted, and an `/etc/fstab` entry is added to ensure persistence.

3.2.1.1 Configuration Details

- The NFS network range must be specified, and all nodes must be part of that network.
- The export path must be defined.

3.3 Rancher Installation and Setup

3.3.1 Rancher Installation with Ansible and Helm

Although not strictly required for the project, Rancher was deployed in the `cattle-system` namespace to support debugging and system analysis. The installation was automated using an Ansible playbook that integrates Helm. The key steps were:

- **Helm installation:** Helm was installed on the control-plane node to deploy Rancher and its dependencies.
- **Namespace creation:** The `cattle-system` namespace was created for the Rancher deployment.
- **Cert-Manager deployment:** Cert-Manager was installed to manage TLS certificates.
- **Rancher deployment:** Rancher was installed using the official Helm chart. During installation:
  - **Hostname:** A hostname was defined for accessing Rancher.
  - The chart was configured with `-set tls=external` to enable external access.
  - **Bootstrap password:** A secure bootstrap password was set for the default administrator account.
- **Ingress configuration:** An ingress resource was created to route traffic to Rancher via the defined hostname.

3.4 Monitoring Stack Installation and Setup with Ansible

The monitoring stack (Prometheus, Grafana, and AlertManager) was deployed using the `kube-prometheus-stack`[117] Helm chart from the `prometheus-community/helm-charts` repository. Although the repository was forked for convenience, no upstream modifications were made, ensuring compatibility with future updates.

3.4.1 Prometheus and Grafana Installation with Ansible and Helm

The installation was automated using Ansible roles, ensuring idempotency and centralized configuration management. The following key steps were executed:

- **Persistent Storage Configuration:**
  - Directories for Prometheus, Grafana, and AlertManager were created on the NFS-mounted disk.
  - A custom `StorageClass` was defined for NFS storage. The default `local-path` `StorageClass` was overridden to ensure it is no longer the default.

  - PersistentVolumes (PVs) were created for Prometheus, Grafana, and AlertManager. A PersistentVolumeClaim (PVC) was created explicitly for Grafana, while the PVCs for Prometheus and AlertManager were managed by the Helm chart.

- **Helm Chart Installation:**

  - A Helm values file was generated dynamically using a Jinja template. This template incorporated variables from the central Ansible configuration file to ensure consistency. Sensitive information, such as the Grafana admin password, was included in the values file and removed from the control node after installation to mitigate security risks.

  - The Helm chart was installed via an Ansible playbook. The following customizations were applied through the generated values file:

    * PVC sizes for Prometheus and AlertManager were set based on the central configuration.

    * A Grafana admin password was defined.

    * Prometheus scrape configurations were extended to include Kepler endpoints.

    * Changes to the `securityContext` were applied to allow Prometheus to scrape Kepler metrics.

- **Service Port Forwarding:**

  - Prometheus, Grafana, and AlertManager services were exposed using static `NodePort`s defined in the central configuration file, enabling external access.

- **Cleanup:**

  - A cleanup playbook was executed to remove sensitive configuration files from both the control-plane node and the Ansible control node.

3.4.2 Removal Playbook

An Ansible playbook was created to handle complete uninstallation of the monitoring stack. This ensures that all PVs and PVCs are explicitly removed, avoiding residual artifacts in the Kubernetes cluster.

3.5 Kepler Installation and Setup with Ansible and Helm

3.5.1 Preparing the Environment

The Kepler deployment uses the official Kepler Helm chart repository. Before deploying Kepler, several prerequisites must be met to ensure correct operation.

3.5.1.1 Redfish Interface

The Redfish Scalable Platforms Management API is a RESTful API specification for out-of-band systems management. On the Lenovo servers used in this project, Redfish exposes IPMI-based power metrics, which Kepler accesses through its Redfish interface. To verify Redfish functionality, navigate to the Lenovo XClarity Controller and ensure that the following setting is enabled:

- **IPMI over LAN:** Located under `Network → Service Enablement and Port Assignment`.

Redfish API functionality can be tested using the following endpoints in a web browser:

- General Redfish information:
  `https://<BMC-IP>/redfish/v1`
- Power metrics:
  `https://<BMC-IP>/redfish/v1/Chassis/1/Power#\PowerControl`

3.5.1.2 Kernel Configuration

Kepler requires kernel-level access for `eBPF` tracing, which involves calling the `perf_event_open` syscall. By default, this syscall is restricted. To enable Kepler's tracing functionality, an Ansible role adjusts the kernel parameter `perf_event_paranoid` via `sysctl` without requiring a reboot.

The restriction level can be verified by reading `/proc/sys/kernel/perf_event_paranoid`. For this project, all restrictions were removed by setting the value to `-1`.

3.5.2 Kepler Deployment with Ansible and Helm

Kepler was deployed using the Kepler Helm chart[**Kepler_helm_chart**] from the `sustainable-computing-io/Kepler-helm-chart` repository, with Ansible automating configuration and deployment. Deployment parameters were centralized in a Jinja template, rendered locally, and copied to the control-plane node before installation.

Key configurations in the Helm values file include:

- **Enabled metrics:** Various metric sources were activated for detailed energy monitoring.
- **Service port:** The Kepler service port was defined to allow Prometheus to scrape metrics.
- **Service interval:** The Kepler service interval was set to 10 seconds.
- **Redfish metrics:** Redfish/IPMI metrics were enabled, and Redfish credentials were provided. These credentials match those used for the Lenovo XClarity Controller interface. Note that the BMC IP differs from the node's IP address.

3.5.3 Verifying Kepler Metrics

After deployment, it was essential to verify that Kepler correctly collected and exposed metrics. Verification involved the following steps:

3.5.3.1 Prometheus Scraping

After deployment, successful Prometheus scraping of the Kepler endpoints was verified via the Prometheus web interface.

3.5.3.2 Metric Availability

All Kepler metrics were inspected individually in the Prometheus web interface to ensure that non-zero values were being reported. Each metric presented a single data point, offering additional confirmation that the corresponding data source was being monitored correctly.

3.5.3.3 Kepler Logs

The Kepler logs were reviewed to examine which data sources were successfully utilized:

LISTING 3.1: `kepler.log`

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:103] Kepler running on version: v0.7.12-dirty
1 config.go:293] using gCgroup ID in the BPF program: true
1 config.go:295] kernel version: 5.15
1 config.go:322] The Idle power will be exposed. Are you running on Baremetal or using single VM per node?
1 power.go:59] use sysfs to obtain power
1 node_cred.go:46] use csv file to obtain node credential
1 power.go:79] using redfish to obtain power
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:84] Number of CPUs: 6
1 watcher.go:83] Using in cluster k8s config
1 reflector.go:351] Caches populated for *v1.Pod from pkg/kubernetes/watcher.go:211
1 watcher.go:229] k8s APIserver watcher was started
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_TOTAL Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms]
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_COMPONENTS Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms bpf_cpu_time_ms bpf_cpu_time_ms   gpu_compute_util]
1 node_component_energy.go:62] Skipping creation of Node Component Power Model since the system collection is
    supported
1 prometheus_collector.go:90] Registered Process Prometheus metrics
1 prometheus_collector.go:95] Registered Container Prometheus metrics
1 prometheus_collector.go:100] Registered VM Prometheus metrics
1 prometheus_collector.go:104] Registered Node Prometheus metrics
1 exporter.go:194] starting to listen on 0.0.0.0:9102
1 exporter.go:208] Started Kepler in 2.2651724s
```

3.5.3.4 Power Metrics from ACPI / IPMI

When both ACPI and IPMI were enabled for platform power measurement, Kepler preferred IPMI as its primary data source. In this case, Kepler used IPMI for overall platform energy, while relying on ACPI to derive lower-level component estimates. This behavior is expected, as IPMI typically provides more complete platform-level information. In the absence of IPMI data, Kepler automatically falls back to ACPI as the sole power source.

3.5.3.5 Redfish Issues

Kepler occasionally failed to handle individual Redfish data values correctly. These incidents were sporadic and varied across different metrics. The underlying cause could not be resolved within the scope of this thesis. The following log entry illustrates such an error:

```
Failed to get power: json: cannot unmarshal number 3.07 into Go struct
    field Voltages.Voltages.ReadingVolts of type int
```

3.5.3.6 Error Message `cpu0/online`

The following error message is noteworthy:

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/
    online: no such file or directory
```

This warning occurs because the Intel Xeon processor used in this project does not support core offlining, the dynamic disabling of individual CPU cores at runtime. While core offlining is an interesting feature for energy-efficiency analysis, this limitation can be accepted as a hardware constraint of the project.

Chapter 4

Test Procedure

This chapter describes the test procedure used to verify Kepler-produced metrics. The verification process involves executing dynamic workloads on the Kubernetes cluster and analyzing the correlation between workload intensity and the metrics reported by Kepler. For better intuitive understanding, all Joule-based metrics are converted to Watts, and all operations-based metrics are converted to IOPS.

The collected data is visualized through diagrams to support interpretation. However, this thesis does not provide a detailed energy-efficiency analysis. Instead, the goal is to verify whether Kepler metrics reliably correlate with workload fluctuations, thereby confirming its suitability for a more in-depth energy-efficiency study in future work.

4.1 Test Setup

All test workloads were created using Ansible within a dedicated Kubernetes namespace, referred to as the *testing-namespace*. While the cluster was designed to be largely hardware-independent, the test setup requires manual adjustments when deployed on different hardware. Specifically, CPU and memory allocations for test pods must be reviewed, and the storage disk used for Disk I/O experiments must be empty and correctly identified.

4.1.1 Benchmarking Pod

A dedicated Ubuntu-based *benchmarking pod* was provisioned (using Ansible) to serve as the central test agent for all experiments. This pod enabled fully self-contained testing inside the cluster, without any dependency on external machines. The benchmarking pod was configured with a complete `kubectl` setup, an OpenSSH client, and essential tools such as `wget`, `curl`, `vim`, and `git`.

4.1.2 Testing Pods

Test workloads were deployed as DaemonSets to ensure that every node in the cluster hosted the required test pods. Depending on the experiment, different resource allocations were used:

- **CPU stress testing:** A test pod and a background load pod were deployed, each with 2.5 vCPU and 1 GB memory.

- **Memory stress testing:** A test pod and a background load pod were deployed, each with 150m vCPU and 25 GB memory.
- **Network I/O and Disk I/O testing:** A single test pod was deployed with 2.5 vCPU and 20 GB memory.

Resources were allocated to ensure an even split between CPU *testing* and *background load* pods, and a similar balance for memory-intensive workloads. A resource margin was maintained to prevent system instability. Benchmarking tools were installed on each pod, including `stress-ng`[118] for CPU and memory stress tests, `fio`[119] for Disk I/O testing, and `iperf3`[120] for network performance measurements.

4.1.3 Disk Formatting and Mounting

For Disk I/O experiments, an unused HDD on each worker node was partitioned, formatted, and mounted using Ansible. Persistence across reboots was ensured through an `/etc/fstab` entry.

4.2 Test Procedure

Since energy consumption is not calculated beyond the node level, all tests were conducted on a single worker node. The test pod (either high-CPU or high-memory) generated workloads at predefined levels of 10%, 30%, 50%, 70%, and 90% for a fixed duration of 30 minutes per workload level.

For CPU and memory testing, each experiment was run under two cluster conditions:

- **Idle cluster:** No background load.
- **Busy cluster:** Background pods at 90% utilization.

Because disk and network usage are not restricted by default in Kubernetes, this distinction was not applied for Disk I/O and Network I/O tests.

4.2.1 CPU Stress Test

CPU-intensive workloads were generated using `stress-ng`, with a CPU worker initiated on each available core. This test was performed under both idle and busy cluster conditions.

4.2.2 Memory Stress Test

Memory-intensive workloads were generated using `stress-ng`, where a virtual memory worker allocated all available memory. As with the CPU tests, experiments were conducted under idle and busy cluster conditions.

4.2.3 Disk I/O Stress Test

Disk performance was evaluated using `fio`. The test consisted of two phases: first, the maximum achievable IOPS were measured using random read operations on the

mounted HDD. Next, controlled read operations were issued at predefined percentages of the measured maximum. Random reads and direct I/O were used exclusively to eliminate caching effects.

4.2.4 Network I/O Stress Test

Network performance was evaluated using `iperf3`. First, the maximum bandwidth between pods on different nodes was measured. Then, controlled tests were executed at various percentages of the maximum bandwidth. To avoid server-side measurement overhead, only client-side results were analyzed.

4.3 Data Analysis

Data collected from each experiment was analyzed in two phases using Python.

4.3.1 Data Querying

Prometheus was queried to extract Kepler metrics corresponding to each experiment's duration. The retrieved data was stored as CSV files. The analysis relied on the Python libraries `pandas`, `requests`, and `datetime` for data querying and processing.

4.3.2 Diagrams

Visualization of Kepler metric data was performed using `matplotlib`. Each diagram included:

- X-axis: Time
- Primary Y-axis: Kepler metric values (Watts or operations per second)
- Secondary Y-axis: Test workload percentage
- A moving-average overlay to improve readability

By correlating workload levels with Kepler metrics, the structured analysis validated Kepler's suitability for future energy-efficiency studies.

Chapter 5

Test Results

This chapter presents the results of the test procedures conducted to analyze Kepler-produced metrics. Each section corresponds to a specific resource type with further division into container-level and node-level metrics. The results are discussed alongside their respective figures, which illustrate Kepler-deduced energy-consumption and performance trends.

It is important to note that all Kepler metrics exhibit strong oscillations. A closer analysis shows that these oscillations follow a highly regular pattern, suggesting an issue with either Kepler's metric publication intervals or the Prometheus scraping intervals. The data has been analyzed as-is, with moving averages added to improve readability. The implications of this irregularity will be discussed further in Chapter 6.

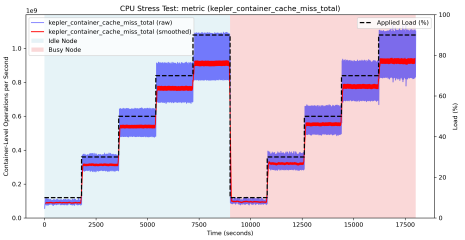
For clarity and to avoid confusion, three Kepler metric concepts are reiterated before discussing the results:

- **Package metrics:** Metrics representing the entire CPU package, including all cores and uncore components.
- **Platform metrics:** Metrics representing the entire node.
- ***Other* metrics:** Metrics capturing platform components other than the CPU package and DRAM.

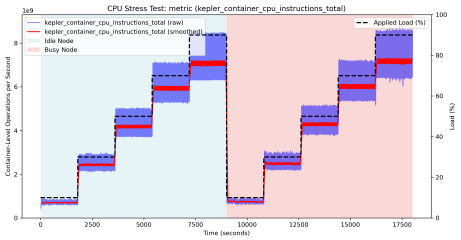
5.1 CPU Stress Test Results

5.1.1 Container-Level Metrics During a CPU Stress Test

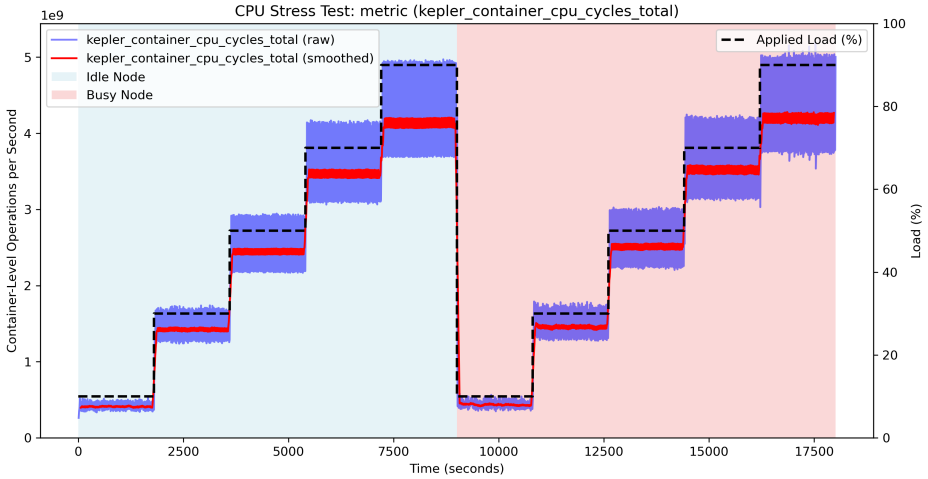
A set of figures illustrating cache misses, CPU cycles, and CPU instructions during testing is provided in Figures 5.7a to 5.1c.



(A) Cache misses



(B) CPU instructions



(C) CPU cycles

FIGURE 5.1: Kepler container-level CPU metrics during a CPU stress test

These figures illustrate cache misses, CPU cycles, and CPU instructions for the test container during execution. The diagrams show uniform trends, as the three metrics directly reflect the workload generated by `stress-ng`, which is designed to be consistent and stable. The strong correlation between the applied workload and the metrics (cache misses, CPU cycles, and CPU instructions) confirms the correct execution of the test.

Because the CPU workload running on the rest of the cluster should not affect the test container's workload, the metric values under idle and busy cluster conditions are expected to be identical. This is indeed the case, confirming that the testing procedure was executed correctly.

A figure illustrating Kepler's deduced Package energy consumption is provided in Figure 5.2.

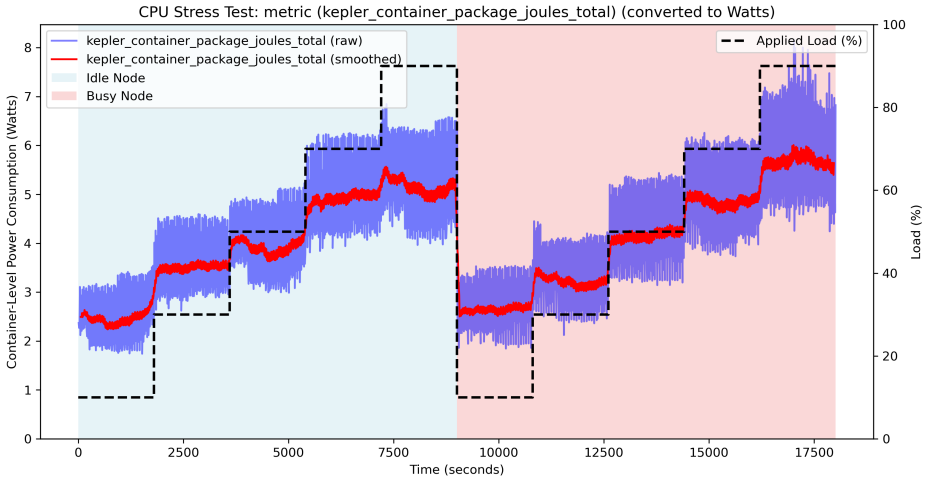
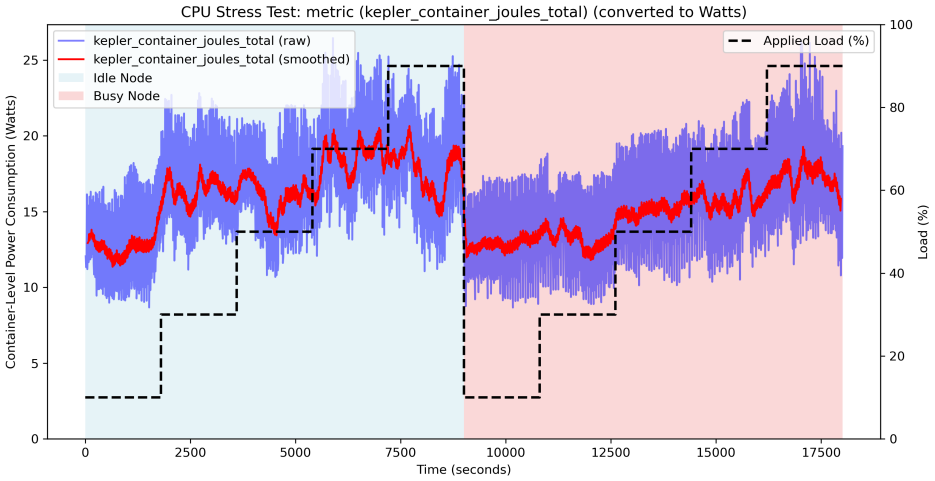


FIGURE 5.2: Kepler container-level Package energy consumption

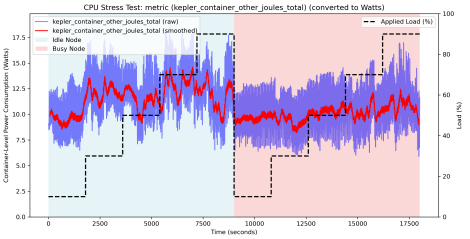
The figure shows a clear upward trend in Package energy consumption, with distinct steps that correspond to the expected workload increases. A strong correlation is observed between Kepler's reported Package energy consumption and the test workload. However, the relationship between energy consumption and workload is non-linear: while a 10% workload averages around 2.5 W, a 90% workload results in only about twice the energy consumption, despite the workload increasing by a factor of nine.

Furthermore, Kepler's Package energy measurements remain consistent regardless of whether the node is idle or busy, showing no statistically significant difference.

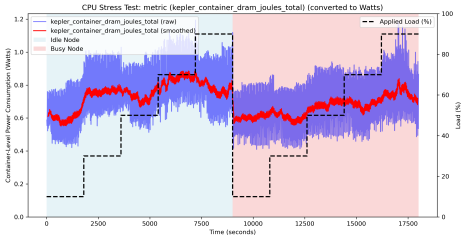
A set of figures illustrating total container energy consumption, DRAM energy consumption, and *Other* energy consumption components is provided in Figures 5.3a to 5.3c.



(A) Total container-level energy consumption



(B) *Other* energy consumption



(C) DRAM energy consumption

FIGURE 5.3: Kepler container-level energy consumption during a CPU stress test

The figures for container energy consumption, DRAM energy consumption, and *Other* components (representing host components excluding CPU and DRAM) show a less direct correlation with workload than the Package energy.

The main observations are:

- In Figure 5.3a, a slight upward trend is visible: total container energy consumption increases by roughly 5 W. This change mirrors the increase in Package energy consumption in Figure 5.2, where an approximate 5 W increase can also be observed.

- In Figure 5.3b, which shows *Other* (non-CPU/DRAM) container energy consumption, no clear trend can be identified. This is expected, since only the CPU was explicitly stressed. However, the overall magnitude of *Other* energy consumption is surprisingly high, reaching roughly twice the CPU Package energy.

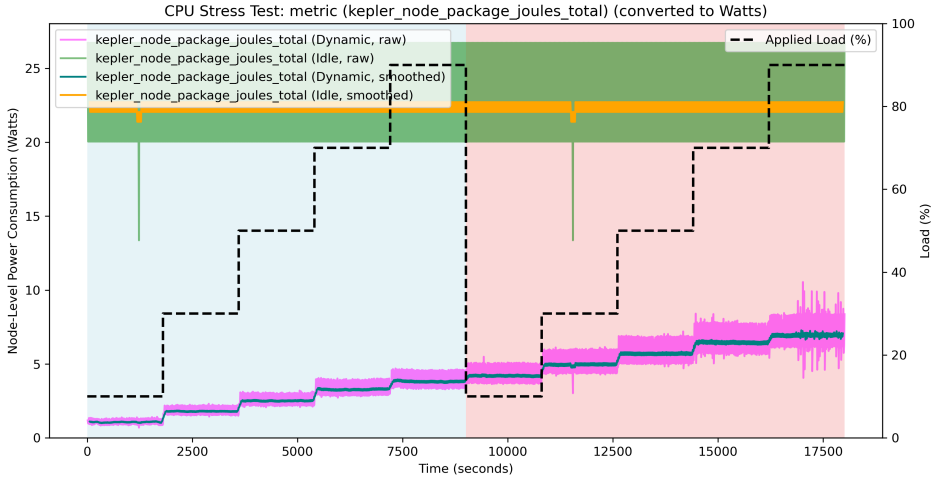
- The measured DRAM energy consumption is largely unaffected by CPU stress, as expected. With values between 0.5 and 1 W, DRAM energy remains comparatively low.

- During the second part of the experiment (the busy-node condition), all metrics appear slightly smoother, but they are neither significantly higher nor

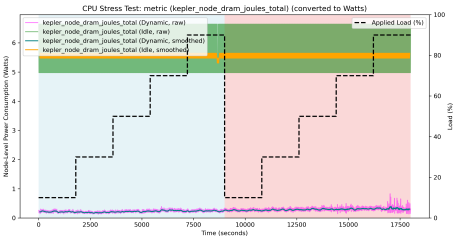
lower compared to the idle-node experiment.

5.1.2 Node-Level Metrics During a CPU Stress Test

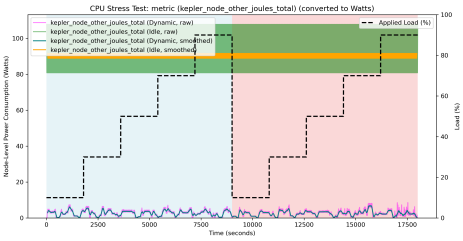
Figures illustrating node-level package, DRAM, and *Other* energy consumption are provided in Figures 5.4a to 5.4c. For node-level energy consumption, Kepler distinguishes between idle and dynamic power.



(A) Node Package energy consumption



(B) Node DRAM energy consumption



(C) Node *Other* energy consumption

FIGURE 5.4: Kepler node-level energy consumption during a CPU stress test

The following observations can be made:

- The most striking observation is the relatively high idle energy consumption of the node, which is visible in all figures. While Figure 5.4a shows increasing dynamic Package energy due to the CPU stress test, idle energy still far exceeds the dynamic component.
- The dynamic *Other* energy consumption shown in Figure 5.4c appears to be largely independent of the CPU stress load. This further supports the conclusion that *Other* system components contribute significantly to overall platform energy consumption but are generally unaffected by CPU workload.

5.1.3 Overall Conclusions

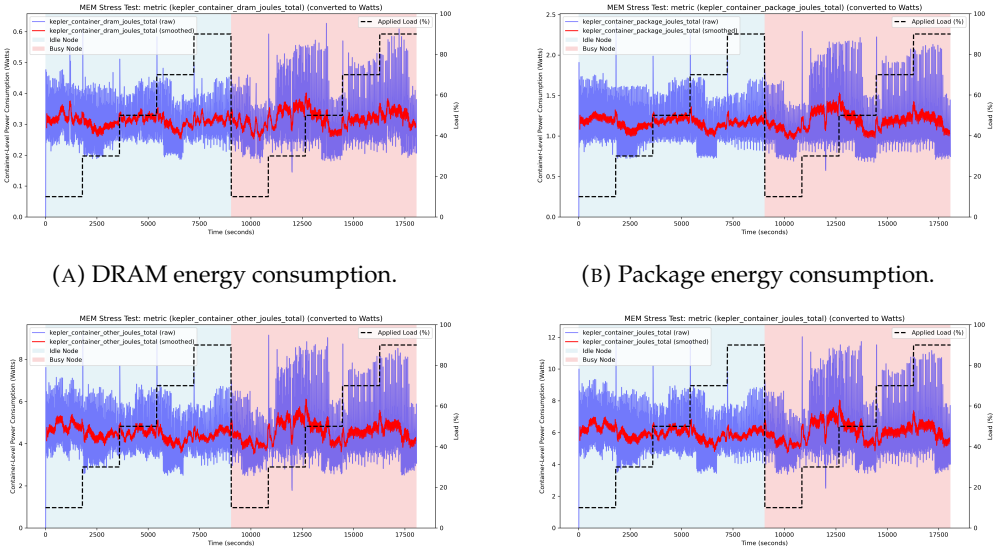
The CPU stress test results demonstrate that Kepler captures workload-dependent variations in energy consumption with reasonable accuracy. Key takeaways from the analysis include:

- Kepler's CPU Package energy measurements correlate with workload intensity, although the relationship is clearly non-linear.
- High idle energy consumption at the node level suggests that a substantial share of total energy use is independent of CPU workload.
- The *Other* component's energy consumption remains largely static with respect to workload, indicating that these components primarily contribute to baseline (idle) energy consumption rather than dynamic variations.

5.2 Memory Stress Test Results

5.2.1 Container-Level Metrics During a Memory Stress Test

Figures 5.5a to 5.5d show the container-level energy consumption metrics published by Kepler during the memory stress test.



(A) DRAM energy consumption.

(B) Package energy consumption.

(C) *Other* energy consumption.

(D) Total container energy consumption.

FIGURE 5.5: Container-level energy consumption during a memory stress test.

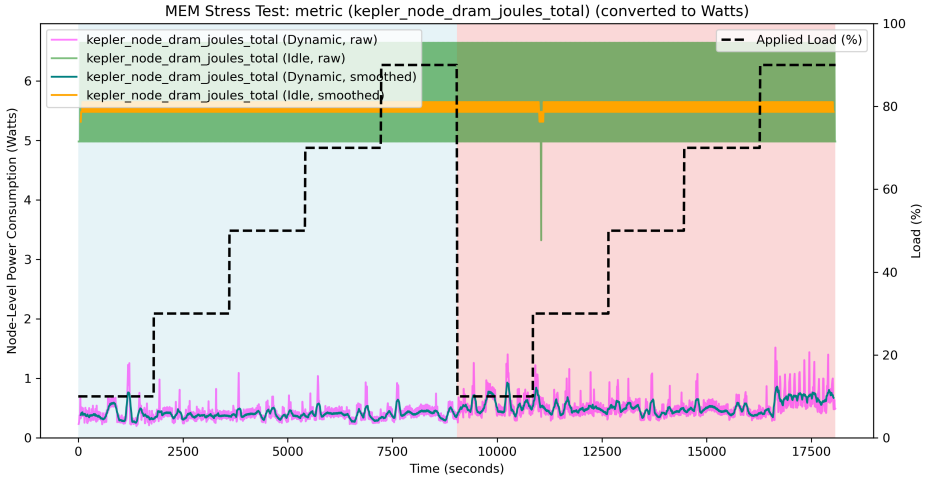
The following observations can be made:

- None of the four published energy metrics correlate with the applied memory stress load. There is also no significant difference between executing the stress test on an idle versus a busy node. None of the energy metrics indicate that a memory stress test is being performed.

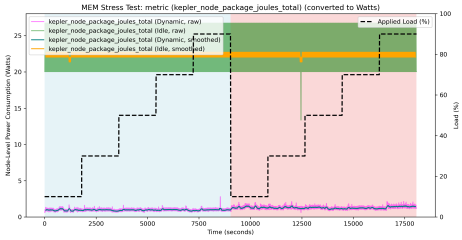
- Figure 5.5b shows an average container-level DRAM energy consumption of approximately 0.3 W. This is considerably lower than the 0.7 W measured during the CPU stress test (Figure 5.3c), which also exhibits a clear upward trend during higher CPU workloads.

5.2.2 Node-Level Metrics During a Memory Stress Test

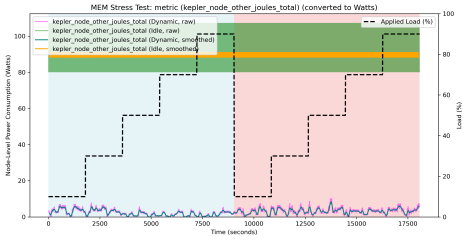
Figures 5.6a to 5.6c show the node-level idle and dynamic energy consumption metrics published by Kepler during the memory stress test.



(A) DRAM energy consumption.



(B) Package energy consumption.



(C) *Other* energy consumption.

FIGURE 5.6: Node-level energy consumption during a memory stress test.

The node-level metrics collected during the memory stress test present a similar picture to the container-level metrics. The following observations can be made:

- No node-level energy consumption metric correlates with the memory stress applied during the test.
- All node-level energy consumption metrics exhibit significantly higher idle energy consumption than dynamic energy consumption.

5.2.3 Overall Conclusions

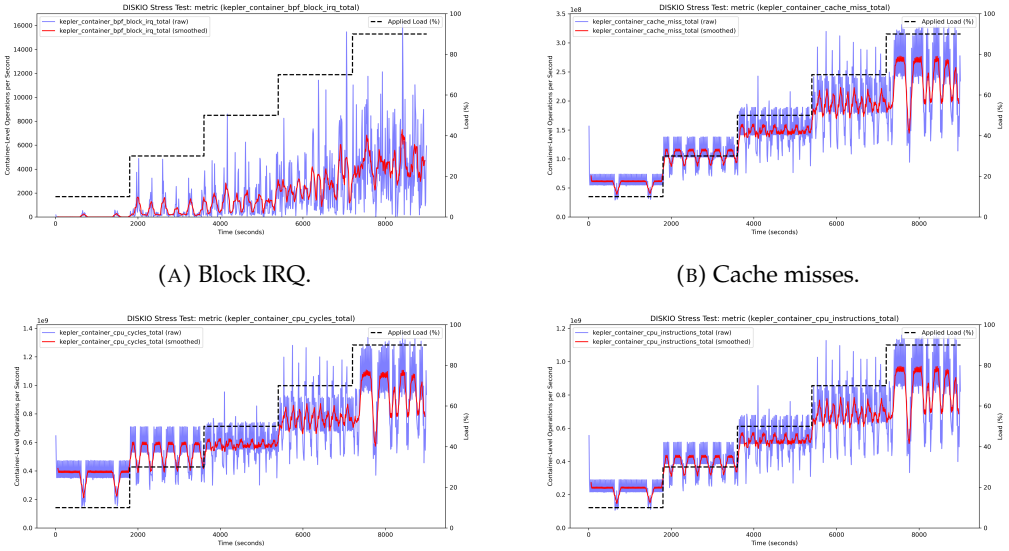
The following key takeaways can be derived from the memory stress test results:

- The memory stress test does not indicate any capability of Kepler to reliably track memory energy consumption. None of the metrics respond to the various stimuli of the test scenario.
- However, the container-level DRAM metric appears to be affected by CPU stress, as observed in the CPU stress test.

5.3 Disk I/O Stress Test Results

5.3.1 Container-Level Metrics During a Disk I/O Stress Test

Figures 5.7a to 5.7d show the CPU metrics published by Kepler during the Disk I/O stress experiment.



(A) Block IRQ.

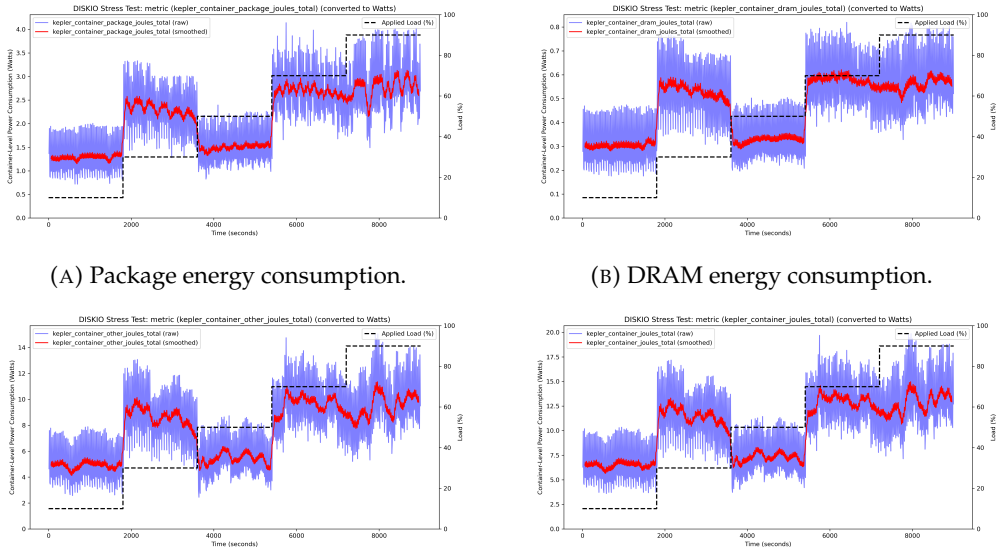
(B) Cache misses.

(C) CPU instructions.

(D) CPU cycles.

FIGURE 5.7: Container-level CPU metrics during a Disk I/O stress test.

All four figures behave as expected and validate the general test procedure. Notably, the operations per second for cache misses, CPU instructions, and CPU cycles do not scale linearly with the workload. The relative difference between the low-workload and high-workload tests is approximately 330% (cache misses), 250% (CPU instructions), and 350% (CPU cycles), all significantly lower than the 900% relative difference in the applied load.



(A) Package energy consumption.

(B) DRAM energy consumption.

(C) *Other* energy consumption.

(D) Total container energy consumption.

FIGURE 5.8: Container-level energy consumption during a Disk I/O stress test.

Figures 5.8a to 5.8d show the Kepler-reported energy metrics for the package, DRAM, *Other* components, and total container energy consumption. Several observations can be made:

- All four metrics exhibit a bimodal distribution, oscillating between distinct *low* and *high* states. While varying HDD rotation speeds might explain the curves observed for *Other* components and total container energy consumption, this does not account for the behaviour observed in package and DRAM energy consumption.
- The relative energy consumption across all four metrics appears nearly identical regardless of the component measured. This is particularly evident for the *Other* components and total container energy consumption, which appear identical aside from a scaling factor of approximately 1.5.
- All figures show strong temporal correlation with the test intervals, indicating that Kepler does detect changes in disk load.

To further analyze the metrics during a Disk I/O stress test, the experiment was repeated; the container-level Package energy consumption is shown in Figure 5.9.

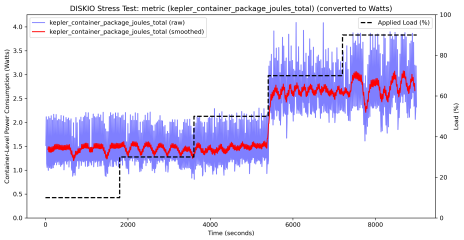


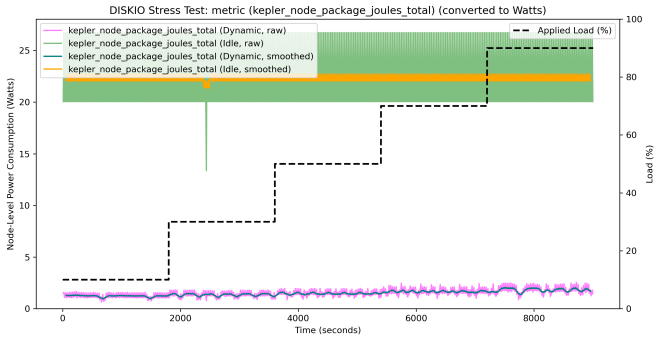
FIGURE 5.9: Container-level Package energy consumption during a second experiment.

The following observations were consistent with the first experiment:

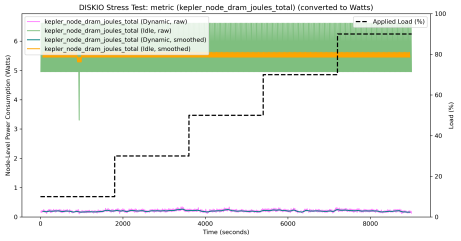
- All joule-based metrics still displayed a bimodal distribution.
- All joule-based metrics appear identical aside from differences in scale.
- All joule-based metrics show an observable upward trend.

5.3.2 Node-Level Metrics During a Disk I/O Stress Test

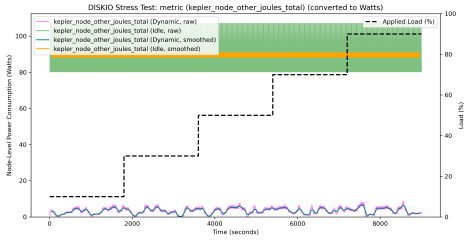
Figures 5.10a to 5.10c show the node-level energy consumption for the package, DRAM, and *Other* components, split into idle and dynamic energy consumption.



(A) Node Package energy consumption.



(B) Node DRAM energy consumption.



(C) Node *Other* energy consumption.

FIGURE 5.10: Kepler node-level energy consumption during a Disk I/O stress test.

The following observations can be made regarding node-level energy consumption:

- Across all components, the difference between idle and dynamic power is pronounced. The majority of node energy consumption occurs as idle power.
- The node DRAM and *Other* components do not show an increasing trend in dynamic energy consumption. In contrast, Figure 5.10a shows an upward trend of approximately 50% in dynamic Package energy consumption, although this remains significantly overshadowed by idle energy consumption.

5.3.3 Overall Conclusions

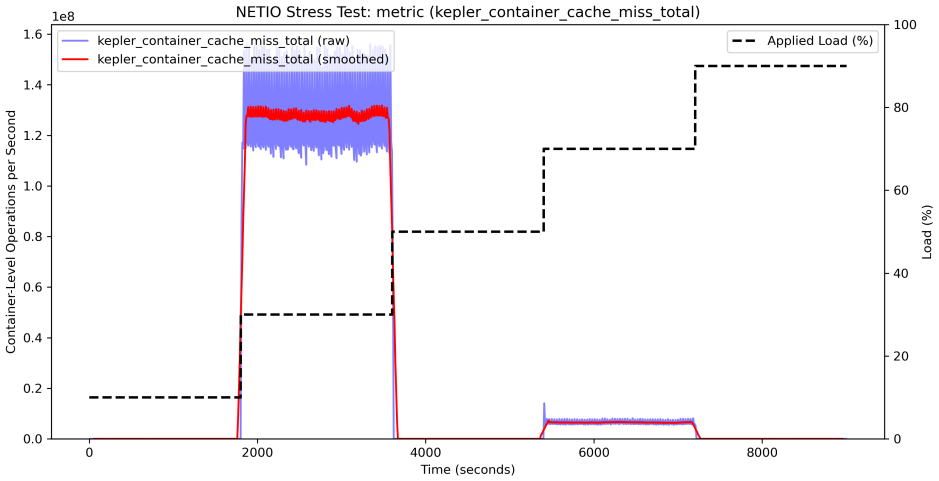
The following overall conclusions can be drawn regarding Kepler metrics during a Disk I/O test:

- Kepler provides plausible metrics for Block IRQ, cache misses, CPU instructions, and CPU cycles during a Disk I/O stress test.
- The accuracy of Kepler's container-level joule-based metrics is questionable. While they exhibit an intriguing bimodal distribution, they do not necessarily align with the Disk I/O workload.
- Kepler's container-level joule-based metrics reliably *detect* changes in Disk I/O workload, but their predictability remains uncertain.
- Node-level Kepler metrics do not suggest any significant impact of Disk I/O workload on DRAM or *Other* components.
- The fact that the variations in energy consumption shown in container-level metrics cannot clearly be traced in either idle or dynamic node energy consumption raises further questions.
- The hypothesis that Disk I/O stress does not significantly contribute to node power cannot be rejected or proven. Further research is necessary.

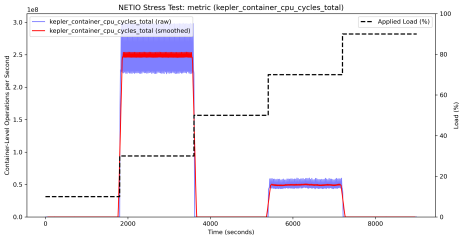
5.4 Network I/O Stress Test Results

5.4.1 Container-Level Metrics During a Network I/O Stress Test

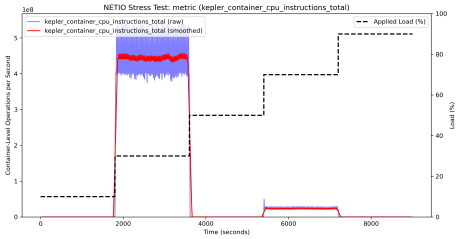
Figures 5.11a to 5.11c show the CPU metrics published by Kepler during the Network I/O stress experiment.



(A) Cache misses.



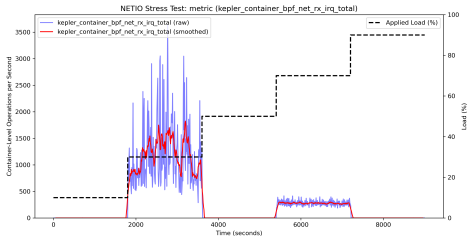
(B) CPU instructions.



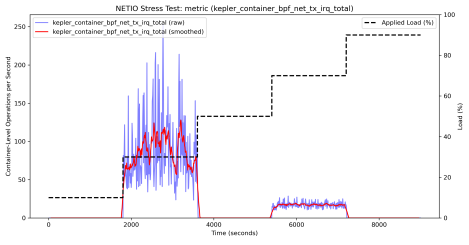
(C) CPU cycles.

FIGURE 5.11: Container-level CPU metrics during a Network I/O stress test.

The results do not provide a clear explanation for the applied test stress. While the metrics correlate in time with the applied Network I/O load, their values do not clearly correspond to the applied stress. The Kepler metrics for RX IRQ and TX IRQ in Figures 5.12a and 5.12b present a similar trend, though with significantly higher variance.

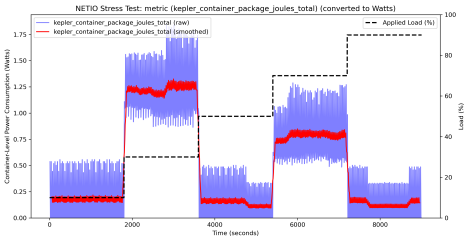


(A) RX IRQ.

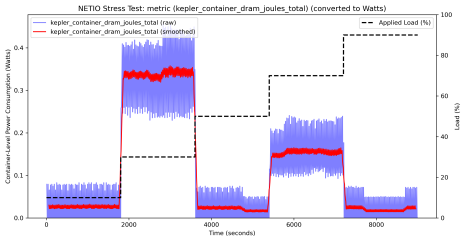


(B) TX IRQ.

FIGURE 5.12: Container-level IRQ metrics during a Network I/O stress test.

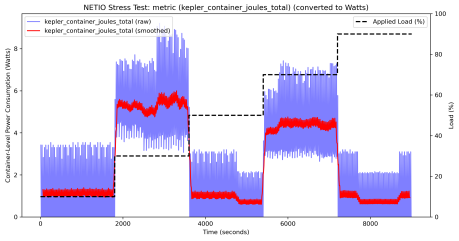


(A) Package energy consumption.



(B) DRAM energy consumption.

(C) *Other* energy consumption.



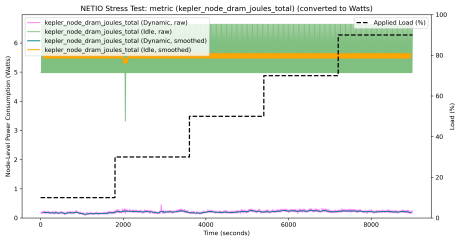
(D) Total container energy consumption.

FIGURE 5.13: Container-level energy consumption during a Network I/O stress test.

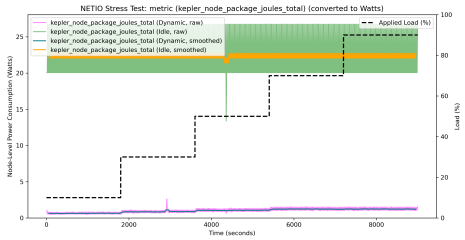
Figures 5.13a to 5.13d display Kepler metrics for package, DRAM, *Other*, and overall container-level energy consumption. The patterns remain consistent: the metrics indicate transitions between different workloads, but the values (while nearly constant for each experiment) appear unrelated to the Network I/O activity.

5.4.2 Node-Level Metrics During a Network I/O Stress Test

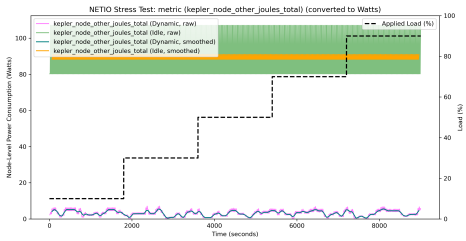
Figures 5.14a to 5.14c show the node-level idle and dynamic energy consumption metrics published by Kepler during the Network I/O stress test.



(A) DRAM energy consumption.



(B) Package energy consumption.



(C) *Other* energy consumption.

FIGURE 5.14: Node-level energy consumption during a Network I/O stress test.

Much like the previous Disk I/O test, the node-level energy consumption metrics do not show any clear indication of the applied network load. The only notable observation is the stepwise increase in dynamic Package energy consumption, which is likely related to the actual generation of network traffic.

5.4.3 Overall Conclusions

Unfortunately, the test does not indicate a clear relationship between the applied Network I/O load and the Kepler metrics. Similar to the Disk I/O test, the following conclusions can be drawn:

- The Kepler metrics are affected by the Network I/O stress test, as evidenced by their temporal correlation with the applied workload.
- However, the metric values do not logically correlate with the actual network traffic being generated.

This test was repeated multiple times with consistent results: the workload continued to correlate in time with the test steps, but the metric values plateaued at seemingly arbitrary levels for the duration of each load phase.

Chapter 6

Discussion

6.1 Conclusion and Evaluation

6.1.1 Evaluation of Cluster Setup

The cluster setup was successful and proved viable for further Kubernetes testing. While implementing the entire setup in an automated manner required substantial additional effort, the resulting deployment functioned reliably throughout the project. The ability to tear down and re-deploy the cluster at any desired depth (ranging from Kubernetes deployments and configurations to a complete reinstallation) was invaluable during experimentation. This ensured that any misconfigurations introduced during implementation could be fully removed, preventing residual effects from failed installations or incorrect configurations.

The automated setup was intentionally designed for portability across different hardware platforms. Although this capability was not tested within the project, it significantly increases the setup's reusability. Future work could adapt and deploy the setup with minimal adjustments, enabling researchers and engineers to quickly establish experimental Kubernetes clusters in diverse environments.

One of the primary constraints of this project was the decision not to implement a high-availability (HA) cluster. Given the project's focus on energy efficiency measurements rather than production-grade reliability, this was an appropriate trade-off. However, HA clusters are standard in large-scale production systems. Energy efficiency research in such environments could offer additional insights into how energy optimization strategies behave under real-world workload distributions.

Finally, graphical tools such as Rancher proved extremely useful during configuration and experimentation. Rancher's centralized UI provided a clear overview of cluster state, simplifying Kubernetes troubleshooting and management. While the project emphasized automation, Rancher complemented the setup by offering real-time monitoring and fast identification of configuration issues.

6.1.2 Evaluation of Monitoring Setup

The monitoring setup proved effective for energy consumption testing. Prometheus, the de facto standard for Kubernetes monitoring, ensured compatibility with a wide range of tools and provided access to extensive community support, documentation, and integrations. This was particularly advantageous for Kepler, which integrates

directly with Prometheus, resulting in a smooth deployment and data collection process.

A notable limitation of Prometheus is the overhead and granularity constraints introduced by periodic metric scraping. It is well suited for system monitoring at multi-second or minute-level intervals, but this limits its usefulness in high-resolution energy consumption analysis. While Kepler collects large amounts of data through eBPF and RAPL, the need to reduce data density for Prometheus-friendly exports leads to a loss of granularity. Thus, Prometheus excels at long-term trend analysis but is suboptimal for capturing rapid fluctuations in power consumption.

The use of NFS-based persistent storage on the Kubernetes control node proved reliable. Throughout the project (including multiple cluster redeployments) no monitoring data was lost. The NFS setup ensured persistent storage for Prometheus and Grafana, maintaining historical data even when the cluster was reset.

Although this monitoring setup is well suited for experimentation and research, deploying it in a production environment would require significant enhancements to ensure data integrity, resilience, and security. For instance, Prometheus data retention and storage configurations would need reinforcement, authentication and authorization mechanisms would require strengthening, and redundancy would be necessary to prevent data loss in the event of node failure.

6.1.3 Evaluation of Kepler

Kepler was integrated successfully using the provided Helm chart, although several configuration adjustments were required to ensure compatibility with the existing infrastructure. The project documentation, while helpful, has not yet reached full maturity, resulting in occasional challenges during setup and troubleshooting. Despite these hurdles, Kepler's underlying concept is well-founded, using suitable data sources to estimate energy consumption at both the container and node levels.

6.1.3.1 General Observations

- All metrics exhibited pronounced and consistent oscillations. Since the metrics are computed from simple counters, this indicates a synchronization issue, possibly between Kepler's metric publication intervals and Prometheus' scraping intervals. While this does not undermine the credibility of the data, resolving it would greatly improve the usability of the metrics.

- **CPU Energy Metrics:** Kepler successfully captures workload-dependent energy variations, showing a strong correlation between CPU stress levels and estimated power consumption. Since CPU load is the dominant factor in overall server energy consumption, this is a significant strength of Kepler.

- **Memory Energy Metrics:** Unlike CPU metrics, memory energy estimates did not show a clear correlation with applied workload. However, this is not necessarily a flaw in Kepler's methodology. Memory typically accounts for a small proportion of overall server energy consumption and varies far less than CPU power. Thus, the lack of a pronounced correlation is not unexpected. However, the experiment did not verify Kepler's ability to measure memory energy consumption.

- **Disk I/O and Network I/O Metrics:** Kepler's energy estimates for disk and network activity did not follow the expected trends. Although the metrics responded to workload transitions in a time-synchronized manner, the exact correlation remained unclear. In particular, disk and network energy consumption values were not proportional to the applied stress levels. This anomaly warrants further investigation, especially considering that HDD power consumption is known to depend only weakly on workload intensity.

6.1.4 Credible Takeaways from the Test Results

Kepler's package metrics appear reliable and provide plausible results. Although the absolute values were not validated within the scope of this project, the reported metrics closely matched the CPU workload trends observed during testing.

Energy consumption does not scale linearly with workload. For package power, increasing the workload from 10% to 90% resulted in only an approximately 250% increase in Kepler-estimated energy consumption. This aligns with the well-known observation that servers operate most efficiently at higher utilization levels.

Idle energy consumption was consistently estimated to be much higher than dynamic energy consumption. While older servers are known to exhibit high idle power usage, Kepler's estimate that idle energy consumption reaches roughly 90% on a CPU-stressed server seems unlikely and requires further investigation.

These findings suggest that Kepler's CPU energy estimation is robust, while inconsistencies in memory, disk, and network energy metrics highlight areas requiring additional validation. This naturally motivates future research into improving Kepler's measurement accuracy.

6.2 Future Work

6.2.1 Detailed Analysis of Kepler

While Kepler demonstrates strong capabilities in CPU power estimation, the inconsistencies in its memory, disk, and network metrics indicate areas requiring further research. A limitation of this study was its reliance on a single server platform. A meaningful next step would be to compare Kepler's energy estimates across different hardware configurations to evaluate generalizability.

6.2.2 Kepler Metrics Verification Through Elaborate Tests, Possibly Using Measuring Hardware

Kepler's metrics should be validated through more extensive and diverse test scenarios, using different stress tools and workloads. In some cases, integrating physical power measurement hardware could offer an additional validation layer. With deeper insight into Kepler's internal mechanisms, it may become possible to verify the reported metrics directly. Ultimately, ensuring that Kepler metrics accurately reflect energy consumption at both the node and container levels is essential for establishing their reliability.

6.2.3 Kubernetes Cluster Energy Efficiency Optimization

If Kepler metrics prove reliable for cluster-wide energy estimation, future research could investigate energy efficiency optimizations in Kubernetes environments. Potential areas of study include evaluating existing energy-saving techniques (e.g. carbon-aware schedulers), developing new optimization strategies, and analyzing the effects of different cluster configurations. For example, experiments could explore potential energy savings achieved by disabling high-availability features or dynamically powering servers on and off based on workload demand.

6.3 Final Conclusion

This project successfully established an experimental Kubernetes cluster with integrated energy monitoring. The results demonstrate that Kepler is a promising tool for CPU energy estimation, although further refinement is needed for other resource types. Going forward, improved metric validation, hardware comparisons, and research into cluster-wide optimization strategies will be essential to fully leverage Kepler for practical energy efficiency improvements.

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] Linux Foundation Energy and Performance Working Group. *Kepler: Kubernetes-based Power and Energy Estimation Framework*. Accessed: 2025-11-14. 2025. URL: https://github.com/sustainable-computing-io/kepler.

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

[11] Caspar Wackerle. *Tycho: an accuracy-first container-level energy consumption exporter for Kubernetes (based on Kepler v0.9)*. GitHub repository. 2025. URL: https://github.com/casparwackerle/tycho-energy.

[12] NVIDIA Corporation. *NVML API Reference Guide*. Function `nvmlDeviceGetPowerUsage`: retrieves GPU power usage in milliwatts. 2024. URL: https://docs.nvidia.com/deploy/pdf/NVML_API_Reference_Guide.pdf (visited on 11/13/2025).

[13] Yole Group. *Data Center Semiconductor Trends 2025: Artificial Intelligence Reshapes Compute and Memory Markets*. Press Release. 2025. URL: https://www.yolegroup.com/press-release/data-center-semiconductor-trends-2025-artificial-intelligence-reshapes-compute-and-memory-markets/.

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

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] 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).

[20] 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).

[21] 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).

[22] 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).

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

[24] 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).

[25] 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.

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

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

[28] 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).

[29] 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).

[30] 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).

[31] 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).

[32] 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.

[33] 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).

[34] 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).

[35] 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).

[36] 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).

[37] 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).

[38] 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).

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

[40] 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).

[41] 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.

[42] 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/.

[43] 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.

[44] 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).

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

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

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

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

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

[50] 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.

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

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

[53] 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).

[54] 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).

[55] 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).

[56] 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).

[57] 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).

[58] 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).

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

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

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

[62] 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).

[63] 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).

[64] 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).

[65] 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.

[66] 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).

[67] 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).

[68] 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).

[69] 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.

[70] 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.

[71] 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.

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

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

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

[75] 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).

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

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

[78] 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).

[79] 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).

[80] 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).

[81] 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.

[82] 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).

[83] 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).

[84] 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).

[85] 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).

[86] 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).

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

[88] 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.

[89] 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.

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

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

[92] 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.

[93] 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).

[94] 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.

[95] 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/.

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

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

[98] 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.

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

[100] 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/.

[101] 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.

[102] 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).

[103] 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).

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

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

[106] Inc. Meta Platforms. *Kepler v0.9.0 (pre-rewrite): Kubernetes-based power and energy estimation framework*. Accessed: 2025-04-28. 2023. URL: https://https://github.com/sustainable-computing-io/kepler/releases/tag/v0.9.0.

[107] 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).

[108] Sustainable Computing. *Kepler: Kubernetes Efficient Power Level Exporter Documentation (Deprecated)*. https://sustainable-computing.io/. Deprecated documentation, Accessed: June. 2025.

[109] 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).

[110] Marcelo Amaral et al. *Exploring Kepler's Potentials: Unveiling Cloud Application Power Consumption*. https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/. CNCF Blog. Oct. 2023. URL: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/.

[111] Global e-Sustainability Initiative (GeSI) and Carbon Trust. *ICT Guidance on GHG Protocol Product Life Cycle Accounting and Reporting Standard*. https://www.gesi.org/public-resources/ict-guidance-on-ghg-protocol-product-life-cycle-accounting-and-reporting-standard/. Accessed 2 Jul 2025. Nov. 2024.

[112] Lars Andringa. "Estimating energy consumption of Cloud-Native applications". PhD thesis. 2024.

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

[114] 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).

[115] 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).

[116] k3s-io. *k3s-ansible: Ansible playbook for deploying K3s clusters*. https://github.com/k3s-io/k3s-ansible. Accessed: 2025-01-05. 2025. URL: https://github.com/k3s-io/k3s-ansible.

[117] Prometheus Community. *Prometheus Helm Charts*. Accessed: 2025-01-05. 2025. URL: https://github.com/prometheus-community/helm-charts.

[118] Colin Ian King. *stress-ng: A Linux System Stressing Tool*. https://github.com/ColinIanKing/stress-ng. Accessed: 2025-01-27. 2023.

[119] Jens Axboe. *fio: Flexible I/O Tester*. https://fio.readthedocs.io/en/latest/. Accessed: 2025-01-27. 2023.

[120] The ESnet Project. *iperf3: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool*. https://iperf.fr/. Accessed: 2025-01-27. 2023.