Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

Master Thesis

Title

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: Title
Author: Caspar Wackerle
Date: January 31, 2026
Keywords: energy efficiency, cloud, kubernetes
Copyright: Zurich University of Applied Sciences

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

Chapter 1

Introduction and Context

[1]

1.1 System Environment for Development, Build and Debugging

This section documents the environment used to develop, build, and debug *Tycho*; detailed guides live in [2].

1.1.1 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.

1.1.2 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*).

1.1.2.1 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.

1.1.2.2 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.

1.1.2.3 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.

1.1.2.4 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.

1.1.3 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.

1.1.3.1 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.

1.1.3.2 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.

1.1.3.3 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.

1.1.4 Supporting Tools and Utilities

1.1.4.1 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 § **??**. Repository scripts map configuration keys to explicit flags for local runs, debug sessions, and ad hoc deploys.

1.1.4.2 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.

1.1.5 Relevance and Limitations

1.1.5.1 Scope and contribution

The development, build, and debugging environment described in § 1.1.2 and § 1.1.3 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 § 1.1 provide the operational detail that is intentionally omitted from the main text.

1.1.5.2 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 § 1.1.

1.2 ebpf-collector-Based CPU Time Attribution

1.2.1 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.

1.2.2 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.

Data flows linearly from tracepoints to per-CPU maps and onward to the userspace collector, forming a continuous and low-overhead measurement path. This architecture transforms Kepler's periodic snapshot model into a streaming telemetry layer that maintains temporal consistency and provides the necessary basis for accurate, time-aligned energy attribution.

1.2.3 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.

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 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 sampling intervals.

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, userspace reads these bins, derives total CPU activity for the window, and resets them to zero for the next interval.

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.

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 then to the userspace collector through batched lookups, forming a deterministic, lock-free telemetry path from kernel to analysis.

1.2.4 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.

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.

1.2.5 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 process contributes its accumulated runtime and performance-counter deltas, while system activity and idle phases are derived from the per-CPU bins. The sum of these components represents the total active time observed by the processor, matching the energy sample boundaries defined by the timing engine. This 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.

1.2.6 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. 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 sampling windows. Together, these measures yield a stable and version-tolerant tracing layer that can run indefinitely without producing inconsistent or overlapping samples.

1.2.7 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 sampling windows to remain stable. Within shorter intervals, 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 be evicted from the LRU map before collection, slightly undercounting transient workloads.

1.3 GPU Collector Integration

The GPU collector extends Tycho's measurement framework to include accelerator energy and utilization data. Building on Kepler's existing device abstraction, it reuses and refines NVIDIA-specific collection paths while integrating them into Tycho's modular timing and buffering architecture.

1.3.1 Overview and Objectives

The GPU collector extends Tycho's energy measurement framework to include accelerator telemetry. Its purpose is not to introduce new metrics, but to integrate existing GPU energy and utilization data into Tycho's synchronized collection cycle. By reusing and refining Kepler's accelerator interface, Tycho can obtain GPU-level power and activity data without duplicating existing logic.

The collector operates identically across hardware tiers. On systems equipped with NVIDIA's DCGM or NVML interfaces, it retrieves instantaneous power, utilization, and memory metrics, aligning them with Tycho's monotonic timebase for unified analysis with CPU and platform energy data.

1.3.2 Architecture and Backend Selection

Kepler's accelerator abstraction exposes a uniform device interface backed by multiple telemetry providers. Tycho reuses this structure to maintain compatibility and minimize maintenance effort. Two NVIDIA backends are supported: `DCGM` (Data Center GPU Manager) and `NVML` (NVIDIA Management Library).

DCGM is preferred when available, as it provides high-resolution telemetry, process-level utilization, and Multi-Instance GPU (MIG) awareness on enterprise hardware. NVML serves as a fallback for consumer-grade devices with limited instrumentation. This layered design ensures that Tycho can operate across development and production environments without configuration changes.

Through this abstraction, the GPU collector accesses metrics through a single interface. Backend selection, device enumeration, and capability handling are managed internally, allowing Tycho to treat GPU data as a consistent input source, regardless of the underlying driver.

1.3.3 Collected Metrics

The GPU collector retrieves instantaneous and cumulative telemetry from the active accelerator backend (`DCGM` or `NVML`). All values are sampled at fixed intervals and aligned to Tycho's monotonic timebase. Table 1.2 lists the available input metrics.

Metric	Unit	Description
Utilization metrics		
`SMUtilPct`	%	Percentage of active streaming multiprocessors (SMs), representing compute load.
`MemUtilPct`	%	GPU memory controller utilization.
`EncUtilPct`	%	Hardware video encoder utilization.
`DecUtilPct`	%	Hardware video decoder utilization.
Energy and thermal metrics		
`PowerMilliW`	mW	Instantaneous power draw per device.
`EnergyMicroJ`	µJ	Integrated energy derived from power samples over time.
`TempC`	°C	Current GPU temperature.
Memory and frequency metrics		
`MemUsedBytes`	bytes	Allocated frame-buffer memory.
`MemTotalBytes`	bytes	Total available frame-buffer memory.
`SMClockMHz`	MHz	Streaming multiprocessor clock frequency.
`MemClockMHz`	MHz	Memory clock frequency.

TABLE 1.2: Metrics collected by the GPU collector.

Together these metrics describe the GPU's operational state and power consumption. They provide the foundation for process-level energy attribution by combining instantaneous power with per-process utilization data retrieved from the same backend.

1.3.4 Integration and Data Flow

The GPU collector operates as an independent module within Tycho's collection framework. Initialization occurs during startup, where Kepler's accelerator registry detects available devices and activates either the DCGM or NVML backend. Once initialized, the collector periodically polls device metrics using Tycho's scheduling engine and stores each result in a synchronized ring buffer.

Each collected sample is timestamped through the system's monotonic clock to maintain temporal consistency with other subsystems such as RAPL and eBPF. This design allows GPU data to be directly correlated with CPU and platform measurements during post-processing. By aligning all sources under a shared timebase and buffer structure, Tycho guarantees consistent sampling intervals and deterministic integration across heterogeneous energy domains.

1.3.5 Robustness and Limitations

The collector was designed for stability across varying hardware and driver configurations. Additional validation and error handling were introduced to tolerate missing or partially initialized devices, ensuring safe operation even when GPUs are unavailable or the driver interface is incomplete. The NVML backend was slightly refactored for safer initialization and shutdown semantics. Device enumeration and map handling were hardened to prevent stale handles or nil dereferences during partial driver availability, improving resilience when GPUs are not yet ready or temporarily absent. These changes primarily improve reliability rather than extend measurement scope.

Process-level resolution depends on backend capabilities. Enterprise GPUs exposed through DCGM support per-process utilization, while consumer-grade devices using NVML typically provide aggregate values only. This limitation affects attribution accuracy but does not compromise energy sampling itself.

The current implementation was validated on a consumer GPU; full verification on data-center hardware will follow once suitable test systems become available. Despite these differences, the collector provides stable and temporally precise GPU telemetry, completing Tycho's set of primary energy input sources.

Appendix A

Appendix Title

Bibliography

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

[2] 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`.