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

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-Based CPU Time Attribution

1.2.1 Scope and Objectives

This section describes the kernel-level `eBPF` program implemented for Tycho. It captures CPU scheduling, interrupt, and performance-counter data required to attribute energy consumption to individual processes and system components. The focus lies on the in-kernel instrumentation that provides these measurements in near real time.

The `eBPF` program operates as a continuous, low-overhead tracing layer. It observes task scheduling events, interrupt activity, and selected hardware counters to record temporal ownership of CPU resources. All aggregation and interpretation occur later in userspace, outside the kernel. Consequently, this section excludes the user-level collectors and analysis routines, which are documented separately.

The objective is to provide accurate temporal metrics that represent how long each process, kernel thread, or system activity keeps a CPU core active. These metrics form the foundation for time-aligned energy estimation at the process level, complementing the coarser 50 ms energy readings obtained from RAPL.

1.2.2 Baseline: eBPF Programs in Kepler

Kepler's original design includes a small set of `eBPF` tracepoints that collect process-level CPU activity and selected hardware counter values. Its central hook, `tp_btf/sched_switch`, monitors context switches to determine how long each process remains on a CPU core. During a switch-out event, the previous task's runtime is computed and accumulated in a per-process map. This forms the foundation for attributing CPU time and hardware performance metrics to individual processes.

Additional probes in Kepler capture page cache activity and interrupt-related signals. The functions `fexit/mark_page_accessed` and `tp/writeback_dirty_folio` increment per-process counters whenever a page is read from or written to the cache. These events serve as indicators for I/O intensity but are not directly linked to CPU energy. Kepler also integrates hardware performance counters through the `perf_event_array` interface, exposing metrics such as retired instructions, CPU cycles, and cache misses.

While this design enables coarse process-level attribution, it provides only partial visibility into system-level activity. Kepler does not explicitly separate idle, interrupt, or kernel time, and it lacks continuous per-CPU state tracking between scheduling events. As a result, portions of CPU activity that occur outside user processes—such as interrupt handling or idle periods—remain unaccounted for. These limitations motivated several architectural extensions in Tycho to achieve a more complete temporal representation of CPU usage.

1.2.3 Motivation for Extending Kepler

Although Kepler's existing eBPF implementation captures essential process-level metrics, it does not provide the temporal precision required for reliable energy correlation. Its userspace aggregation loop collects samples in variable intervals that depend on polling frequency and system load, leading to timing misalignments with RAPL energy updates. As RAPL data becomes stable only over windows of roughly 50 ms, consistent time alignment between utilization and energy data is essential for meaningful attribution.

Another limitation lies in Kepler's treatment of CPU activity as a single, undifferentiated category. The kernel's own execution time, interrupt handling, and idle periods all appear as missing or ambiguous data in the process-level view. Without explicit classification of these states, total CPU energy cannot be partitioned into user, system, and idle components. This hinders both accuracy and interpretability of the resulting energy models.

Further, Kepler records the `cgroup_id` of a process only when its metrics are first created. Since task–to–cgroup assignments may change during runtime, this causes occasional mislabeling in containerized environments. Finally, Kepler lacks resettable per-CPU bins that would allow consistent temporal aggregation aligned with the 50 ms RAPL sampling intervals.

These limitations motivated Tycho's redesigned kernel instrumentation. By introducing continuous per-CPU state tracking, explicit accounting of idle and interrupt time, corrected cgroup labeling, and resettable bins, Tycho provides a temporally precise and energy-aligned foundation for process-level energy attribution.

1.2.4 Architecture Overview

The extended eBPF subsystem in Tycho introduces a coordinated set of kernel-level programs and data maps that together provide continuous, low-latency CPU activity monitoring. The design preserves Kepler's general architecture but refines it to achieve precise temporal alignment between kernel events, hardware counters, and energy measurements.

The main tracepoints include:

- `tp_btf/sched_switch`, which records task transitions and drives process-level runtime accounting.
- `tp/irq_handler_{entry,exit}` and `tp/softirq_{entry,exit}`, which measure time spent in hardware and software interrupt handling.
- Page cache probes (`fexit/mark_page_accessed` and `tp/writeback_dirty_folio`) inherited from Kepler, retained for I/O-related metrics.

Several new kernel maps support these programs:

- `processes` (LRU hash) stores per-task counters such as runtime, cycles, and cache misses.

- `pid_time_map` tracks per-thread timestamps to compute elapsed CPU time between context switches.
- `cpu_states` maintains per-CPU information on the currently executing task, its type, and the last scheduling timestamp.
- `cpu_bins` aggregates idle, IRQ, and softirq durations into resettable per-CPU time bins aligned to the energy sampling interval.
- `perf_event_arrays` provide access to hardware performance monitoring units (PMUs) for CPU cycles, instructions, and cache misses.

At runtime, each scheduling or interrupt event updates the relevant per-CPU and per-process structures inside the kernel. The userspace collector periodically retrieves these aggregates through `BatchLookupAndDelete` or `Lookup` operations, matching their temporal window with RAPL-based energy samples. This design enables a continuous flow of measurements from kernel to userspace without losing temporal precision, forming the basis for accurate process-level energy attribution in the subsequent analysis stage.

1.2.5 Kernel Programs and Responsibilities

1.2.5.1 Scheduler Switch (`tp_btf/sched_switch`)

The scheduler switch program remains the central element of the `eBPF` subsystem. It is triggered whenever the Linux scheduler replaces the currently running task with another, allowing precise observation of CPU ownership transitions. At each switch, the program computes the elapsed on-CPU time of the outgoing task and updates its entry in the `processes` map. Hardware counter deltas for instructions, cycles, and cache misses are collected in the same event, ensuring synchronized utilization metrics.

In addition to Kepler's original functionality, Tycho extends this program with explicit per-CPU state management. Each CPU maintains a small `cpu_state` structure that records the last timestamp, the currently running PID, and the task type. When the scheduler switches to the idle task (PID 0), this state is used to accumulate idle time directly in the kernel, preventing the loss of non-user activity between sampling intervals.

The program also detects kernel threads by inspecting the `PF_KTHREAD` flag in the task's `task_struct`. This information is stored in the corresponding process entry and allows the later separation of kernel activity from regular user processes. Finally, the `cgroup_id` of the newly scheduled task is refreshed at each switch, ensuring accurate container attribution even when processes migrate between control groups.

1.2.5.2 Hard Interrupts (`tp_btf/irq_handler_{entry,exit}`)

Two tracepoints record the entry and exit of hardware interrupt handlers. These events measure the time each CPU spends servicing hardware interrupts such as network or storage requests. On `irq_handler_entry`, the current timestamp is stored in the per-CPU `cpu_state`; on `irq_handler_exit`, the elapsed time is calculated and added to the corresponding counter in `cpu_bins`. This provides an accurate cumulative measure of hardware interrupt load for each sampling interval.

Unlike process runtime tracking, IRQ accounting is not associated with specific PIDs or cgroups, as interrupt handlers execute in kernel context. Instead, these durations are maintained purely per CPU and later aggregated as a separate category in the analysis stage. This allows Tycho to quantify system-level energy consumption that does not belong to any user process, reducing attribution errors when aligning RAPL energy data with process activity.

1.2.5.3 Soft Interrupts (`tp_btf/softirq_{entry,exit}`)

Soft interrupts represent deferred kernel work such as network packet processing, timers, or block I/O completion. Tycho extends Kepler with dedicated tracepoints for `softirq_entry` and `softirq_exit`, enabling explicit measurement of this activity. Each entry event records the current timestamp in the per-CPU `cpu_state`, while each exit event computes the elapsed time and adds it to the `softirq_ns` counter within `cpu_bins`.

To provide additional diagnostic context, each soft interrupt vector is also counted in the per-process map. Although these vectors are not directly linked to user-space processes, they serve as enrichment data to identify workloads that generate high deferred kernel activity. Together with hard interrupt accounting, this mechanism allows Tycho to capture the complete interrupt workload of the system, isolating its contribution to overall CPU energy consumption.

1.2.5.4 Page Cache Probes (`fexit/mark_page_accessed`, `tp/writeback_dirty_folio`)

Two existing Kepler probes were retained in Tycho to monitor page cache activity. The first, `fexit/mark_page_accessed`, triggers whenever a cached page is accessed, while `tp/writeback_dirty_folio` fires when dirty pages are written back to storage. Each event increments a counter in the `processes` map corresponding to the currently active task.

These metrics do not directly represent CPU energy but serve as secondary indicators of I/O intensity and memory subsystem activity. They provide contextual information that complements CPU-centric data, allowing workloads with similar computational behavior but differing memory access patterns to be distinguished. Their retention also ensures backward compatibility with Kepler's existing data model.

1.2.6 Kernel Data Structures (Maps)

The kernel maps form the persistent data layer of the `eBPF` subsystem. They hold the intermediate state between events and define how information is shared across CPUs and exported to userspace. Each map serves a specific role in maintaining temporal consistency and minimizing contention.

1.2.6.1 `processes` (LRU hash)

This map stores per-task aggregates such as runtime, hardware counter deltas, and page cache events. Each entry represents a thread group (`TGID`) and includes fields for `cgroup_id`, `is_kthread`, and the process name. The least-recently-used policy automatically evicts inactive tasks, keeping the memory footprint predictable even on busy systems.

1.2.6.2 `pid_time_map` (LRU hash)

A lightweight timestamp map that records the start time of each thread's on-CPU period. When a context switch occurs, the current timestamp is compared with the stored value to compute elapsed runtime, after which the entry is removed. This mechanism provides precise per-task CPU time measurement with negligible overhead.

1.2.6.3 `cpu_states` (per-CPU array)

Each CPU maintains a small state structure that holds its last scheduling timestamp, current PID, and flags for active IRQ or softirq handling. This allows continuous accounting of CPU ownership even between context switches, ensuring that idle and interrupt time are captured without loss.

1.2.6.4 `cpu_bins` (per-CPU array, resettable)

This map aggregates idle, IRQ, and softirq durations within each analysis window. Userspace periodically reads these counters and resets them, aligning the accumulated times with the 50 ms RAPL energy sampling interval. The bins therefore provide temporally bounded activity summaries for energy correlation.

1.2.6.5 Perf Event Readers

Kepler's existing `perf_event_array` readers are reused for CPU cycles, instructions, and cache misses. These PMU counters are read at each scheduling event, allowing per-task hardware-level activity to be combined with temporal metrics from the other maps.

Together, these data structures form a coherent kernel-side buffer that maintains a precise and low-overhead record of CPU activity. They enable a deterministic export path to userspace, where energy attribution can be performed with full temporal alignment.

1.2.7 Collected Metrics

The extended `eBPF` subsystem provides a unified set of metrics describing CPU activity at both process and system level. These values are aggregated in kernel maps and exported periodically to userspace, where they are aligned with RAPL energy samples. Table 1.2 summarizes the collected data and their respective sources.

Together, these metrics provide a comprehensive temporal and structural view of CPU utilization. Time-based data quantify how processor resources are distributed, hardware counters describe the intensity of execution, and classification attributes link activity to processes, kernel threads, or system services. This combined dataset enables precise alignment of utilization and energy measurements in the analysis phase.

1.2.8 Integration with Energy Measurements

RAPL provides the total CPU package energy consumption over discrete measurement intervals, typically aggregated in 50 ms windows. While this value reflects the overall electrical energy drawn by the processor, it offers no information on how that

TABLE 1.1: Summary of metrics collected by the kernel eBPF subsystem.

Metric	Source hook	Granularity	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; used to estimate IPC.
Cache misses	PMU (`perf_event_array`)	per process	Last-level cache miss count; indicates 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.

energy was distributed among processes, kernel activities, and idle states within the same period. The extended eBPF instrumentation in Tycho fills this temporal gap by describing the detailed ownership of CPU time across all execution contexts.

For each sampling window, the kernel maps record how long each entity—user process, kernel thread, interrupt handler, or idle task—occupied the CPU. When the corresponding RAPL energy delta is read, these time shares define a proportional basis for partitioning total energy. The process-level metrics account for user workloads, the aggregated kernel and interrupt times capture system overhead, and the idle counters isolate baseline power consumption unrelated to active computation.

This alignment of energy and utilization over identical time boundaries allows energy attribution to proceed without statistical inference or model-based extrapolation. Instead, energy is divided directly according to observed CPU activity, enabling a clear distinction between workload energy, kernel energy, and idle power. The result is a temporally coherent foundation for process-level energy analysis, upon which higher-level modeling and validation layers of Tycho can operate.

1.2.9 Performance and Overhead Considerations

The eBPF subsystem was designed to operate continuously with negligible impact on system performance. Several structural decisions ensure that metric collection remains lightweight even under high event rates.

All frequently updated data reside in per-CPU maps, eliminating contention between cores and avoiding global locking. Each CPU writes exclusively to its local entries in `cpu_states` and `cpu_bins`, while process-level updates use an LRU hash map that bounds memory usage and automatically evicts inactive entries. Arithmetic within tracepoints is deliberately minimal, limited to timestamp subtraction and counter increments.

Userspace collection is performed in batches through `BatchLookupAndDelete` operations, minimizing system call frequency and ensuring predictable latency regardless of map size. The data model avoids dynamic memory allocation and long-lived kernel objects, reducing both cache pressure and garbage buildup. Hardware counters are accessed via pre-configured `perf_event_arrays`, which the kernel maintains asynchronously and efficiently.

Together, these design choices allow the instrumentation to record fine-grained temporal and performance data at sub-millisecond latency while remaining transparent to the workload. In typical operation, the added CPU overhead remains well below one percent, even on systems with hundreds of context switches per second.

1.2.10 Correctness and Robustness

Several safeguards were incorporated into the eBPF design to ensure correctness under diverse kernel configurations and workloads. Struct definitions follow CO–RE (Compile Once, Run Everywhere) conventions, allowing field offsets in `task_struct` and related kernel types to be resolved dynamically at load time. This guarantees compatibility across kernel versions without requiring rebuilds.

Cgroup attribution is handled with particular care. The `cgroup_id` is refreshed only for the newly scheduled task during each `sched_switch` event, ensuring that the process entering the CPU is labeled correctly even if its control group membership changed during prior execution. This avoids misattribution in containerized environments where processes migrate between cgroups.

Idle and kernel-thread handling follow explicit rules. The idle task (PID 0) is recognized as a special case and contributes only to the per-CPU idle counters, never to user-level aggregates. Kernel threads are identified through the `PF_KTHREAD` flag and tracked separately to distinguish system activity from user workloads.

Finally, the resettable bin mechanism ensures strict temporal consistency between CPU activity and RAPL energy readings. Each collection cycle begins with zeroed bins, guaranteeing that every reported duration belongs exactly to one energy window. Together, these measures provide a robust and version-stable instrumentation layer that can operate continuously without producing inconsistent or overlapping samples.

1.2.11 Limitations and Future Extensions

Despite its improved temporal precision, the eBPF subsystem remains bounded by the resolution of the available energy telemetry. RAPL energy readings stabilize only over windows of roughly 50 ms, limiting the temporal granularity at which utilization can meaningfully be correlated with power data. Within shorter intervals, energy samples are too noisy to support reliable attribution.

The current implementation also omits processor C-state and frequency information. While idle time is captured explicitly, variations in core power states and frequency scaling are not yet reflected in the data. Integrating tracepoints such as `power:cpu_idle` and `power:cpu_frequency` would enable finer differentiation between active and low-power phases. Similarly, processes with extremely short lifetimes may be evicted from the LRU maps before userspace collection, leading to minor underrepresentation of transient workloads.

Future extensions may broaden the scope of per-CPU bins beyond the CPU domain, adding similar mechanisms for GPU activity, memory bandwidth, or I/O subsystems. These additions would enable a more comprehensive representation of platform-level energy distribution and further improve the precision of workload energy attribution across heterogeneous computing environments.

1.3 eBPF-Based CPU Time Attribution

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 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 1.2 summarizes all metrics grouped by category.

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.

1.3.5 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.

TABLE 1.2: Metrics collected by the kernel eBPF subsystem.

Metric	Source hook	Granularity	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; used to estimate IPC.
Cache misses	PMU (perf_event_array)	per process	Last-level cache miss count; indicates 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.

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.

1.3.6 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 typically well below one percent.

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.

1.3.7 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.

Future extensions may expand the binning mechanism to other domains, such as GPU, memory, or I/O subsystems, allowing unified energy attribution across heterogeneous hardware. These refinements would further enhance Tycho's capacity to capture platform-level energy behavior with high temporal precision.

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.