METRION: a framework for accurate software energy measurement

Benjamin Weigell, Simon Hornung, Bernhard Bauer

Angaben zur Veröffentlichung / Publication details:

Weigell, Benjamin, Simon Hornung, and Bernhard Bauer. 2026. "METRION: a framework for accurate software energy measurement." In *GREENS '26: proceedings of the IEEE/ACM 10th International Workshop on Green and Sustainable Software, April 12–18, 2026, Rio de Janeiro, Brazil*, 131–38. New York, NY: ACM. https://doi.org/10.1145/3786148.3788635.

METRION: A Framework for Accurate Software Energy Measurement

Benjamin Weigell
University of Augsburg
Augsburg, Germany
benjamin.weigell@uni-a.de

Simon Hornung
University of Augsburg
Augsburg, Germany
simon.hornung@uni-a.de

Bernhard Bauer
University of Augsburg
Augsburg, Germany
bernhard.bauer@uni-a.de

Abstract

The Information and Communication Technology sector accounted for approximately 1.4% of global greenhouse gas emissions and 4% of the world's electricity consumption in 2020, with both expected to rise. To reduce this environmental impact, optimization strategies are employed to reduce energy consumption at the IT infrastructure and application levels. However, effective optimization requires, firstly, the identification of major energy consumers and, secondly, the ability to quantify whether an optimization has achieved the intended energy savings. Accurate determination of application-level energy consumption is thus essential. Therefore, we introduce an energy attribution model that quantifies the energy consumption of applications on CPU and DRAM at the thread level, considering the influence of Simultaneous Multithreading, frequency scaling, multi-socket architectures, and Non-Uniform Memory Access. To ensure cross-platform applicability, we integrate the proposed model into an extensible framework, METRION, including a platform-independent data model and an initial implementation for Linux systems using Intel CPUs. We evaluate METRION across three different workloads and demonstrate that the energy attribution model can accurately capture the CPU energy consumption of applications targeting solely the CPU with a Mean Absolute Percentage Error of 4.2%, and the DRAM energy consumption of applications targeting DRAM with an 16.1% error.

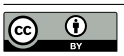
CCS Concepts

• **Hardware → Power estimation and optimization**; • **Software and its engineering → Software organization and properties**; • **Computer systems organization → Architectures**.

Keywords

software energy attribution, sustainable computing, energy efficiency, thread-level profiling

ACM Reference Format:
Benjamin Weigell, Simon Hornung, and Bernhard Bauer. 2026. METRION: A Framework for Accurate Software Energy Measurement. In *10th International Workshop on Green and Sustainable Software (GREENS '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3786148.3788635



GREENS '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2381-0/26/04
https://doi.org/10.1145/3786148.3788635

1 Introduction

Increasing digitalization boosts the importance of the Information and Communication Technology (ICT) sector, which supports areas from critical infrastructure to personal entertainment. However, this increase comes at a significant environmental cost. In 2020, the Greenhouse Gas (GHG) emissions of the ICT sector accounted for approximately 1.4% of global emissions and consumed around 4% of the world's electricity [17]. Moreover, it is predicted that by 2030, the ICT sector could consume up to 21% of global energy, with data centers contributing nearly one-third [1] . These estimates likely understate future consumption, as they do not account for the rising energy demands of AI and blockchain. While data centers are the visible energy consumers, the underlying cause lies in the software applications executed on these infrastructures.

Therefore, improving energy efficiency requires accurately determining the energy consumption of a software application. Such visibility enables the identification and optimization of energy hotspots at the macro- and micro-architectural levels. At the macro-architectural level, energy visibility enables the identification of energy-intensive applications and the development of strategies for reducing overall energy consumption, for example, by shutting down unused applications. At the micro-architectural level, detailed and accurate energy consumption allows fine-grained comparisons between different implementations. For instance, developers can evaluate whether reimplementing a component in a different programming language or applying specific code optimizations lead to measurable reductions in energy consumption [11].

To determine the energy consumption, recent hardware offers on-device measurement sensors that measure the energy consumption of the CPU and memory [15]. Moreover, RAPL measurements have been shown to correlate strongly with total system power consumption [15]. However, these measurements capture energy consumption at the component level and do not directly attribute it to a software application. To overcome this limitation, several attribution approaches combine runtime information with on-device energy measurements to infer application-level energy use.

Existing methods fall roughly into two categories. Utilization-based models [6, 13, 20], quantify energy consumption from coarse-grained resource utilization metrics, e.g., CPU time. However, they typically neglect hardware-level effects such as frequency scaling and Simultaneous Multithreading (SMT), which substantially influence energy behavior [5, 24]. Event-based models [5, 9, 24], in contrast, use Hardware Performance Counters (HWPCs) that correlate closely with CPU power consumption [3, 4]. While these approaches capture fine-grained thread-level detail, they often compromise transparency or completeness. Some approaches model CPU energy consumption and account for SMT effects, but neglect

frequency variation and DRAM energy [5, 24]. Other dynamically select HWPC events and builds frequency-specific regression models, but neglecting SMT and Non-Uniform Memory Access (NUMA) effects [9]. Overall, existing models remain limited in scope and do not account for the combined effects of frequency scaling, SMT, and NUMA in multi-socket systems, thereby limiting their accuracy.

Accordingly, we want to answer this research question (RQ): *How can the energy consumption of software applications be accurately and transparently attributed to underlying hardware components, such as the CPU and memory, in complex multi-socket and SMT environments, using only the measurements available from on-device sensors?* To answer this research question, we present the following contributions:

(1) An interpretable thread-level energy attribution model that determines CPU and DRAM energy using HWPCs, that explicitly accounts for multi-socket topology, NUMA locality, frequency scaling, and SMT, with final aggregation at the application level.
(2) METRION, a modular, extensible, and platform-agnostic framework for realizing the energy attribution model.
(3) A platform-agnostic and extensible data model for platform-independent application of the energy attribution model.

2 Related Work

On the one hand, one of the first approaches to determine the energy consumption of software applications is introduced by Flinn et al. [10] with their tool PowerScope. PowerScope correlates external power measurements with kernel events to map energy consumption to software components. On the other hand, one of the first HWPC-based approaches is JouleWatcher by Bellosa [3], which determines the energy consumption of a thread based on HWPC events. The approach requires an initial calibration step in which the energy per event is determined. Bellosa [3] demonstrates that events such as retired micro-operations per second, floating-point operations per second, second-level cache address strobes, and memory bus transactions correlate with system energy consumption. Do et al. [8] introduce pTop, one of the early solely software-based tools for determining the energy consumption of applications. pTop determines the energy consumption of an application by correlating predefined power values of the CPU, hard disk and network interface card (NIC) with the measured resource utilization of these components by the application. In this utilization-oriented line of work, Noureddine et al. [19] introduce PowerAPI, a purely software-based framework that calculates the power consumption of applications based on their CPU and NIC usage through different power models. Around the same time, Bircher and John [4] propose a software-based approach to determine the power consumption of a complete system using HWPCs. They demonstrate that events such as Cycles, Fetched µops, and L3 cache load misses, can be used to build linear regression models that calculate the power consumption of the CPU and memory, respectively.

Intel introduced, around 2012 with the Sandy Bridge architecture, the Running Average Power Limit (RAPL) interface [22]. Depending on the chip's architecture, RAPL exposes energy consumption data for different power domains of the processor. For example, the Package domain includes the core and uncore components of the chip [15], while the DRAM domain reports the energy consumption of the attached integrated memory controller. With the introduction of RAPL two kinds of energy granularity attribution tools have emerged [14], software that determines the energy consumption of the whole system, such as Carbon Tracker [2], CodeCarbon [6], and Experiment Impact Tracker [12]. And energy measurement software that determines the energy consumption of a single application, such as HaPPy [24], DEEP-mon [5], a new version of PowerAPI [9], Scaphandre [20] and EnergAt [13], HaPPy [24] uses RAPL energy readings as input for a hyperthread-aware Energy Attribution Model (EAM) to determine the energy consumption of an application. This model leverages non-halted CPU cycles together with hyperthreading information to determine the share of energy consumption for each application. DEEP-mon [5] refines the approach of HaPPy by making the hyperthread-aware energy attribution resilient to time-shared CPUs and extending its coverage to application containers. Furthermore, Fieni et al. [9] replace the EAM of PowerAPI with an regression-based one that uses RAPL counters and dynamically selects HWPCs to build a frequency-aware EAM. This model determines the energy consumption of an application on both the CPU and DRAM, however primarily targets containers. Scaphandre [20] determines the energy consumption of an application by attributing the CPU energy consumption reported by RAPL according to its time share on the CPU. With EnergAt Hè et al. [13] propose an EAM that determines the energy consumption at a thread level while considering influencing factors such as multi-socket NUMA architectures, the noisy-neighbour effect and and multi-tenancy. In summary, existing approaches remain limited in their scope because they do not take into account the combined effects of frequency scaling, SMT, and NUMA in multi-socket systems, thereby limiting their accuracy.

3 Energy Attribution Model

Given the increasing availability of on-device sensor hardware that reports the current energy consumption of CPU and memory components, we present an Energy Attribution Model (EAM) that redistributes these component-level energy measurements to threads based on HWPC events and thereby determines the energy consumption of a software application at the thread level.

3.1 Design Considerations

The design of the EAM is influenced by hardware and execution characteristics that affect the energy consumption of the application and are discussed in the following as key design considerations.

3.1.1 Idle and Active Power Consumption. The power consumption of a hardware component consists of idle and active parts. Idle power represents the baseline consumption independent of resource usage, while active power depends on actual resource usage. An EAM should attribute active energy to applications based on their resource usage and distribute this idle baseline energy fairly among all running applications.

For example, a CPU with a total power consumption of $50W$ under load, of which $10W$ is idle power, running two processes A and B using 10% and 20% of the CPU resources, respectively. If idle power is ignored, the total power ($50W$) may be distributed

proportionally to utilization, resulting in $16.7W$ for process A and $33.3W$ for process B. In contrast, when idle power is considered, the active portion ($50W - 10W = 40W$) is first distributed according to utilization: process A consumes 13.3 W and process B $26.7W$. The idle power is then divided equally between the two ($5W$ each), leading to total allocations of $18.3W$ and $31.7W$, respectively an 8.7% and 5.0% difference. Therefore, EAMs should first isolate active power and then distribute idle power fairly among consumers.

3.1.2 Multiple-Socket Architectures. Multiple-socket architectures, in which each socket comprises a CPU package and an associated memory domain, are commonly employed in modern server hardware [13]. However, most current power models do not consider the resulting NUMA effect and therefore incorrectly distribute the energy consumed by applications [13]. This aspect becomes increasingly important as current on-device sensors, such as Intel RAPL, provide energy readings for each CPU package and its attached memory independently. Therefore, an EAM must consider where the applications consume energy.

3.1.3 Thread-Level Execution and SMT. Modern applications and operating systems employ both multi-processing and multi-threading. An application can consist of multiple processes, each containing one or more threads. In modern operating systems, the thread is the basic unit of execution and can be viewed as a lightweight process [21]. The kernel schedules these individual threads on a CPU core. Additionally, SMT enables two or more threads, depending on the processor vendor and model, to execute concurrently on the same physical core. In the case of SMT with factor two, two threads can execute concurrently on the logical cores of the same physical core, as illustrated in Figure 1a. However, this concurrency affects energy consumption, as concurrent execution has been shown to increase total energy usage by approximately 15 % when using SMT with a factor of two, compared to single-threaded execution [5].

3.1.4 Frequency Effects. The power consumption of the CPU is approximated by the CMOS circuit power model, whereby it consists of A the switching activity, C the physical capacitance, V the supply voltage, and f for clock frequency [7] and is given by: $P = A \cdot C \cdot V^2 \cdot f$. Furthermore, the supply voltage scales approximately linearly with frequency, so that $P \approx A \cdot C \cdot f^2$ applies [18]. Hence, an EAM attributing CPU power consumption to an application should consider the frequency.

3.2 Model Foundations

Before presenting the model itself, we first formalize its foundations by defining how components and their active and idle energy are represented, as well as how an application is modeled.

3.2.1 Components and their Active and Idle Energy. For a given device, we consider each hardware component that provides independent energy measurements via on-device sensors, such as Intel RAPL. The current model focuses on the set of CPU packages P, where each package consists of a set of logical cores, and DRAM components D, which together form the set of considered components C:

$$\begin{aligned} C &= P \cup D \\ &= \{CPU_1, \ldots, CPU_n, \} \cup \{DRAM_1, \ldots, DRAM_m\} \quad n, m \in \mathbb{N} \end{aligned} \tag{1}$$

For each component $c \in C$, the total, active, and idle energy measured over the fixed time interval Δt are denoted by E^c_{total}, E^c_{active}, and E^c_{idle}, respectively. Subtracting the idle energy from the total energy, results in the active energy, as shown in Equation (2). The on-device sensors provide E^c_{total} for each component c during the interval Δt, as well as E^c_{idle} under an appropriate configuration.

$$E^c_{active} = E^c_{total} - E^c_{idle} \tag{2}$$

3.2.2 Dividing Applications into Threads. To determine the energy consumption of an application a_i, the energy attribution model considers each application as a set of threads $t_j^{a_i}$ that belong to the application and are executed during the time interval Δt:

$$a_i = \{t_1^{a_i}, t_2^{a_i}, \ldots t_l^{a_i}\} \quad l \in \mathbb{N} \tag{3}$$

Furthermore, a thread is scheduled multiple times on a core for execution, and each scheduling-in and -out of a thread on a core is captured by an execution interval $e_k^{t_j^{a_i}}$. Each execution interval forms a triple containing the core, the scheduling-in timestamp, and the scheduling-out timestamp. For a thread $t_j^{a_i}$, the set $e^{t_j^{a_i}}$ contains all of the thread's execution intervals $e_k^{t_j^{a_i}}$:

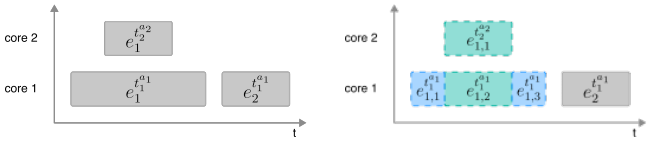
$$e^{t_j^{a_i}} = \{e_1^{t_j^{a_i}}, e_2^{t_j^{a_i}}, \ldots e_p^{t_j^{a_i}}\} \quad p \in \mathbb{N} \tag{4}$$

The duration of an execution interval $e_k^{t_j^{a_i}}$ is given by the duration function $\delta(e_k^{t_j^{a_i}})$, and the CPU package to which the core belongs is determined by the location function $L(e_k^{t_j^{a_i}}, c)$. The location function evaluates to 1 if the core on which the thread is executed during the interval belongs to CPU package c, and 0 otherwise.

Furthermore, to account for SMT, the execution interval $e_k^{t_j^{a_i}}$ of a thread $t_j^{a_i}$ is split into sub-intervals. If the thread $t_j^{a_i}$ is executed concurrently with other threads on the same physical core, the execution interval is divided into consecutive sub-intervals during which it runs either exclusively or concurrently with other simultaneously running threads. Otherwise, if the thread is executed exclusively on a single physical core, the execution interval $e_k^{t_j^{a_i}}$ remains unsplit. The final set of execution intervals for a thread $t_j^{a_i}$ is defined as:

$$e_k^{t_j^{a_i}} = \{e_{k,1}^{t_j^{a_i}}, e_{k,2}^{t_j^{a_i}}, \ldots, e_{k,s}^{t_j^{a_i}}\} \quad s \in \mathbb{N} \tag{5}$$

An example of this splitting is illustrated for two threads $t_1^{a_1}$ and $t_2^{a_2}$, with three execution intervals $e_1^{t_1^{a_1}}$, $e_2^{t_1^{a_1}}$ and $e_1^{t_2^{a_2}}$. The intervals $e_1^{t_1^{a_1}}$ and $e_1^{t_2^{a_2}}$ are executed concurrently, resulting in the sub-intervals $e_{1,1}^{t_1^{a_1}}$, $e_{1,2}^{t_1^{a_1}}$, $e_{1,3}^{t_1^{a_1}}$, as illustrated in Figure 1b.



(a) Parallel thread execution. **(b) Execution interval splitting.**

Figure 1: Concurrent thread execution and execution interval splitting of two threads

3.3 The Energy Attribution Model

The energy consumption of an application a_i is the sum of its active and idle energy shares for each component, as shown in Equation (6), thus addressing the first two considerations. The active and idle energy shares of an application are determined from the energy consumption of its threads, which in turn are computed over their respective execution intervals, as described in Equation (7).

$$E_{total}(a_i) = \sum_{c \in C} \left(E^c_{active}(a_i) + E^c_{idle}(a_i) \right) \tag{6}$$

$$E^c_{active}(a_i) = \sum_{j,k,l} E^c_{active}(e^{t_j^{a_i}}_{k,l}), \qquad E^c_{idle}(a_i) = \sum_{j,k,l} E^c_{idle}(e^{t_j^{a_i}}_{k,l}) \tag{7}$$

3.3.1 Active Energy Attribution. The active energy consumption of a thread during an execution interval, $E^c_{active}(e^{t_j^{a_i}}_{k,l})$, is computed independently for each component c, and the total active energy of the thread is obtained by summing these per-component contributions. In both cases, the energy consumption depends on the amount of work the thread performs during this interval on a component c, relative to the total work performed on that component in the same interval. To capture this, we define component-specific work functions $w_c(\cdot)$ that quantify the activity of a thread on component c, e.g., $w_c = w_{CPU}$ for computational activity, when $c \in P$, and $w_c = w_{DRAM}$ for memory activity, when $c \in D$. Based on these work functions, the active energy of component c attributed to an execution interval is given by Equation (8):

$$E^c_{active}(e^{t_j^{a_i}}_{k,l}) = \frac{w_c(e^{t_j^{a_i}}_{k,l}, c)}{\sum_{i',j',k',l'} w_c(e^{t_{j'}^{a_{i'}}}_{k',l'}, c)} \cdot E^c_{active} \tag{8}$$

CPU Work Function. The CPU work function quantifies the amount of CPU work performed during an execution interval. Thereby, the work function respects the design considerations of multiple-socket architectures, thread-level execution and SMT, and frequency effects.

Each thread executes on a specific logical core within a CPU package c. To capture the general work a thread performs during an execution interval, the HWPC Unhalted Core Cycles per Thread (*UCC*) is used. The HWPC *UCC* captures the number of core cycles executed by a logical core up to a given timestamp, and the difference, ΔUCC, between two timestamps strongly correlates with the CPU's energy consumption during that interval [3–5, 9, 24]. However, to account for frequency scaling effects, ΔUCC is scaled with the core's average frequency ratio during this time interval. To calculate the average frequency ratio, the HWPCs *APERF* and *MPERF* are used. *APERF* and *MPERF* measure actual and maximum performance clock counts per logical core, respectively. The HWPC *APERF* changes with the current operating frequency. In contrast, *MPERF* increments at the logical core's reference frequency. The frequency ratio $\frac{\Delta APERF}{\Delta MPERF}$ is greater than 1 if the logical core operates above the base frequency, such as with turbo boost. It falls below 1 when the core operates below the base frequency, for example, under energy-saving conditions. Scaling the work by this ratio adjusts it to match the effective operating frequency.

To consider SMT effects, ΔUCC is scaled with an SMT factor that is determined by the σ function. The σ function equals 1 if the thread executes exclusively on the physical core, and 1.15 when SMT is active during the interval [5]. Returning to the previous example, the execution intervals $e^{t_1^{a_1}}_{1,2}$ and $e^{t_2^{a_2}}_{1,1}$ are SMT-active phases and are scaled with a factor of 1.15, while the other intervals represent exclusive execution and are scaled with 1, as illustrated in Figure 1b, where the green intervals indicate SMT-active phases.

Overall, the work performed by a thread during the execution time interval on the given CPU package c is specified by Equation (9), whereby the previously defined location function $L(\cdot)$ ensures that only the work performed on the corresponding CPU package contributes to the result:

$$w_{CPU}(e^{t_j^{a_i}}_{k}, c) = \Delta UCC \cdot \frac{\Delta APERF}{\Delta MPERF} \cdot \sigma(e^{t_j^{a_i}}_{k}) \cdot L(e^{t_j^{a_i}}_{k}, c) \tag{9}$$

DRAM Work Function. The DRAM work function quantifies a thread's memory activity, which correlates with energy consumption, during an execution interval, while accounting for the design considerations of multi-socket architectures and thread-level execution. The energy-relevant memory activity of a thread during an execution interval is quantified by the number of off-chip memory transactions it causes that are served by a DRAM component. Off-chip transactions include read and write operations, but only read transactions are modelled, as write operations are first written to the cache and later transferred to the DRAM during cache line eviction, which makes them difficult to attribute to a specific execution interval. The read transactions are recorded by a HWPC that counts DRAM-serviced LLC-miss loads per DRAM component. The change in this counter for DRAM component d during an interval is given by $\Delta O_d(e)$. Further, to account for higher energy consumption due to remote memory access, remotes are scaled with the weighting factor γ. For local DRAM accesses, $\gamma = 1$, and for remote accesses, $\gamma = 9.67$, as DRAM energy consumption increases by this factor for memory-intensive applications due to remote access [23]. Accordingly, the work performed by a thread on a DRAM component during an execution interval is given by Equation (10).

$$w_{DRAM}(e^{t_j^{a_i}}_{k}, d) = \Delta O_d(e^{t_j^{a_i}}_{k}) \cdot \gamma \tag{10}$$

3.3.2 Idle Energy Attribution. The idle energy attributed to an execution interval of a thread is determined by proportionally assigning the idle energy of the component c measured during the monitoring interval Δt to that interval. For the CPU, it is attributed according to the thread's time share on the component c during Δt,

and for the DRAM, it is distributed in proportion to the number of concurrently active threads on the component c during the same monitoring interval, as shown in Equation (11).

$$E^c_{idle}(e^{t^{a_i}_j}_{k,l}) = \begin{cases} \dfrac{\delta(e^{t^{a_i}_j}_{k,l}) \cdot L(e^{t^{a_i}_j}_{k,l}, c)}{\sum_{i',j',k',l'} \delta(e^{t^{a_{i'}}_{j'}}_{k',l'}) \cdot L(e^{t^{a_{i'}}_{j'}}_{k',l'}, c)} \cdot E^c_{idle}, & \text{if } c \in P, \\ \dfrac{1}{n(\Delta t, c)} \cdot E^c_{idle}, & \text{if } c \in D. \end{cases} \tag{11}$$

With $n(\Delta t, c)$ representing the number of active threads observed on component c during the monitoring interval Δt.

4 METRION

The proposed energy attribution model is implemented within the METRION framework. The main objective of the framework is to determine the energy consumption of applications on different platforms using the proposed energy attribution model. However, achieving platform independence is challenging due to the diversity and continuous evolution of operating systems and hardware. The following section describes how this challenge is addressed within the architecture of the framework and by an extensible, platform-independent data model. In addition, an initial implementation is provided for Linux-based systems with Intel processors.

4.1 Platform Independent Data Model

To ensure platform independence, all components of the METRION framework exchange data through the platform-independent data model (PIDM). The PIDM provides a standardized representation of the required data and abstracts hardware- and operating-system-specific details and thereby enables the framework to operate consistently across platforms. The model is visualised in Figure 2

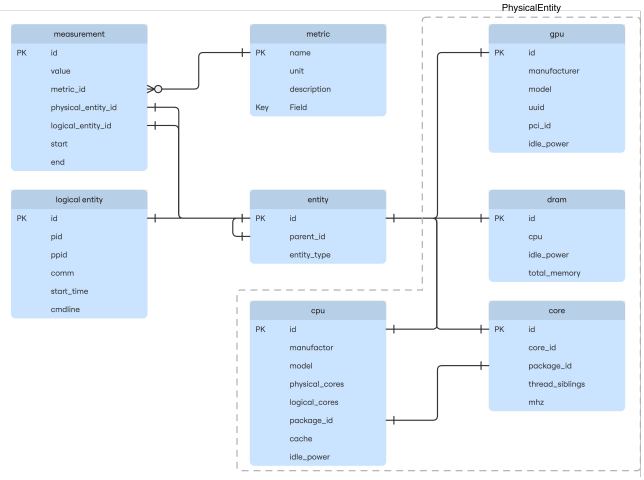


Figure 2: The Platform Independent Data Model

The PIDM is structured into the four main parts: Metric, LogicalEntity, PhysicalEntity and Measurement. The PhysicalEntity set represents the component set C of the energy attribution model and includes the physical components such as CPUs, Cores, and DRAM banks along with their minimal platform-agnostic metadata information; also, the model is prepared to support GPUs, though not yet integrated. The LogicalEntity models the application set A and their corresponding threads. Next, the metric defines what is captured, for example *APERF* or E^c_{IDLE}. Finally, the Measurement expresses where, what, when, and how much is measured by linking these elements together: it references the PhysicalEntity on which the event occurred, the Metric describing what was measured, the execution interval via start and end timestamps, and, if applicable, the LogicalEntity for the involved thread. For example, to express ΔUCC for an execution interval, the following measurement would be taken. The measurement references $t^{a_i}_j$ through the logical_entity_id, identifies the corresponding core via the physical_entity_id, defines the execution interval using the start and stop timestamps, and stores the measured ΔUCC in the value attribute while referencing the Metric UCC to indicate that the value represents unhalted core cycles per thread.

4.2 Architecture and Dataflow

The METRION framework is divided into the data-gathering, storage, energy-attribution and orchestration components, connected through the PIDM, as shown in Figure 3.

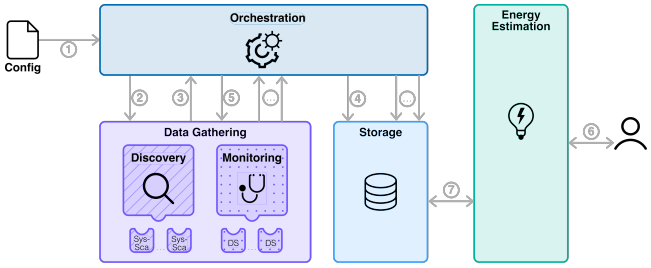


Figure 3: Metrion Architecture and Dataflow

The data gathering component is responsible for obtaining the necessary data points through its discovery and monitoring modules and transforming them into the platform-independent data representation required by the energy attribution model. The discovery module identifies and collects metadata about the physical components of the system and exposes these information as PhysicalEntity data objects. To deal with platform dependency, the discovery module offers the system scanning endpoint, through which platform-dependent discovery can be integrated. Hence, to support a new OS only a new dedicated system scanning functionality must be provided. The monitoring module obtains the measurements required by the energy attribution model through a set of data sources and provides them as Measurement data objects. To integrate heterogeneous data source backends for the monitoring module, a data source interface is provided. Each backend implementation specifies the physical resources and corresponding metrics it can collect. This design enables new data source backends to be added without modifying the core framework.

Next, the storage component stores the data, provided by the data gathering component, using a suitable storage backend. Since

the data is consistently represented in this platform-independent format, the storage backend can be changed as needed without affecting the rest of the framework.

The energy attribution component forms the core of METRION. It implements the proposed energy attribution model to determine the energy consumption of the applications. To perform this attribution, the component retrieves the necessary measurement data from the storage component. Due to the platform-independent data model, the energy attribution process is rendered independent of platforms and hardware configurations. In addition, the component provides an endpoint that allows users to query the attributed energy consumption of a given application.

The orchestration component integrates the independent modules and coordinates the entire data and execution flow, as visualised in Figure 3. In the setup phase, the orchestrator first reads a configuration file that specifies parameters such as the monitoring time resolution and the selected storage backend ①. It then calls the discovery module to collect metadata about the physical resources ②,③ and stores this information using the storage component ④. Based on the discovered hardware information, the orchestrator instructs the monitoring module to collect the measurements required by the energy assessment model ⑤, which then uses the provided hardware information to select and configure the appropriate data sources. After the setup phase, the orchestrator regularly retrieves the collected metrics from the monitoring module ◌ and forwards them to the storage component ◌. Finally, users can query the energy attribution component to obtain the attributed energy consumption values ⑥,⑦.

4.3 Initial Implementation with Linux and Intel Extensions

The METRION framework is primarily implemented in Python. The data-gathering and orchestration components are written entirely in Python, while the storage and energy attribution modules combine Python with SQL. The framework defines the system scanning and data source interfaces as abstract base classes to ensure consistent integration of new extensions.

Python is chosen for its portability across operating systems and its wide adoption, enabling easy extension of the framework with new system-scanning and data-source submodules. The storage component persists all measurements in an SQLite database, and the analysis module implements the EAM by combining Python logic with SQL queries. SQL is employed to leverage the database engine's built-in optimization and aggregation capabilities.

In order to support Linux-based systems with Intel CPUs featuring an SMT factor of two and two CPU packages, the METRION framework is extended with a system-scanning and two monitoring submodules. The system-scanning component uses /proc/cpuinfo and dmidecode to gather CPU and DRAM metadata. In addition, it determines the idle power of the CPU and DRAM by freezing all non-essential cgroups, after which the system remains idle for 60 seconds, and the average idle power of the CPU and DRAM is determined based on Intel RAPL readings.

To obtain the current energy consumption values of the CPU E^{CPU}_{total} and DRAM E^{DRAM}_{total} a data source for the Intel Powercap interface is implemented. In addition, an eBPF-based data source captures the required per-thread data points, i.e. $\Delta APERF$, $\Delta MPERF$, ΔUCC, $\Delta Reads^{DRAM}_{local}$, and $Reads^{DRAM}_{remote}$, the core location, and the SMT factor. Therefore, the eBPF program attaches to the sched_switch tracepoint and then employs perf events to collect these metrics for each execution interval of a thread. Whether SMT is active during a thread's execution interval is determined in post-processing. This approach is characterised by low-overhead, high-precision, and continuous sampling directly in the kernel and thereby reducing the likelihood that the work of a thread is not captured. The implementation is publicly available on GitHub [1].

5 Evaluation

The main goal of the proposed energy attribution model is to accurately capture the energy consumption of applications. To quantify its accuracy and assess its impact on system performance, the model is evaluated empirically across three dimensions.

The first dimension, attribution accuracy, measures how precisely the model attributes the active energy consumption of a workload on a component basis. Thereby, the accuracy is quantified with the Mean Absolute Percentage Error (MAPE) between the attributed value and a ground-truth reference. MAPE is chosen because its scale independence allows comparison between workloads and components with different energy ranges. The second dimension, stability, evaluates the consistency of the model's attribution accuracy across three workloads, where the first workload primarily stresses the CPU, the second the memory, and the third both components. Thirdly, the dimension comparative accuracy compares the assessed accuracies to established tools. The following sections first describe the in-detail evaluation design, followed by the results and their discussion.

5.1 Evaluation Design

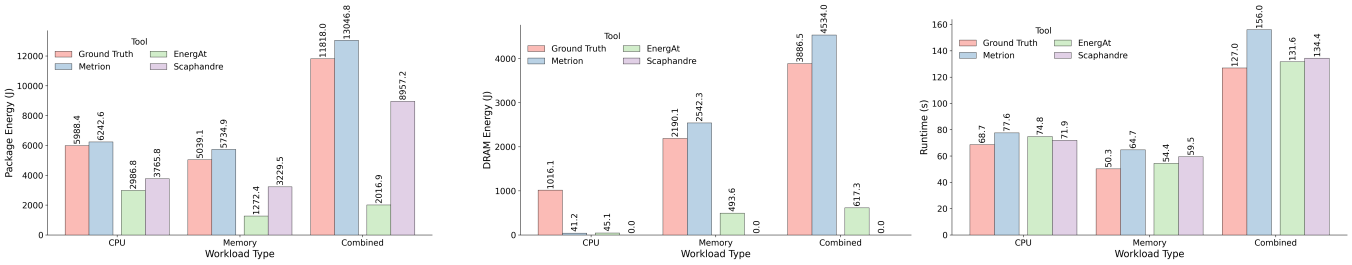
To achieve reproducibility, the hardware environment, the experimental condition, and the ground-truth measurement procedure, as well as the selected workloads and energy measurement tools are described in detail.

Hardware Setup: All experiments are conducted three times on a system equipped with two Intel Xeon Silver 4110 CPUs, each with 8 physical cores and an SMT factor of two. The system is equipped with 384 GB of DDR4 ECC memory and runs Ubuntu 24.04.3 LTS with the Linux 6.8.0-85-generic kernel.

Experimental Condition: To provide a controlled environment, all nonessential cgroups are frozen, and only the workload and a measurement tool are executed. Additionally, the CPU frequency governor is set to on-demand, and turbo mode is enabled to capture frequency-scaling effects.

Ground Truth Measurement: The ground-truth reference energy consumption is determined separately for each workload in two steps. In the first step, the idle power of each CPU and DRAM component is measured. Therefore, after all nonessential cgroups are

[1] https://github.com/ds-lab/Metrion



(a) CPU Package Energy **(b) DRAM Energy** **(c) Runtime**

Figure 4: CPU package energy, DRAM energy, and runtime results, each determined across three workloads: CPU-, Memory-, and Combined-workload.

frozen, the RAPL counters are read for each component, and the system is kept in this state for 60 seconds. Afterward, the RAPL counters are reread. The idle power consumption of each component is then determined by calculating the energy difference between two measured values and dividing it by the measurement interval. Following the idle power measurement, the active energy consumption for a given workload is determined in the second step. The nonessential cgroups remain frozen. The RAPL counters are read immediately before and after the workload execution, and the runtime of the workload is recorded. The difference between these readings gives the total energy consumption of the execution. To obtain the active energy, the idle energy is subtracted, which is calculated by multiplying the previously measured idle power by the workload's runtime.

Workloads: Three stressor from the stress-ng suite [16] are used to assess the stability of the energy attribution model. The `CPU workload` uses the cpu-method int32 stressor with $2,000,000$ operations to stress only the CPU, while the `Memory workload` uses the stream stressor with $200,000$ operations to specifically stress the memory. To stress both components simultaneously, the two stressors are combined to form the `Combined workload`.

Energy Measurement Tools: For the third dimension, the comparative accuracy, the tools Scaphandre and EnergAt are selected, as these tools determine energy consumption at the application level. DEEP-Mon is not included in the comparison, as it is no longer actively maintained. Scaphandre reports the total CPU energy consumption, including the idle share, for each application, whereas EnergAt reports the active energy consumption of the CPU and DRAM components separately at the application level. The total EAM employed by Scaphandre is explicitly considered in the interpretation and discussion of the results.

5.2 Results and Discussion

The results are presented and discussed per physical component, and for each, the attribution accuracy, stability, and comparative accuracy are analyzed. Additionally, the overall system impact introduced by the measurement tools is examined.

5.2.1 CPU Energy Attribution Accuracy. The proposed EAM determines CPU energy consumption for workloads with relatively high accuracy, with an average MAPE of 9.5% across all three workloads. For comparison, EnergAt and Scaphandre achieve average MAPEs of 69.3% and 32.4%, respectively. The model performs best for the `CPU workload`, with an MAPE of 4.2%, and worst for the `Memory workload`, with an MAPE of 13.8%, as shown in Figure 4a. The low error for the `CPU workload` and the higher error for the `Memory workload` may be due to the design of the RAPL package's power domain and the model's focus on core components. The package power domain includes the core components and uncore components, such as the last-level cache. In the case of the `CPU workload`, mainly the core components are stressed, whereas for the `Memory workload`, both the core and uncore components are utilized. As the proposed model does not explicitly model the uncore activity by a workload, a higher error is introduced. In contrast, both other tools determine the CPU energy consumption considerably lower across all three workloads, including Scaphandre, even though it reports total energy consumption. Nevertheless, Scaphandre captures the overall trend of the ground-truth energy consumption and shows that when the ground-truth increases, as in the case of the `Combined workload`, the reported value also increases, whereas for the `Memory workload`, both values decrease. EnergAt, however, does not capture this trend. Overall, the proposed model accurately captures CPU energy consumption, and its standard deviation of 3.95% and coefficient of variation of 4.17% indicate stable performance across different workload types.

5.2.2 DRAM Energy Attribution Accuracy. The DRAM energy consumption of a workload is partially accurately determined by the proposed EAM across different workloads. For the `CPU workload`, the model underestimates DRAM energy consumption, with an MAPE of 95.6%. This may result from CPU operations targeting DRAM that do not correlate with read operations as modeled by the EAM. In contrast, for the `Memory workload` and `Combined workload`, the MAPE decreases notably to around 16%, as shown in Figure 4b. This improvement may occur because these workloads stress the memory through read operations captured by the model, while the remaining error likely arises from unmodeled write operations. Only EnergAt reports per-application DRAM energy consumption and captures the overall trend across workloads, but still determines absolute values considerably lower, with an average MAPE of 85.7%. Overall, the EAM requires refinement, as it captures DRAM energy consumption with mixed accuracy.

5.2.3 System Impact Analysis. All three tools affect system performance, as their monitoring increases the total runtime of each workload. However, the impact varies with the workload type, with the `Memory workload` showing the most significant impact and the CPU workload the least, as shown in Figure 4c. Metrion causes the highest average increase in runtime across all three workloads at 21.1%, followed by Scaphandre at 9.2% and EnergAt at 6.5%. The different effects on system performance may be caused by the different implementation languages used. Metrion is mainly implemented in Python, Scaphandre in Rust, and EnergAt uses C/C++ for performance-critical components. To reduce Metrion's higher overhead, performance-critical parts could be implemented in Rust.

6 Conclusion

Accurately determining the energy consumption of applications is required to identify optimization potential at both the macro- and micro-architectural levels and to quantify the improvements achieved by optimization. To achieve this, we introduced an EAM that accounts for the influence of SMT, frequency scaling, multi-socket architectures, and NUMA on the energy consumption of an application at the thread level. In order to use the EAM across different platforms and at both the micro- and macro-architectural levels, we propose the extensible framework METRION, along with a platform-independent data model. Additionally, we provided an initial implementation of METRION for Linux-based systems using Intel CPUs. The evaluation demonstrates that the EAM can accurately capture the CPU energy consumption of applications targeting solely the CPU with a MAPE of 4.2%, and applications that target the DRAM explicitly with a MAPE of 16.1%. However, the framework increases the runtime of workloads, introducing a system overhead. In further work, we aim to improve the EAM and the METRION framework. Firstly, we plan to refine the CPU model by accounting for uncore activity, and the DRAM model by capturing DRAM write activity. Additionally, we intend to extend the overall model to capture energy consumption resulting from network activity and GPU usage. Secondly, we plan to reduce the system impact of METRION by rewriting performance-critical components in Rust. Thirdly, we plan to evaluate METRION on additional hardware platforms, such as AMD-based systems, and to systematically study how varying the number of executed workload operations influences attribution accuracy. Overall, the proposed EAM and framework lay the foundation to identify and quantify optimisations across micro- and macro-architectural levels, such as within the edge-fog-cloud continuum.

Acknowledgments

Funding. This research was supported by the Federal Ministry of Research, Technology and Space under grant number 16IS24070G.

Transparency note. The readability of this text has been improved with the support of AI technologies.

References

[1] Anders S. G. Andrae and Tomas Edler. 2015. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges* 6, 1 (2015), 117–157. doi:10.3390/challe6010117

[2] Lasse F Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. *arXiv* (2020). arXiv:2007.03051 doi:10.48550/arxiv.2007.03051

[3] Frank Bellosa. 2000. The benefits of event: driven energy accounting in power-sensitive systems. *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system* (2000), 37–42. doi:10.1145/566726.566736

[4] W. L. Bircher and L. K. John. 2012. Complete System Power Estimation Using Processor Performance Events. *IEEE Trans. Comput.* 61, 4 (2012), 563–577. doi:10.1109/tc.2011.47

[5] R. Brondolin, T. Sardelli, and M. D. Santambrogio. 2018. DEEP-Mon: Dynamic and Energy Efficient Power Monitoring for Container-Based Infrastructures. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2018), 676–684. doi:10.1109/ipdpsw.2018.00110

[6] Benoit Courty, Victor Schmidt, Sasha Luccioni, et al. 2024. *mlco2/codecarbon: v2.4.1.* doi:10.5281/zenodo.11171501

[7] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. 2015. Data Center Energy Consumption Modeling: A Survey. *IEEE Communications Surveys & Tutorials* 18, 1 (2015), 732–794. doi:10.1109/comst.2015.2481183

[8] Thanh Do, Suhib Rawshdeh, and Weisong Shi. 2009. pTop : A Process-level Power Profiling Tool. https://api.semanticscholar.org/CorpusID:18469514

[9] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. 2020. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)* 00 (2020), 479–488. arXiv:2001.02505 doi:10.1109/ccgrid49817.2020.00-45

[10] J. Flinn and M. Satyanarayanan. 1999. PowerScope: a tool for profiling the energy usage of mobile applications. *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications* (1999), 2–10. doi:10.1109/mcsa.1999.749272

[11] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. 2017. Analyzing Programming Languages' Energy Consumption. *Proceedings of the 21st Pan-Hellenic Conference on Informatics* (2017), 1–6. doi:10.1145/3139367.3139418

[12] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning. arXiv:2002.05651 [cs.CY]

[13] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. 2024. EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy. *ACM SIGENERGY Energy Informatics Review* 4, 3 (2024), 18–25. doi:10.1145/3698365.3698369

[14] Mathilde Jay, Vladimir Ostapenco, Laurent Lefevre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. 2023. An experimental comparison of software-based power meters: focus on CPU and GPU. *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* 00 (2023), 106–118. doi:10.1109/ccgrid57682.2023.00020

[15] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26. doi:10.1145/3177754

[16] Colin Ian King. 2017. . https://github.com/ColinIanKing/stress-ng

[17] Jens Malmodin, Nina Lövehagen, Pernilla Bergmark, and Dag Lundén. 2024. ICT sector electricity consumption and greenhouse gas emissions – 2020 outcome. 48, 3 (04 2024), 102701. doi:10.1016/j.telpol.2023.102701

[18] Srinivasan Murali, Martijn Coenen, Andrei Radulescu, Kees Goossens, and Giovanni De Micheli. 2006. Mapping and configuration methods for multi-use-case networks on chips. *Proceedings of the 2006 conference on Asia South Pacific design automation - ASP-DAC '06* (2006), 146–151. doi:10.1145/1118299.1118344

[19] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. 2012. A Preliminary Study of the Impact of Software Engineering on GreenIT. *2012 First International Workshop on Green and Sustainable Software (GREENS)* (2012), 21–27. doi:10.1109/greens.2012.6224251

[20] Benoit Petit. 2023. scaphandre. (2023). https://github.com/hubblo-org/scaphandre

[21] Andrew S. Tanenbaum and Albert S. Woodhull. 2006. *Operating Systems: Design and Implementation* (3rd ed.). Pearson Prentice Hall, Upper Saddle River, New Jersey.

[22] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring Energy and Power with PAPI. *2012 41st International Conference on Parallel Processing Workshops* 1 (2012), 262–268. doi:10.1109/icppw.2012.39

[23] Shijie Yu, Hailong Yang, Rui Wang, Zhongzhi Luan, and Depei Qian. 2017. Evaluating architecture impact on system energy efficiency. *PLoS ONE* 12, 11 (2017), e0188428. doi:10.1371/journal.pone.0188428

[24] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. 2014. HaPPy: Hyperthread-aware Power Profiling Dynamically. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 211–217. https://www.usenix.org/conference/atc14/technical-sessions/presentation/zhai