Towards Modeling the Power Usage Efficiency of Software-Defined Computing Infrastructures

Guillaume Fieni

University of Lille

This dissertation is submitted for the degree of
Doctor of Philosophy in Computer Science

Thesis committee:

Supervisors	Prof. Romain Rouvoy	–	University of Lille
	Prof. Lionel Seinturier	–	University of Lille
Reviewers	Prof. Patricia Stolf	–	University of Toulouse - Jean Jaurès
	Prof. Alain Tchana	–	Grenoble INP
Examiner	Dr. Valerio Schiavoni	–	University of Neuchâtel
Chair	Prof. Giuseppe Lipari	–	University of Lille

Defended on Thursday, December 15, 2022

Vers la Modélisation de l'Efficience Énergétique des Infrastructures de Calcul Virtualisées

Guillaume Fieni

Université de Lille

Thèse présentée pour l'obtention du titre de
Docteur en Informatique

Jury de thèse :

Directeurs de thèse	Prof. Romain Rouvoy	–	Université de Lille
	Prof. Lionel Seinturier	–	Université de Lille
Rapporteurs	Prof. Patricia Stolf	–	Université Toulouse - Jean Jaurès
	Prof. Alain Tchana	–	Grenoble INP
Examinateur	Dr. Valerio Schiavoni	–	Université de Neuchâtel
Président	Prof. Giuseppe Lipari	–	Université de Lille

Défendue le Jeudi 15 Décembre 2022

Acknowledgements

I would like to thank the people who contributed to the realization of this thesis.

Foremost, I would like to thank my thesis supervisors Romain Rouvoy and Lionel Seinturier for giving me this opportunity, as well as for their trust and support throughout this adventure. This thesis allowed me to work on interesting subjects and with great autonomy, for which I am very grateful. I would particularly like to thank Romain Rouvoy for his precious help, his numerous remarks and suggestions, as well as his kindness and patience.

I would also like to thank the members of my thesis committee for having accepted to devote their time to the evaluation of my work and for their remarks and suggestions. In particular, I would like to thank the reviewers, Patricia Stolf and Alain Tchana, for the review of this manuscript and for the richness of their reports which allowed me to improve it.

I would like to thank the members of the Spirals team for their kindness, and their willingness to share their knowledge. Thanks also to the former members of the team such as Maxime Colmant, Antoine Vastel, Walter Rudametkin and Antonin Durey, for their kindness and the numerous exchanges.

To conclude, I would like to thank my parents, who have always trusted me, supported me and encouraged me in what I do, especially during these long years of study. As well as my friends, on whom I could always count and who always supported me.

Remerciements

Je souhaite remercier l'ensemble des personnes ayant contribué à la réalisation de cette thèse.

Dans un premier temps, je remercie mes directeurs de thèse Romain Rouvoy et Lionel Seinturier pour m'avoir donné cette opportunité, ainsi que pour leur confiance et soutien tout au long de cette aventure. Cette thèse m'aura permis de travailler sur des sujets intéressants et avec une grande autonomie, ce dont je vous suis très reconnaissant. Je tiens tout particulièrement à remercier Romain Rouvoy pour son aide précieuse, ses nombreuses remarques et suggestions, ainsi que de sa bienveillance et sa patience.

Je souhaite également remercier les membres de mon jury de thèse d'avoir accepté de consacrer leur temps à l'évaluation de mes travaux ainsi que pour leurs remarques et suggestions. Je tiens à remercier en particulier les rapporteurs, Patricia Stolf et Alain Tchana, d'avoir accepté de relire ce manuscrit ainsi que pour la richesse de leurs rapports qui m'a permis de l'améliorer.

Je voudrais ensuite remercier les membres de l'équipe-projet Spirals pour leur gentillesse, bonne humeur au quotidien, ainsi que de leur volonté de partager leurs connaissances. Merci également aux anciens membres de l'équipe telle que Maxime Colmant, Antoine Vastel, Walter Rudametkin et Antonin Durey, pour leur bienveillance et les nombreux échanges.

Pour conclure, je tiens à remercier mes parents, qui m'ont toujours accordé leur confiance, soutenu et encouragé dans ce que je fais, surtout durant ces longues années d'études. Ainsi que mes amis, sur qui j'ai toujours pu compter et qui m'ont toujours apporté leur soutien.

Abstract

Energy is one of the biggest expenses for a data center, most of which is attributed to the cooling system, as well as the many underlying parts, such as network equipment and the large number of machines used. These infrastructures are very energy-intensive, and their number is constantly increasing around the world, especially due to the growing popularity of the Cloud Computing.

A lot of software is needed to run these infrastructures, especially for network management, data storage, task scheduling and the supervision of all hardware and software. All these software consumes a significant amount of energy, but are not taken into account in the calculation of the energy efficiency of the infrastructures. The scientific community as well as data center operators have developed many approaches to evaluate and optimize energy consumption globally, but the question of the energy cost of software infrastructures remains rarely studied.

The objective of this thesis is to propose methods to analyze the end-to-end software energy efficiency of data processing infrastructures. To do so, we propose approaches and tools to accurately estimate the energy consumption of software running on a distributed infrastructure, as well as an indicator to calculate their energy efficiency.

Firstly, we introduce SmartWatts, a software power meter to estimate the energy consumption of software containers deployed on a machine. Secondly, we propose SelfWatts, a controller to automate the configuration of software power meters to facilitate their deployment in heterogeneous infrastructures. And finally, we propose xPUE, a metric to calculate the energy efficiency of software and hardware in real-time at different levels of an infrastructure.

Through these contributions, we aim to advance the knowledge in the field of software energy consumption, and allow to accurately measure the energy consumption of software deployed at different levels of the infrastructure. This allows infrastructure operators, as well as software developers and users, to observe and analyze in detail the energy consumption and thus assist in its optimization.

Résumé

L'Énergie représente l'un des principaux postes de dépense pour un centre de données, dont la majeure partie est attribuée au système de refroidissement, ainsi qu'aux nombreuses parties sous-jacentes, comme les équipements réseau et au grand nombre de machines utilisées. Ces infrastructures sont très énergivores, et leur nombre ne cesse d'augmenter à travers le monde, notamment grâce à la popularité croissante du Cloud Computing.

De nombreux logiciels sont nécessaires au bon fonctionnement de ces infrastructures, notamment pour la gestion du réseau, du stockage de données, de l'ordonnancement des tâches ainsi que de la supervison de l'ensemble du matériel et des logiciels. Tous ces logiciels consomment une quantité significative d'énergie, mais ne sont pourtant pas pris en compte dans les calculs de l'efficience énergétique des infrastructures. La communauté scientifique ainsi que les opérateurs de centres de données ont développé de nombreuses approches afin d'évaluer et d'optimiser globalement la consommation énergétique, mais la question du coût en énergie des infrastructures logicielles reste peu étudiée.

L'objectif de cette thèse est de proposer des méthodes permettant d'analyser de bout en bout l'efficacité énergétique logicielle des infrastructures de traitement de données. Pour cela, nous proposons des approches et outils permettant d'estimer fidèlement la consommation énergétique des logiciels exécutés sur une infrastructure distribuée, ainsi qu'un indicateur permettant de calculer leur efficience énergétique.

Dans un premier temps, nous proposons SmartWatts, un wattmètre logiciel permettant d'estimer la consommation énergétique des conteneurs logiciels déployés sur une machine. Ensuite, nous proposons SelfWatts, un contrôleur permettant d'automatiser la configuration des wattmètres logiciels afin de faciliter leur déploiement dans des infrastructures hétérogènes. Et enfin, nous proposons le xPUE, un indicateur permettant de calculer l'efficience énergétique des logiciels et du matériel en temps réel aux différents niveaux d'une infrastructure.

À travers ces différentes contributions, nous visons à faire évoluer la connaissance dans le domaine de la consommation énergétique des logiciels, et permettre de mesurer avec précision la consommation énergétique des logiciels déployés aux différents niveaux des infrastructures. Cela permet aux opérateurs de ces infrastructures, mais également aux développeurs et utilisateurs de logiciels d'observer et d'analyser en détails la consommation énergétique et ainsi d'assister dans l'optimisation de celle-ci.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Computing infrastructures are a massive energy consumption hotspot due to their underlying nature and popularity across the world. Being particularly energy-intensive, the data center industry accounts for around 0.9–1.3% of global electricity consumption and nearly 1% of global energy-related greenhouse gas emissions [31]. Computing infrastructure providers, such as on-demand cloud computing platforms, aim at continuously improving the energy efficiency of their infrastructures in order to reduce the operating costs. This trend, combined with environmental concerns, makes energy efficiency a prime technological and societal challenge.

While trying to optimize the energy consumption of computing infrastructures, one needs to have the means to precisely assess its efficiency. Researchers and infrastructure operators have been proposing solutions to increase energy efficiency at all levels, from the application down to the underlying hardware. As surveyed by Orgerie *et al.*, examples of such methods include energy-based task scheduling, energy-efficient software, dynamic frequency and voltage scaling, and energy-aware workload consolidation using virtualization [43].

The massive increase of cloud infrastructures raises a key challenge to operate at scale, while being energy-efficient. Cloud infrastructures are mostly based on virtual machines and software containers, where the underlying hardware is virtualized and the execution confined from other VMs, and/or software containers where only a light resources separation is applied. Such environments have various machine configurations to cope with a wide range of use cases, such as generic compute, memory, networks and disks intensive workloads, to more specific machine learning problems that requires dedicated hardware.

This thesis aims to propose new solutions to observe the energy consumption of modern computing infrastructures by proposing tools that allow an end-to-end analysis of the energy consumption of the various parts of the infrastructure. To do so, first we propose a lightweight power monitoring system that adopts online calibration to automatically adjust the CPU and DRAM power models by using the activity of the machine to maximize the accuracy of runtime power estimations of software containers. Beyond the power model, we offer a framework to foster the research on green computing and to offer tools to measure the energy consumption of various parts of the infrastructure, and helps building software-defined power meters. This solution can be used to evaluate the energy consumption of an infrastructure based on software containers, but also to manage a power budget to cap the energy consumed from users to applications and down to individual instances [18].

Then, we extend our previous work to propose a lightweight power monitoring system that explores and selects the relevant performance events to automatically optimize the power models to the underlying architecture. The deployment of software power meters in heterogenous environments is challenging as the specificities of the underlying hardware can drastically change between nodes. As time passes, hardware evolves to be more performant, gains features advantageous for specific workloads (vectors, cryptography) and becomes more energy-efficient. This leads to previously accurate power models to become either obsolete or inaccurate by not taking into account changes in the micro-architecture, such as the energy optimization mechanisms, for example. To solve this problem, we propose a self-adaptive power meter that automatically explores and chooses the relevant performance events for the machine hardware at runtime. This allows to further simplify the deployment of software power meter at scale and in heterogenous environments without *a priori* knowledge about the underlying hardware.

Finally, we propose an energy efficiency metric that allows to measure the efficiency from various levels of the computing infrastructures. Various energy efficiency metrics exists to evaluate the efficiency of data centers, and some of them are part of standards such as the *Power Usage Effectiveness* (PUE) [22] and the *Data Center Infrastructure Efficiency* (DCiE) [19]. However, these metrics only show the general efficiency of the infrastructures and do not cover the efficiency of the underlying hardware and software needed to make them work. This contribution aims to provide a multi-level energy efficiency metric that allows data center operators and users to analyze in depth the energy hotspots of the software running on their infrastructures.

The following chapter is organized as follows. Section 1.1 introduces the problem statements extracted from the state-of-the-art. Section 1.2 exposes the goals of this thesis and shows how our contributions bring new solutions to face these problems. Section 1.3 reports on all the papers (published or under submission) with the Proofs-of-Concept contributed as part of this thesis. And finally, Section 1.4 summarizes the content of this thesis.

1.1 Problem Statement

The research community showed a high interest about the energy efficiency of the hardware and software running in computing infrastructures. To optimize the energy efficiency of such infrastructures, one must be able to discover and analyze its various energy hotspots.

Most cloud computing and data-center providers, such as *Google Cloud Platform* (GCP) [25], for example, publish detailed information about the efficiency of theirs data centers each year/quarter. These infrastructures are built with efficiency in mind and leverage various optimization methods, such as building data centers in countries where the air is cool to limit the usage of the *Air Conditioning* (AC), using advanced cooling methods, such as water cooling for the servers, and consuming renewable energy to greatly lower their energy prices and CO_2 emissions footprint. Unfortunately, such metrics only scratches the surface of the energy efficiency measurements, and do not allow stakeholders to obtain a precise insight of the energy hotspots in the hardware and the software hosted in the infrastructures. To this extent, we choose to focus on providing approaches enabling an in-depth analysis of the energy consumption for the various hardware and software layers composing a modern computing infrastructure.

Power monitoring is usually achieved by hardware measurement equipments, such as Wattmeters, *Power Supply* (PSU) and *Power Distribution Units* (PDU), which only report power measurements at a coarse-grained level. However, one needs to monitor at a finer level to optimize the energy of the whole system by targeting the applications that are running on it. Several solutions have been proposed over the years to estimate the power consumption of the software running on a machine, such as software power models [13, 37]. Unfortunately, these approaches require *a priori* learning steps that hinder their deployments at scale and in heterogeneous environments, in addition to be extremely fragile to workload changes and non-portable across machines.

Consequently, such environments complicate the deployment at scale of software power meters, mainly because of the required knowledge about the machine hardware to manually tune the power models. For example, most of the CPU power models rely on a thorough expertise of the targeted architectures, thus leading to the design of hardware-specific solutions that can hardly be ported beyond the initial settings. State-of-the-art solutions are facing several key limitations and are often limited to specific hardware components and are mostly offline based, which complicates even more theirs usage for heterogenous infrastructures.

The power estimation of software processes provides critical indicators to drive scheduling policies [27], power capping heuristics [18] or even improve the performance of workloads [53]. Combined with existing hardware power measurements, software power models introduce another dimension in the evaluation of the energy efficiency of computing infrastructures. Software power measurements are the missing link in the chain of the energy efficiency indicators of computing infrastructures, and the state-of-the-art currently fails to provide end-to-end approaches to this problem.

Therefore, the objective of this thesis is to propose an end-to-end approach to evaluate the energy efficiency of the underlying software running on heterogeneous computing infrastructures. To this effect, we focus on the following research problems and propose practical solutions throughout various contributions.

1.1.1 Estimating the energy consumption of software containers

Nowadays, most of the computing infrastructures are built around the concept of software containers, such as Docker/Podman and Kubernetes. These containers are used to isolate the software processes from the underlying hardware and to provide a portable and reproducible environment to run the software. However, the energy consumption of the software running in such containers is not easily observable, especially when they are treated as black box components by the platform provider.

In the state of the art, most software power models require a calibration phase to estimate the CPU energy consumption of the software processes. This phase is usually done offline by using synthetic benchmarks to generate workload data, and can last several hours to multiple days depending of amount of samples needed by the power models. We believe that such approaches are not only wasteful in resources, time and energy, but also not suitable for the deployment of power models at scale.

Hence, a solution that allows to estimate the CPU and DRAM energy consumption of software containers at runtime, without requiring any calibration phase could allow users and developers to easily monitor the energy consumption of their software containers.

1.1.2 Optimizing the power models for heterogenous environments

As stated in Section 1.1.1, power models are useful to estimate the energy consumption of software processes. However, their configuration is usually written manually by the operator or user, which requires a thorough knowledge of the underlying hardware. Over the time, the underlying hardware evolves, and the topology of the infrastructure changes, thus requiring to re-tune the power models to keep them working with the new hardware. In some cases, using the previous configuration of the power models on new hardware can even lead to wrong estimations of the energy consumption of the software, due to major changes in the underlying architecture.

Power models need performance events in order to accurately estimate the energy consumption. Performance events are deeply linked to the underlying hardware and requires context specific attention. In the state of the art, power models have an offline benchmark phase where the performance events for the CPU of the machine are explored, and where the correlation coefficient of the various events is used to select the most relevant [13, 37]. This allows to select the most relevant performance events for the software power models used to monitor the power consumption of the software processes.

Hence, a solution that allows to automatically configure software power meters, would allow to automatically adapt to the underlying hardware, and to avoid the need of manual tuning while easing the deployment of power meters at scale and in heterogenous environments.

1.1.3 Evaluating the software energy efficiency of an infrastructure

Modern data centers go to great length to limit the energy consumption of their services, by optimizing the placement of the different services, choosing equipment that consume less electricity, or by fine-tuning how the software interact with the hardware, in addition to using environmental enhancement building methods. Various energy efficiency measurements have been proposed or are being investigated to fully grasp the energy efficiency of the infrastructures. The most used indicator is the *Power Usage Effectiveness* (PUE) [22], which is the ratio of the total amount of energy used by a computer data center facility to the energy delivered to computing equipment. While this metric widely used, it lacks of granularity needed to evaluate the underlying software energy of the computing infrastructure.

In the state of the- art, the proposed energy efficiency metrics only allow to evaluate parts of the data centers, but do not dive much deeper than the machine's power supply [22, 19, 20]. The lack of solutions to evaluate the energy efficiency of the software running on the infrastructures prevents the analysis of the architecture's hotspots, and is a major limitation to the deployment of energy efficient infrastructures.

Hence, an indicator that incorporates the energy efficiency of the underlying software layers of the infrastructure will provide a new dimension to analyze and optimize the energy efficiency of computing infrastructures.

1.2 Contributions

To address the aforementioned problems, we propose in this thesis multiple approaches that contribute to the research about the evaluation of the energy efficiency of the software running on heterogeneous computing infrastructures. We summarize these contributions in the following subsections.

1.2.1 Self-Adaptive Software Power Models

As explained in Section 1.1.1, existing software power models rely on specific hardware or need an offline calibration phase to train the power models. This leads to a lack of portability and scalability of the power models, and complicates the deployment of power meters in infrastructures. The lacks of support of the various energy efficiency features of the CPU lead to inaccurate power models, especially when the machine is idle, or when the workload is not CPU bound.

For the first contribution of this thesis, we focus on the energy consumption of software containers running on modern computing infrastructures. The main goal of this contribution is to propose a lightweight power monitoring system that allows to estimate in real-time the energy consumption of software containers, helping to better understand the energy consumption of software and discover and plan possible optimizations.

To do so, we propose a novel approach to estimate the CPU and DRAM energy consumption of software containers, based on the monitoring of the performance events of the CPU. This contribution provides a self-adaptive power model that provides real-time power estimations for the running software containers by automatically recalibrating the power models when an error threshold is exceeded. The power models are based on the *Hardware Performance Counters* (HwPC) for the CPU activity of the software containers, and the *Running Average Power Limit* (RAPL) interface for the CPU's energy consumption measurements. The models are automatically calibrated over the time, by using the activity and inactivity of the machine, without interfering with the normal operation of the software containers. This allows to provide accurate power estimations for the running software containers in real-time, which can then be used to analyze the energy consumption and/or integrate such measurements for other usage in the infrastructure.

1.2.2 Self-Optimizing Software Power Models

As depicted in Section 1.1.2, existing software power models are not easily deployable in heterogeneous environments, and require a manual tuning to be able to work with new hardware. Software power models rely on the hardware performance counters to account the hardware activity, and needs a set of performance events related to the activity in order to accurately estimate the energy consumption. Most contributions that rely on such power models select the performance events by various methods, such as manually by an expert, semi-automatically by filtering the events *a priori*, or letting an automated process to select the relevant events.

For the second contribution of this thesis, we focus on the automation of the performance events selection at runtime for software power models. The main goal of this contribution is to completely automate the configuration of software power meters and allows their unsupervised deployment in heterogenous infrastructures.

To do so, we propose a novel approach that automatically selects the performance events for the power models of the running software containers. This contribution provides a self-calibrating software-defined power meter that performs an online performance events exploration and selection, and automatically adapts to the underlying hardware. The goal of this contribution is to provide a fully plug-and-play software power meter that provides power estimations for the running software containers and can be deployed on a heterogeneous infrastructure.

1.2.3 Extending the Power Usage Effectiveness for Cloud Infrastructures

As stated in Section 1.1.3, the *Power Usage Effectiveness* (PUE) is the most used metric to evaluate the energy efficiency of computing infrastructures. However, this metric only allows to evaluate the energy efficiency at a coarse grain, and does not help in assessing the energy efficiency of the software running on the infrastructure. Other metrics have been proposed to evaluate the energy efficiency of the infrastructures, but they are not widely used, and do not take into account the software energy efficiency of the infrastructure either.

For the third contribution of this thesis, we focus on the in-depth analysis of the energy efficiency of the software running on computing infrastructures. The main goal of this contribution is to provide an energy efficiency indicator that allows to evaluate the efficiency of the software running on the infrastructure.

To do so, we propose a novel approach that extends the PUE metric to the software running on the computing infrastructure. This contribution provides an in-depth analysis of the energy efficiency of the software of the various levels of the infrastructure. The goal of this contribution is to provide a new metric that allows to evaluate the energy efficiency of the software running on the infrastructure, while remaining flexible enough to fit logical and physical infrastructures.

1.3 Publications

In this section, we summarize the scientific contributions that have been published in conferences or still under evaluation.

1.3.1 Published

The following papers have been published in International Conferences and Workshops:

1. Guillaume Fieni, Romain Rouvoy, Lionel Seinturier, *"SelfWatts: On-the-fly Selection of Performance Events to Optimize Software-defined Power Meters,"* In proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021.

2. Guillaume Fieni, Romain Rouvoy, Lionel Seinturier, *"SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers,"* In proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 2020.

3. Jonatan Enes, Guillaume Fieni, Roberto R. Expósito, Romain Rouvoy, Juan Touriño, *"Power Budgeting of Big Data Applications in Container-based Clusters,"* In proceedings of the 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020.

4. Guillaume Fieni, *"Towards Sustainable Software Infrastructures for Data-Intensive Systems,"* In proceedings of the 12th EuroSys Doctoral Workshop (EuroDW), 2018.

1.3.2 Future Submission

The following papers are still ongoing work and aims to be published in International Conferences or Journals:

1. Guillaume Fieni, Romain Rouvoy, Lionel Seinturier: *"xPUE: Extending Power Usage Effectiveness Metrics for Cloud Infrastructures,"* 2023.

1.3.3 Proof of Concept

The various contributions proposed in this thesis needed implementations in order to be evaluated empirically. Some software is mature enough to be used in production environments, and some industrial partners are actively using it. This shows strong evidence on the relevance and usability of the contributions of this thesis. Following, a summary of the various software and tools that was developed and are open-source and available on GitHub.

PowerAPI is a middleware toolkit for building software-defined power meters that can estimate the power consumption of software in real-time. PowerAPI supports the acquisition of raw metrics from a wide diversity of sensors (eg., physical meters, processor interfaces, hardware counters, OS counters) and the delivery of power consumptions via different channels (including file system, network, web, graphical). As a middleware toolkit, PowerAPI offers the capability of assembling power meters «*à la carte*» to accommodate user requirements.[1]

HwPC-Sensor is a software agent for monitoring Hardware Performance Counters of cgroups based software containers on Linux. It allows to periodically collect a set of Hardware Performance Counters values for the running software containers of a machine. Various output formats are supported such as CSV files, MongoDB and Socket in order to send the measurements to a processing pipeline. This tool allows to use highly precise hardware-related metrics for the software power meters and is the base of the contributions in Chapter 3 and Chapter 4.[2]

SmartWatts Formula is the implementation of the SmartWatts contribution depicted in Chapter 3. SmartWatts is a formula for a self-adaptive software-defined power meter based on the PowerAPI framework. [3]

SelfWatts Controller is a wrapper that allows to remotely control the HwPC-Sensor monitored events. It is made to be used by the SelfWatts Formula to explore the available Hardware Performance Events of the CPU in order to construct a robust power model over time.[4]

SelfWatts Formula is the implementation of the SelfWatts contribution. SelfWatts is a formula for a software-defined power meter that explores and select the relevant performance events at runtime.[5]

[1]https://github.com/powerapi-ng/powerapi
[2]https://github.com/powerapi-ng/hwpc-sensor
[3]https://github.com/powerapi-ng/smartwatts-formula
[4]https://github.com/powerapi-ng/selfwatts-controller
[5]https://github.com/powerapi-ng/selfwatts-formula

1.4 Outline

The remainder of this thesis is organized as follows.

Chapter 2 introduces the context and the most relevant works about the topic of this thesis. This includes works about hardware and software power models and theirs optimizations for the underlying hardware and workload, then we follow with the energy efficiency metrics and their usages in a green computing context.

Chapter 3 covers SmartWatts, a self-adaptive power model for software containers based on the Hardware Performance Counters of the CPUs for the activity, and the *Running Average Power Limit (RAPL)* interface for the CPU energy measurements.

Chapter 4 presents SelfWatts, a self-calibrating power model that allows to automatically configure software power meters in heterogeneous environments.

Chapter 5 reports on *x*PUE, an extension of the *Power Usage Effectiveness* (PUE) indicator to evaluate the software energy efficiency of the various levels of the computing infrastructures.

Finally, **Chapter 6** concludes this thesis by summarizing the contributions and proposing possible research directions.

Chapter 2

State-of-the-Art

While the field of the energy efficiency of data centers is quite mature, the research on the energy efficiency of software systems is still lacking in-depth contributions, especially with the evolutions of the hardware and software infrastructures. In particular, the works that are related to the energy efficiency of software systems are still limited to the energy efficiency of the hardware components, such as the CPU and the DRAM, and lack the software dimension needed to provide a holistic view of the energy consumption of the complete infrastructure.

The remainder of this chapter will focus on the following contributions. Section 2.1 introduces the state of the art in software-defined power meters, which are in charge of estimating the power consumption from the hardware down to the software services. Section 2.2 presents the works that are related to the performance events selection and other calibration methods which are in charge of adjusting the power models of the software power meters to the underlying hardware. Section 2.3 concludes this chapter by covering the various energy efficiency metrics that are used to evaluate computing infrastructures.

2.1 Power estimation

2.1.1 Hardware Power Meters

Over the years, hardware power meters have evolved to deliver hardware-level power measurements with different levels of granularity, from physical machines to specific electronic components.

POWERMON2 [5] uses an external power monitoring board to be inserted between a system's power supply and a motherboard in order to retrieve the power consumption per connected hardware component. It can measure up to 8 DC rails and can read and report power measurements at a rate up to 3 KHz. While all schematics and source code are freely available online, the hardware requirements of this solution cost up to $150 per-machine.

Icsi *et al.* [29] describe an approach for learning CPU power models based on predefined 15 *Harwdare Performance Counters* (HwPC) for 22 selected processor subunits. In addition, they propose a live CPU power monitoring solution that implies different modules. First, a reader runs inside the system under test for collecting values for the selected HwPC. Once collected, the values are sent via the network to a logger machine which uses the power model and the extracted values for producing live power estimation of the 22 processor subunits. With this approach, the authors show runtime power estimations for one concurrent running application that can be divided per involved subunit.

PowerPack [23] monitors all hardware components separately. To retrieve power measurements, a precision sensing resistor is attached to each DC power line, thus allowing to measure the voltage differences with a power meter. Power measurements are simultaneously collected on all power lines to be representative of all hardware components. The data retrieved are then recorded and used in a *post mortem* analysis. The authors consider their approach as being able to deliver per-process power estimation, but they only consider a single concurrent application during their validation. They also mention that it can target a cluster, but only one node can be monitored at a time, which makes it unsuitable in production.

PowerInsight [36] follows the same principle as PowerMon2 and is built on top of another external board that uses an ARM Cortex processor. This external board can be connected up to 15 components and is used to acquire power measurements from custom power sensing boards connected to it. Each board is then connected through Ethernet and report the measurements to a master node.

WattProf [46] power monitoring platform supports the profiling of *High Performance Computing* (HPC) applications. This solution is based on a custom board, which can collect raw power measurements from various hardware components (CPU, disk, memory, etc.) from sensors connected to power lines. The board can connect up to 128 sensors that can be sampled at up to $12KHz$. The data can be thus be retrieved via an Ethernet interface, or be buffered inside the board for *postmortem* analysis. As in [23], the authors argue that this solution is able to perform per-process power estimation, but they only validate their approach while running a single application.

WattWatcher [37] is a tool that can characterize workload energy power consumption. The authors use several calibration phases to build a power model that fits a CPU architecture. This power model uses a predefined set of *Hardware Performance Counters* (HwPC) as input parameters. As the authors use a special power model generator that can target any

CPU architecture, this model still has to be carefully calibrated to take into account the hardware energy optimization mechanisms, such as P-states and C-states. An efficient network connection is required to send data to the generator to monitor the power estimation in real-time.

RAPL [50] offers specific *hardware performance counters* (HwPC) to report on the energy consumption of the CPU since the "Sandy Bridge" micro-architecture for Intel (2011) and "Zen" for AMD (2017). Intel divides the system into domains (PP0, PP1, PKG, DRAM) that report the energy consumption according to the requested context. The PP0 domain represents the core activity of the processor (cores + L1 + L2 + L3), the PP1 domain the uncore activities (LLC, integrated graphic cards, etc.), and PKG represents the sum of PP0 and PP1, and the DRAM domain exhibits the DRAM energy consumption. Desrochers *et al.* demonstrate the accuracy of the DRAM power estimations of RAPL, especially on Intel Xeon processors [16].

2.1.2 Software-Defined Power Meters

To get rid of the hardware cost imposed by the above solutions, the design of power models has been regularly considered by the research community over the last decade, in particular for CPU [6, 12, 32, 41, 56]. Notably, as most architectures do not provide fine-grained power measurement capabilities, McCullough *et al.* [41] argue that power models are the first step towards enabling dynamic power management for power proportionality at all levels of a system.

While standard operating system metrics (CPU, memory, disk, or network), directly computed by the kernel, tend to exhibit a large error rate due to their lack of precision [32, 56], HwPC can be directly gathered from the processor (*e.g.*, number of retired instructions, cache misses, non-halted cycles). Modern processors provide a variable number of HwPC events, depending on the generation of the micro-architectures and the model of the CPU. As shown by Bellosa [6] and Bircher [8], some HwPC events are highly correlated with the processor power consumption, while the authors in [48] concluded that not all HPC are relevant, as they may not be directly correlated with dynamic power.

Power modeling often builds on these raw metrics to apply learning techniques [7] to correlate the metrics with hardware power measurements using various regression models, which are so far mostly linear [41]. Three key components are commonly considered to train a power model: *a)* the workload(s) to run during sampling, *b)* the minimal set of input parameters, and *c)* the class of regression to use [7, 60, 59, 17].

The workloads used along the training phase have to be carefully selected to capture the targeted system. In this domain, many benchmarks have been considered, but they are mostly *a)* designed for a given architecture [7, 29], *b)* manually selected [9, 12, 14, 17, 38, 58–60], or even *c)* private [60]. Unfortunately, this often leads to the design of power models that are tailored to a given processor architecture and manually tuned (for a limited set of power-aware features) [7, 9, 29, 38, 39, 58, 60, 52].

2.1.3 Limitations & Opportunities

To the best of our knowledge, the state of the art in hardware power meters often imposes hardware investments to provide power measurements with a high accuracy, but a coarse granularity, while software-defined power meters target fine-grained power monitoring, but often fail to reach high accuracy on any architecture and/or workload. Most of the existing solutions are limited to hardware components and require hardware investments to be fully operational [5, 54], thus this kind of tools cannot be then used while targeting fine-grained power estimations. To overcome these limitations, several tools propose to target software power consumption, but a few of them implies costly investments and/or are not usable in practice [23, 37, 46]. Moreover, only a few contributions discuss the overhead of their approach [42] and a lot of research efforts remain to be done.

So far, the state of the art fails to deploy software-defined power meters in production because *i)* the model learning phase can last from minutes to days, *ii)* the power models are often bound to a specific context of execution that do not take into account hardware energy-optimization states, and *iii)* the reference power measurement requires specific hardware to be installed on a large amount of nodes. This therefore calls for methods that can automatically adapt to the hardware and workload diversities of heterogenous environments in order to maintain the accuracy of power measurements at scale.

2.2 Optimization of software power models

2.2.1 Power Model Calibration Methods

Hardware and software power meters keep evolving to deliver hardware-level power measurements with different levels of granularity, from physical machines to electronic components and running software.

KOALA [53] introduce a platform that uses a pre-characterized model at run-time to predict the performance and energy consumption of a piece of software. The generic model is calibrated using a set of benchmarks, during a calibration process which is performed offline. Then, the model is re-calibrated at runtime in order to correct the coefficients and increase the accuracy of the model. However, this method still requires a calibration phase, and the online recalibration does not work if the machine differs too much from the one used during the calibration phase.

RAPL [50] exposes additional *hardware performance counters* (HwPC) to report on the energy consumption of the CPU since the “Sandy Bridge“ micro-architecture for Intel (2011) and “Zen“ for AMD (2017). RAPL divides the system into domains (PP0, PP1, PKG, DRAM) that report the energy consumption according to the requested components. As stated by the authors, RAPL include a set of architectural power meters that does not require any calibration from the user, as it is setup at factory level, where physical properties like the leakage information is coded into the die. The models use a set of architectural events from each cores, GPU, and I/O, and combines them with energy weights, which are scaled with operating conditions such as voltage and temperature, to predict the components active power consumption. Thus, even if RAPL does not capture the software-level energy consumption, we believe it offers a relevant ground truth for modeling the power consumption at the scale of the processor.

POWER CONTAINERS [52] proposes to account for and control the power and energy usage of individual requests in multicore servers. However, the deployment of power containers requires to pre-calibrate the power model with offline samples and then recalibrate these power models with online context samples. This implies that several micro-benchmarks require to be executed to infer the coefficients of the power model, thus imposing a long delay that prevents it to be deployed in production when multiple generations and models of machines are available.

BITWATTS [12] is a monitoring middleware providing real-time power estimations of software processes running at any level of virtualization in a system. BITWATTS includes a power model that computes a polynomial regression for each frequency supported by the CPU (including Turbo Boost frequencies). BITWATTS requires a physical power meter (PowerSpy) with a manual calibration phase to benchmark every frequency supported by the CPU. Unfortunately, this procedure also prevents BITWATTS to be deployed on a wide panel of hardware architectures. WATTSKIT [11], which is an extension of BITWATTS, supports the power monitoring of distributed services, but suffers from the same limitation when it comes to the effective deployment of the solution in a cluster of heterogeneous machines.

WATTWATCHER [37] is a tool that can characterize the energy consumption of a workload. To do so, the authors combine several calibration phases to train a power model that fits a CPU architecture. The authors propose a special power model generator that can target any CPU architecture, but requires to be carefully described *a priori*. Unfortunately, this power model uses a predefined set of HwPC events as input parameters, which may not be available on some CPU architectures, thus preventing the exploitation of the generated power model.

2.2.2 Feature Selection for Software Power Models

As mentioned above, power models are learned from raw metrics that are expected to relate the activity of the hardware components energy consumption. Modern CPUs provide several *Performance Monitoring Units* (PMU), which implement a limited number of HwPC slots that can be used to monitor a large number of performance events. In the literature, the selection of the relevant performance events that feed a power model is mostly achieved offline, thus requiring a calibration phase during which a target machine has to execute several workloads over a long period to identify such relevant raw metrics.

To select the key metrics, the literature usually builds on *Pearson* or *Spearman* correlation [53, 13, 21] and *Principal Component Analysis* (PCA) [59], which often leads to consider performance events like *unhalted core cycles*, *unhalted ref cycles*, *instructions retired*, *llc misses/prefetch*, or *memory transactions cycles* [52, 12, 37, 13, 21]. Yet, this selection phase requires to be executed on every single target architecture to make sure that the relevant performance events are made available, which inevitably impacts the cost and the scale of the deployment of software-defined power meters. Furthermore, all these *a priori* calibration phases share the same limitation: the selected performance events highly depend on the nature of the calibration workload, which may strongly differ from the workload monitored in production and thus question the accuracy of the resulting power model.

2.2.3 Hardware Power Optimizations

This contextual issue is particularly challenging to capture as modern processors embed several mechanisms that are autonomously triggered in order to optimize the idle power consumption and the performance of the host machine upon context. For example, Intel CPUs are currently implementing the following power-aware hardware features:

P-states are *performance power states*, where each state specifies the voltage and the clock frequency at which the CPU operates. This allows the CPU to maintain performance objectives while minimizing power consumption. The operating system picks the most suitable state, according to the current usage of the processor by the running workload. The number of supported P-states depends on the micro-architecture and the model of the CPU. Currently, the highest state is `P0` when the CPU operates at the highest voltage and frequency, leading to an increase in performance along with the dissipated heat.

C-states are *idle power states*, also known as core C-states (CC-states), package C-states (PC-states) and logical C-states, specify parts of the CPU that can be powered down to reduce the power consumption depending on usage and the latency cost imposed by power state transitions. The highest state is `C0` when the CPU is fully operational, and the lowest is `C8` when the CPU is inactive and its state saved to LLC before its power cut-off by the power gate transistors. The number of supported C-states also depends on the micro-architecture and model of the CPU. While the operating system suggests to the *Power Control Unit* (PCU) a target state for each core based on the current load of the machine, the PCU autonomously decides the most suitable core and package C-state to optimize the power consumption of the CPU.

Turbo Boost feature allows the CPU to run one or many cores to higher P-states than usual, leading to an increase in performance for a short period. However, this feature becomes only available when the CPU is operating below its rated maximum temperature, current, and power limits. This mode leverages the C-states as the frequency depends on the number of active cores—*i.e.*, the more cores in idle states the higher frequency of active cores. Typically, turbo boost becomes active when the system requests a transition to the P-state `P0`, the operating conditions are below certain model-specific limits and the workload demands more performance. Several turbo frequencies are available depending of the model of the CPU and the current workload. Additionally, while standard workloads can use all turbo frequencies, the AVX2 and AVX512 workloads have dedicated turbo frequencies.

CKE-states are memory power-down modes that allow DIMM ranks to be powered off dynamically when unused. These states are linked to the package C-states where the deepest states allow the memory to enter the self-refresh mode to greatly reduce its power consumption. An IDLE counter is available for each memory rank and determines its CKE mode to maximize the opportunities to power-off unused ranks even under memory-intensive workloads. The *integrated Memory Controller* (iMC) can autonomously power down the DIMM ranks to save energy at the cost of more latency when it will be woken up.

When these mechanisms are triggered by the CPU or the DRAM components, some HwPC events are no longer correlated with the power consumption. In the literature, some power models are *Dynamic voltage and frequency scaling* (DVFS) aware, including the *Turbo Boost*—*i.e.*, P-states effects on the power consumption of the CPU [12]. However, to the best of our knowledge, none of the existing power models take into account the power states in their power models, which leads to incorrect power estimations, when the CPU enters such states, thus preventing their adoption at scale. This limitation is mainly due to the emergence of new energy optimizations, such as the migration from the legacy generic ACPI CPU performance scaling driver to processor-driven ones, such as Intel P-states which can even offload the P-states selection responsibility to the hardware for the most recent CPUs, being known as *Hardware P-states* (HWP). While such behavior can enhance the power efficiency and the performance of the CPU, the lack of fine-grained control of these states prevents the adoption of most of the available power estimation methods, which rely on an *a priori* calibration phase to build a static power model from the execution traces of a given workload run under variable frequencies.

2.2.4 Limitations & Opportunities

In the state of the art, most contributions require offline calibration and previous knowledge of relevant Performance Events for the power model which makes impractical the deployment in a highly heterogeneous infrastructure, like the Cloud. Because of the lack of documentation of performance events, and the constraints imposed by the HwPC slots, the automatic selection of relevant performance events remains an open challenge to support more adaptive power models that can adjust to changes in the workload or the context and keep optimizing the accuracy of power estimations.

2.3 Energy Efficiency Metrics

2.3.1 Data Center Efficiency

Modern *Data Centers* (DC) are continuously trying to maximize the efficiency of their infrastructure to reduce their operating costs. Various metrics have been proposed and standardized to measure the efficiency of the numerous components of computing infrastructures at different levels of granularity.

PUE & DCiE The standard ISO/IEC 30134-2:2016 [22] defines the *Power Usage Effectiveness* (PUE) as a metrics reflecting the energy efficiency of a *Data Center* (DC). More specifically, the PUE is defined as the ratio of energy at the input of the DC to the energy consumed by hosted IT equipment. The PUE of a DC is calculated as:

$$\text{PUE} = \frac{\sum Energy(\mathsf{DC})}{\sum Energy(\mathsf{IT})} = 1 + \frac{\sum Energy(\mathsf{non\text{-}IT})}{\sum Energy(\mathsf{IT})} \tag{2.1}$$

where $\sum Energy(\mathsf{DC})$ is the sum of energies drawn by what is not considered a computing device but required to operate a DC (so-called non-IT), such as lighting, air conditioning, etc., and the IT equipment. The PUE is a widely adopted metric, often cited by major cloud providers to demonstrate their progress in DC efficiency. While the ideal PUE is 1.0, the average PUE for a DC in 2020 was 1.58 [28]. For example, Google publishes quarterly and trailing 12-month PUE values going back to 2008 for their DC hosted globally and reports a fleet-wide PUE of 1.10 for 2021 [25]. Nonetheless, PUE has been criticized when adopted as a measure of efficiency, because it only considers energy and does not consider the effective usage of the computational resources [10, 15]. This means that a PUE can mechanically decrease by artificially increasing the IT workload, even though the DC efficiency has not been improved.

Then, the *Data Center infrastructure Efficiency* (DCiE) is the reciprocal of the PUE, defined as:

$$\mathrm{DCiE} = \frac{1}{\mathrm{PUE}} = \frac{\sum Energy(\mathsf{IT})}{\sum Energy(\mathsf{DC})} \tag{2.2}$$

One can observe that, although PUE and DCiE are the most commonly used metrics to compare the energy efficiency of data centers, they only assess the global energy efficiency and fail to provide any insight into the IT efficiency in particular [51].

GEC & ERF The Green Grid also defined three indicators related to DC energy efficiency. The *Green Energy Coefficient* (GEC) allows to compute the part of renewable energy used by the DC, reported as:

$$\mathrm{GEC} = \frac{\sum Energy_{REN}(\mathsf{DC})}{\sum Energy(\mathsf{DC})} \tag{2.3}$$

GEC has a maximum value of 1.0, indicating the ratio of energy consumed by the DC from green sources. Then, the *Energy Reuse Factor* (ERF) reports on the amount of energy reused outside of the DC:

$$\mathrm{ERF} = \frac{\sum Energy_{REUSE}(\mathsf{DC})}{\sum Energy(\mathsf{DC})} \tag{2.4}$$

The ERF value ranges from 0.0 to 1.0, where 0.0 means that no energy is being exported by the DC for further reuse, while 1.0 means that all the energy brought into the DC is reused outside of it.

DCEM The *European Telecommunications Standards Institute* (ETSI) published the ES 205 200-2-1 standard [19], which defines the *Data Centre Energy Management* (DCEM) indicator to measure energy efficiency and compare energy management efficiency in DC. This indicator meets a twofold objective: to assess the level of efficiency in DC and to compare DC efficiency between different locations and/or industrial sectors. It is composed of multiple objective *Key Performance Indicators* (KPI) to evaluate in depth the efficiency of DC [20, 19]. These indicators assess the DC efficiency from multiple perspectives: *Energy Consumption* (EC), *Task Efficiency* (TE), *Energy Reused* (REUSE) and *Renewable Energy* (REN). The documents also define the measurement points, the procedures and constraints that must be followed to take the measurements used in the KPI:

$$\mathrm{KPI}_{EC} = Energy_{REN}(\mathsf{DC}) + Energy_{FEN}(\mathsf{DC}) \tag{2.5}$$

The *energy consumption* objective KPI (KPI_{EC}) is composed of the yearly energy consumption coming from a renewable source ($Energy_{REN}(\mathsf{DC})$) and the yearly energy consumption from other power sources ($Energy_{FEN}(\mathsf{DC})$).

$$\mathrm{KPI}_{TE} = \frac{\mathrm{KPI}_{EC}}{Energy_{HE}(\mathsf{DC})} \tag{2.6}$$

The *task efficiency* objective KPI (KPI_{TE}) is defined by the ratio of energy consumption (KPI_{EC}) to the yearly energy consumed by equipment that manages data for calculation, storage or transport purposes inside the DC ($Energy_{HE}$) [20].

$$\mathrm{KPI}_{REUSE} = \frac{Energy_{REUSE}(DC)}{\mathrm{KPI}_{EC}} \tag{2.7}$$

The *energy reused* (REUSE) objective KPI (KPI_{REUSE}) is the ratio of the yearly energy reused outside of the DC (EC_{REUSE}) to its energy consumption KPI_{EC}.

$$\mathrm{KPI}_{REN} = \frac{Energy_{REN}(DC)}{\mathrm{KPI}_{EC}} \tag{2.8}$$

The *renewable energy* (REN) objective KPI (KPI_{REN}) is the ratio of the yearly energy consumption from renewable sources ($Energy_{REN}(\mathsf{DC})$) to its energy consumption KPI_{EC}.

Then, the *Data Centre Energy Management (DCEM)* is composed of the DC energy consumption ($Energy(DC)$) which is equivalent to the KPI_{EC}, and the *DC Class* (DC_{CLASS}), which is determined according to the energy use management performance (DC_P) for a given energy consumption (DC_G). DC_G is an intermediate KPI defining the energy consumption gauge, based on the range of values of the KPI_{EC} objective KPI. Each gauge has the corresponding weight factors W_{REUSE} and W_{REN}, which are used to compute the DC_P of the DC.

$$DC_P = \mathrm{KPI}_{TE} \times (1 - W_{REUSE} \times \mathrm{KPI}_{REUSE}) \times (1 - W_{REN} \times \mathrm{KPI}_{REN}) \tag{2.9}$$

The *Energy Management Performance* (DC_P) is computed by using the objectives KPIs (TE, REUSE, REN), with a mitigation factor applied to KPI_{REUSE} and W_{REN}, depending on the gauge and the policy the operator choose to promote.

CUE Finally, the *Carbon Usage Effectiveness* (CUE) (in kilograms of carbon dioxide per kilowatt-hours: kgCO2eq) aims to assess the efficiency of the energy used for the DC:

$$\mathrm{CUE} = \frac{\sum Emission_{CO2}(\mathsf{DC})}{\sum Energy(\mathsf{DC})} \tag{2.10}$$

The CUE does not take into account the emissions accountable to the manufacturing of the DC or its equipment. However, the CUE includes the carbon emissions due to the mix of energy being used by the DC.

SPUE Introduced by Barroso *et al.* in 2013 [4], the *Server PUE* (SPUE) is computed as the ratio of the server input power to its useful component power, including all the parts directly involved in the computations, namely motherboard, disks, CPUs, DRAM, GPU, I/O, etc. SPUE aims to quantify the efficiency of individual servers and authors report that state-of-the-art SPUE should be less than 1.2 at the time of writing their book. Low SPUE should highlight from optimized power supply and cooling of the components of the machine. However, to the best of our knowledge, neither the cloud operator nor the literature adopted this indicator to report on best practices in the design of hardware servers.

2.3.2 Limitations & Opportunities

In the state of the art, various energy efficiency indicators aim to evaluate globally or specific hardware parts of the infrastructures. However, none of them proposes an end-to-end approach from the DC input to the machine power supply and down to the hosted software services.

In particular, in the context of cloud infrastructure, providing an end-to-end energy indicator should allow deeper analysis and tuning of the infrastructure hardware and software components to optimize energy usage at large.

Chapter 3

SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers

As demonstrated in Section 2.1, while physical power meters offer a suitable solution to monitor the power consumption of physical servers, they fail to support the energy profiling at a finer granularity: dealing with the software services that are distributed across such infrastructures. To overcome this limitation, software-defined power meters build on power models to estimate the power consumption of software artifacts to identify potential energy hotspots and leaks in software systems [42] or improve the management of resources [53]. However, existing software-defined power meters are integrating power models that are statically designed, or learned prior to any deployment in production [13, 37]. This may result in inaccuracies in power estimations when facing unforeseen environments or workloads, thus affecting the exploitation process. As many distributed infrastructures, such as clusters or data centers, have to deal with the scheduling of unforeseen jobs, in particular when handling black-box virtual machines, we can conclude that the adoption of such static power models [12] has to be considered as inadequate in production. We therefore believe that the state-of-the-art in this domain should move towards the integration of more dynamic power models that can adjust themselves at runtime to better reflect the variation of the underlying workloads and to cope with the potential heterogeneity of the host machines.

In this chapter, we therefore introduce SMARTWATTS, as a self-calibrating software-defined power meter that can automatically adjust its CPU and DRAM power models to meet the power accuracy requirements of monitored software containers. Our approach builds on the principles of sequential learning principles and proposes to exploit coarse-grained power monitors like *Running Power Average Limit* (RAPL), which is commonly available on modern Intel's and AMD's micro-architecture generations, to control the estimation error. We have implemented SMARTWATTS as an open source power meter to integrate our self-calibrating approach, which is in charge of automatically adjusting the power model whenever some deviation from the ground truth is detected. When triggered, the computation of a new power model aggregates the past performance metrics from all the deployed containers to infer a more accurate power model and to seamlessly update the software-defined power meter configuration, without any interruption. The deployment of SMARTWATTS in various environments, ranging from private clouds to distributed HPC clusters, demonstrates that SMARTWATTS can ensure accurate real-time power estimations (less than 2 Watts of error on average, at a frequency of 2 Hz) at the granularity of processes, containers and virtual machines. Interestingly, the introduction of sequential learning in software-defined power meters eliminates the learning phase, which usually last from minutes to hours or days, depending on the complexity of the hosting infrastructure [12, 13].

Additionally, our software-defined approach does not require any specific hardware investment as SMARTWATTS can build upon embedded power sensors, like RAPL, whenever they are available. As stated in Section 1.3, the code of SMARTWATTS is freely available online as open-source software to encourage its deployment at scale and to leverage the adoption and reproduction of our results. The key contributions of this chapter can therefore be summarized as follows:

1. a self-calibrating power modelling approach,
2. CPU & DRAM models supporting power states,
3. an open source implementation of our approach,
4. an assessment on container-based environments.

In the remainder of this chapter, we start by introducing this contribution (cf. Section 3.1). Then, we detail the implementation of SMARTWATTS as an extension of the BITWATTS middleware framework (cf. Section 3.2) and we empirically assess its validity on three scenarios (cf. Section 3.3). We conclude and provide some perspectives for this work in Section 3.4.

3.1 SMARTWATTS Power Monitoring

We therefore propose to support self-calibrating power models that leverage *Reference Measurements* and *Hardware Performance Counters* (HWPC) to estimate the power consumption at the granularity of software containers along multiple resources: CPU and DRAM. More specifically, this contribution builds upon two widely available system interfaces: *RAPL* to collect baseline measurements for CPU and DRAM power consumptions, as well as Linux's

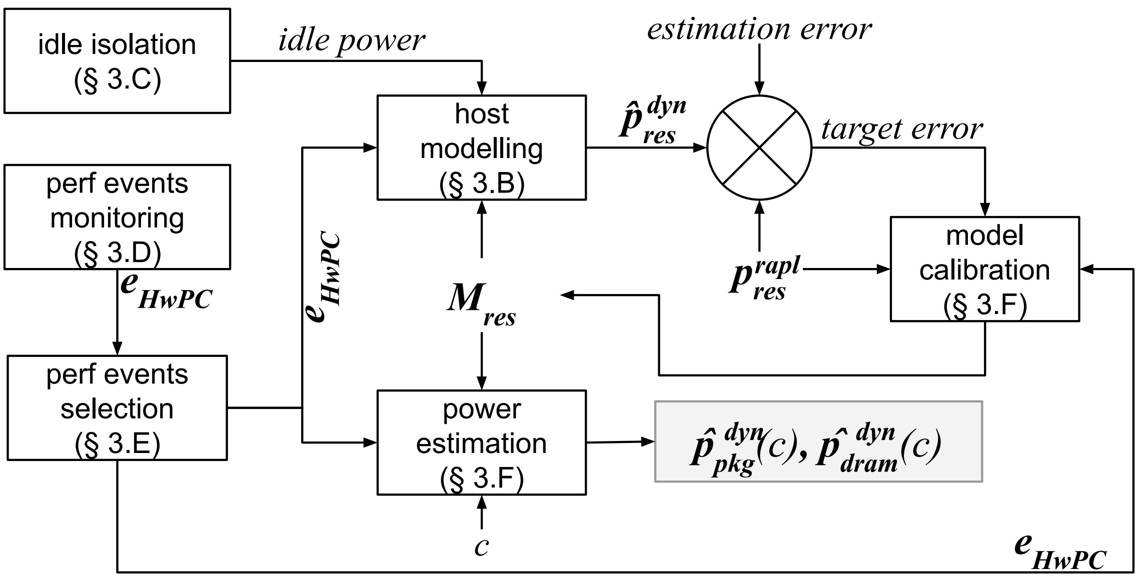

Fig. 3.1 Overview of SMARTWATTS

perf_events interface to capture the *Hardware Performance Counters* (HWPC) events used to estimate the per-container power consumption from resource-specific power models, which are adjusted at runtime.

3.1.1 Overview of SMARTWATTS

Figure 3.1 introduces the general architecture of SMARTWATTS. SMARTWATTS manages at runtime a set of self-calibrated power models (M^f_{res}) for each *power-monitorable* resource *res* (*e.g.*, CPU, DRAM). These power models are then used by SMARTWATTS to estimate the power consumptions of *i)* the host $\hat{p}_{res}$ and *ii)* all the hosted containers c: $\hat{p}_{res}(c)$.

SMARTWATTS uses $\hat{p}_{res}$ to continuously assess the accuracy of the managed power models (M^f_{res}) and to ensure that the estimated power consumption does not diverge from the baseline measurements reported by RAPL (p^{rapl}_{res}, cf. Section 3.1.2). Whenever the estimated power consumption error (ε_{res}) diverges from the baseline measurements beyond a configured threshold, SMARTWATTS automatically triggers a new online calibration process of the diverging power model to better match the current input workload.

To better capture the dynamic power consumption of the host, SMARTWATTS needs to isolate its static consumption. To do so, we use a dedicated component that activates when the machine is at rest—*e.g.*, after booting (cf. Section 3.1.3)—to monitor the power activity of the host.

In addition to the static constant, SMARTWATTS estimates the power consumption of the host from a set of raw input values that refers to HWPC events, which are selected *at runtime* (cf. Section 3.1.5).

This design ensures that SMARTWATTS keeps adjusting its power models to maximize the accuracy of power estimations. Therefore, unlike the state-of-the-art power monitoring solutions, SMARTWATTS does not suffer from estimation errors due to the adoption of an inappropriate power model as it autonomously optimizes the underlying power model whenever an accuracy anomaly is detected.

3.1.2 Modelling the Host Power Consumption

For each resource $res \in \{pkg, dram\}$ exposed by the RAPL interface, the associated power consumption p_{res}^{rapl} can be modelled as:

$$p_{res}^{rapl} = p_{res}^{static} + p_{res}^{dyn} \tag{3.1}$$

where p_{res}^{static} refers to the static power consumption of the monitored resource (cf. Section 3.1.3), and p_{res}^{dyn} reflects the dynamic power dissipated by the processor along the sampling period.

Then, we can compute a power model $M_{res}^{f} = [\alpha_0, \cdots, \alpha_n]$ that correlates, for a given frequency f (among available frequencies F, cf. Section 3.1.4), the dynamic power consumption ($\hat{p}_{res}^{dyn}$) to the raw metrics reported by a set of of *Hardware Performance Counter* (HwPC) events (cf. Section 3.1.5), $E_{res}^{f} = [e_0, \ldots, e_n]$:

$$\exists f \in F,\ \hat{p}_{res}^{dyn} = M_{res}^{f} \cdot E_{res}^{f} \tag{3.2}$$

We build M_{res}^{f} from an *Elastic net* regression—that linearly combines the L1 and L2 penalties of the *Lasso* and *Ridge* methods—applied on the past k samples $S_k^f = \langle p_{res}^{dyn}, E_{res}^{f} \rangle$, with $p_{res}^{dyn} = p_{res}^{rapl} - p_{res}^{static}$. To ensure that the power consumption of every single container is linear with regards to the global power consumption of the node, we enforce the inference of positive coefficients when computing the regression and we check that the intercept belongs to the range $[0, \text{TDP}]$ where TDP refers to the *Thermal Design Power* of the CPU. By comparing $p_{res}^{dyn} + p_{res}^{static}$ with p_{res}^{rapl}, we can continuously estimate the error $\varepsilon_{res} = \mid p_{res}^{dyn} - \hat{p}_{res}^{dyn} \mid$ from estimated values in order to monitor the accuracy of the power model M_{res}^{f}. Whenever the error exceeds a given threshold set by the administrator, a new power model is generated for the frequency f by integrating the latest samples.

3.1.3 Isolating the Static Power Consumption

Isolating the static power consumption of a node is a challenging issue as it requires to reach a quiescient state in order to capture the power consumption of the host at rest. To capture this information, we designed and implemented a power logger component that runs as a lightweight daemon with low priority that periodically logs the package and DRAM power consumptions reported by RAPL. Then, we compute the *median* value and the *interquartile range* (IQR) from gathered measurements to define the p_{res}^{static} constant as : $p_{res}^{static} = median_{res} - 1.5 \times IQR_{res}$. This approach intends to filter out outliers reported by RAPL, including periodic measurement errors we observed, and to consider the lowest power consumption observed along a given period of time as the static consumption of a node.

By default, SMARTWATTS assumes that the static consumption of the host does not require to be spread across the active containers. However, other power accounting policies can be implemented. For example, by reporting an empty static consumption, SMARTWATTS will share it across the running containers depending on their activity.

3.1.4 Monitoring Power States & HWPC Events

As previously introduced, the accuracy of a power model M_{res}^{f} strongly depends on *i)* the selection of relevant input features (HWPC events e_n) and *ii)* the acquisition of input values that are evenly distributed along the reference power consumption range. This is one of the reasons why the input workloads used in standard calibration phases are often critical to capture an accurate power model that reflects the power consumption of a host for a given class of applications. SMARTWATTS rather promotes a self-calibrating approach that does not impose the choice of a specific benchmark or workload, but exploits the ongoing activity variations of the host machine to continuously adjust its power models. To achieve this, SMARTWATTS monitors selected sets of HWPC events and stores the associated samples in memory. To better deal with the power features of hardware components, we group the input samples per operating frequency. This allows to calibrate frequency-specific power models when an estimation arises, with the goal to converge automatically to a stable and precise power model over the time.

By balancing the samples along the range of frequencies operated by the processor, SMARTWATTS ensures that the power model learning phase does not overfit the current context of execution, which may lead to the generation of unstable power models, thus impacting the accuracy of the power measurements. The sampling tuples S_k^f are grouped into memory as frequency layers $L_{res}^{f} = [S_0^f, ..., S_n^f]$, which are the raw features we maintain to build M_{res}^{f}.

To store the samples in the layer corresponding to the current frequency of the processor, SMARTWATTS compute the average running frequency as follows:

$$F_{avg} = F_{base} * \frac{\Delta\ \texttt{APERF}}{\Delta\ \texttt{MPERF}} \tag{3.3}$$

where F_{base} is the processor base frequency constant extracted from the *Model Specific Registers* (MSR) PLATFORM_INFO. APERF and MPERF are MSR-based counters that increment at the current and maximum frequencies, respectively. These counters are continuously updated, hence they report on a precise average frequency without consuming the limited HWPC slots. Interestingly, the performance power states, such as P-states and Turbo Boost, will be accounted by these counters as they act mainly on the frequency of the core in order to boost the performance. The idle optimization states (C-states) will also be accounted as they mainly reduce of the average frequency of the core towards its *Max Efficiency Frequency* before being powered-down.

3.1.5 Selecting the Correlated HWPC Events

The second challenge of SMARTWATTS consists in selecting at runtime the relevant HWPC events that can be exploited to accurately estimate the power consumption. To do so, we list the available events exposed by the host's *Performance Monitoring Units* (PMU) and we evaluate their correlation with the power consumption reported by RAPL. Instead of testing each available HWPC events, we narrow the search using the PMU associated to the modelled component—*i.e.*, we consider the HWPC events from the core PMU to model the PKG power consumption. As reference events, we consider `unhalted-cycles` for the package and `llc-misses` for the DRAM, which are the standard HWPC events available across many processor architectures, and have been widely used by the state of the art to design power models [12, 13, 37]. To elect a HWPC event as a candidate for the power model, we first compute the Pearson coefficient $r_{e,p}$ for n values reported by each monitored HWPC event e and the base power consumption p reported by RAPL:

$$r_{e,p} = \frac{\sum_{i=1}^{n} (e_i - \overline{e})\ (p_i - \overline{p})}{\sqrt{\sum_{i=1}^{n} (e_i - \overline{e})^2}\ \sqrt{\sum_{i=1}^{n} (p_i - \overline{p})^2}} \tag{3.4}$$

Then, SMARTWATTS stores the list of HWPC events that exhibit a better correlation coefficient r than the baseline event for DRAM and PKG. This list of elected HWPC events is further used as input features to implement the PKG and DRAM power models exploited by SMARTWATTS.

3.1.6 Estimating the Container Power Consumption

Given that we learn the power model M_{res}^{f} from aggregated events of the running containers or virtual machines C, $E_{res}^{f} = \sum_{c \in C} E_{res}^{f}(c)$, we can predict the power consumption of any container c by applying the inferred power model M_{res}^{f} at the scale of the container's events $E_{res}^{f}(c)$:

$$\exists f \in F,\ \forall c \in C,\ \hat{p}_{res}^{dyn}(c) = M_{res}^{f} \cdot E_{res}^{f}(c) \tag{3.5}$$

Then, we distribute the value of the intercept i that is included in the estimate $\hat{p}_{res}^{dyn}(c)$ proportionally to the dynamic part of the consumption of c

$$\forall c \in C,\ \tilde{p}_{res}^{dyn}(c) = \hat{p}_{res}^{dyn}(c) - i \times (1 - \frac{\hat{p}_{res}^{dyn}(c) - i}{\hat{p}_{res}^{dyn} - i}) \tag{3.6}$$

In theory, one can expect that $\hat{p}_{res}^{dyn} \overset{!}{=} p_{res}^{dyn}$ if the model perfectly estimates the dynamic power consumption but, in practice, the predicted value may introduce an error $\varepsilon_{res} = \mid p_{res}^{dyn} - \hat{p}_{res}^{dyn} \mid$. Therefore, we cap the power consumption of any container c as:

$$\forall c \in C,\ \lceil \tilde{p}_{res}^{dyn}(c) \rceil = \frac{p_{res}^{dyn} \times \tilde{p}_{res}^{dyn}(c)}{\hat{p}_{res}^{dyn}} \tag{3.7}$$

to ensure that $p_{res}^{dyn} = \sum_{c \in C} \lceil \tilde{p}_{res}^{dyn}(c) \rceil$, thus avoiding potential outliers. Thanks to this approach, we can also report on the confidence interval of the power consumption of containers by scaling down the observed global error:

$$\forall c \in C,\ \varepsilon_{res}(c) = \frac{\tilde{p}_{res}^{dyn}(c)}{\hat{p}_{res}^{dyn}} \times \varepsilon_{res} \tag{3.8}$$

In the following sections, we derive and implement the above formula to report on the power consumption of *pkg* and *dram* resources. Our empirical evaluations report on the capped power consumptions for *pkg* ($\lceil \hat{p}_{pkg}^{dyn} \rceil$) and *dram* ($\lceil \hat{p}_{dram}^{dyn} \rceil$), as well as the associated errors ε_{pkg} and ε_{dram}, respectively.

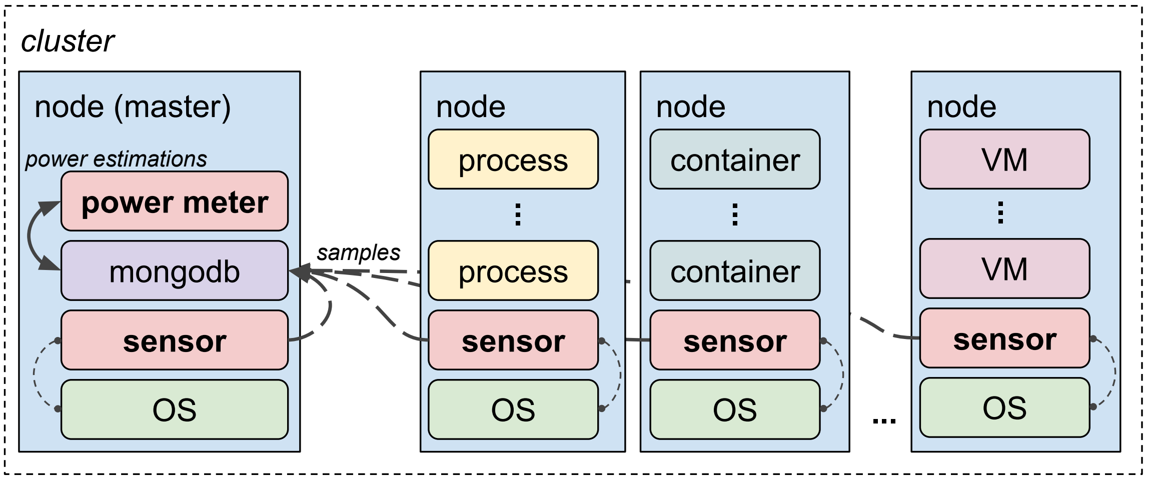

Fig. 3.2 Deployment of SMARTWATTS

3.2 Implementation of SMARTWATTS

We implemented SMARTWATTS as a modular software system that can run atop a wide diversity of production environments. As depicted in Figure 3.2, our open source implementation of SMARTWATTS mostly relies on 2 software components—a sensor and a power meter—which are connected with a MONGODB database.[1] MONGODB offers a flexible and persistent buffer to store input metrics and power estimations. The sensor is designed as a lightweight process that is intended to run on target nodes with a limited impact. The power meter is a remote service that can be deployed whenever needed. SMARTWATTS uses this feature to support both online and *post mortem* power estimations, depending on use cases.

3.2.1 Client-side Sensor

The component sensor consists in a lightweight software daemon deployed on all the nodes that need to be monitored.

Static power isolation When the node boots, the sensor starts the idle consumption isolation phase (cf. Section 3.1.3) by monitoring the PKG and DRAM power consumptions reported by RAPL along the global idle CPU time and the `fork`, `exec` and `exit` process control activities provided by Linux *process information pseudo-filesystem* (procfs). Whenever a process control activity or the global idle CPU time exceed 99 % during this phase,

[1]https://www.mongodb.com

the power samples are discarded to prevent the impact of background activities on the static power isolation process. As stated in Section 3.1.3, this phase is only required when the idle attribution policy considers the idle consumption as a power leakage. It is not needed to run this phase as long as there is no change in the hardware configuration of the machine (specifically CPU or DRAM changes).

Event selection Once completed, the sensor switches to the event selection phase (cf. Section 3.1.5). To select the most accurate *Hardware Performance Counters* (HwPC) to estimate the power of a given node, SmartWatts need to identify the HwPC statistically correlated with the power consumption of the components. For that, the sensor monitors the power consumption reported by RAPL and the maximum simultaneous HwPC events possible without multiplexing, as it can a significant noise and distort the correlation coefficient of the events, over a (configurable) period of 30 ticks. The maximal amount of simultaneous HwPC events depends of the micro-architecture of the CPU and will be detected at runtime using the PMU detection feature of the *libpfm4* library.[2] We then correlate the power consumption with the values of the monitored HwPC events and rank them by highest correlation with RAPL and lowest correlation across the other HwPC. Whenever possible, fixed HwPC event counters are selected in priority to avoid consuming a programmable counter.

Control groups SmartWatts leverages the *control groups* (Cgroups) implemented by Linux to support a wide range of monitoring granularities, from single processes, to software containers (Docker),[3] to virtual machines (using Libvirt).[4] The sensor also implement a kernel module that is in charge of configuring the Cgroups to monitor the power consumption of kernel and system activities, which is not supported by default. To do so, this module defines 2 dedicated Cgroups for the roots of the system and the kernel process hierarchy.

Event monitoring Once done with the above preliminary phases, the sensor automatically starts to monitor the selected HwPC events together with RAPL measurements for the DRAM and CPU components at a given frequency and it reports these samples to the MongoDB backend (cf. Section 3.1.4). The sensor monitors the selected HwPC events for the host and all the Cgroups synchronously to ensure that all the reported samples are consistent when computing the power models.

[2]http://perfmon2.sourceforge.net
[3]https://docker.com
[4]https://libvirt.org

3.2.2 Introducing the POWERAPI Toolkit

POWERAPI is an open-source middleware toolkit for building software-defined power meters (i.e. formulas) in Python.[5] This toolkit supports the acquisition of raw metrics from a wide diversity of sensors (eg., physical power meters, hardware energy monitoring interfaces, performance counters) and the delivery of power consumptions via different channels (including file system and various kinds of databases). POWERAPI adopts a modular architecture based on the actor programming model, which is used to deliver power estimations at scale by devoting one actor per formula. Multiple deployment methods are available, such as containers and packages for various Linux distributions, in order to cope with the heterogeneity of the underlying infrastructures. Application developers can then leverage POWERAPI to focus on the development of the core logic of their own formula and benefit from the various features of the toolkit.

3.2.3 Server-side Power Meter

The power meter is implemented as a software service that requires to be deployed on a single node (*e.g.*, the master of a cluster). The power meter can be used online to produce real-time power estimations or offline to conduct *post mortem* analysis. This component consumes the input samples stored in the MONGODB database and produces power estimations accordingly. We implemented SMARTWATTS as a POWERAPI formula, which allow the use of a wide range of input/output data storage technologies (such as MongoDB, InfluxDB, etc.) and to delivers power estimations at scale.

Power modelling The power meter provides an abstraction to build power models. In this contribution, the power model we report on is handled by *Scikit-Learn*, which is the *de facto* standard Python library for general-purpose machine learning.[6] We embed the Ridge regression of *Scikit-Learn* in an actor, which is in charge of delivering a power estimation whenever a new sample is fetched from the database.

Model calibration When the error reported by the power model exceeds the threshold defined by the user, the power meter triggers a new calibration of the power model to take into account the latest samples. This new power model is checked against the last sample to estimate its accuracy. If it estimates the power consumption below the configured threshold, then the actor is updated accordingly.

[5]https://github.com/powerapi-ng/powerapi
[6]https://scikit-learn.org

Power estimation Power estimations are delivered at the scale of a node and for the Cgroups of interest. These scopes of these Cgroups can reflect the activity of nodes' kernel and system, as well as any job or service running in the monitored environment. These power estimations can then be aggregated by owner, service identifier or any other key, depending on use cases. They can also be aggregated along time to report on the energy footprint of a given software system.

3.3 Validation of SMARTWATTS

This section assesses the efficiency and the accuracy of SMARTWATTS to evaluate the power consumption of running software containers.

3.3.1 Evaluation Methodology

We follow the experimental guidelines reported by [55] to enforce the quality of our results. In this case, the experiments are executed on real environments, we fully share the machine specifications, the version of the software used, and publish our code to ease the reproducibility of our results. The benchmarks used for the validation are standard and widely used in the evaluation of other contributions in the literature. [5, 23, 12, 27, 13] For the sake of reproducible research, SMARTWATTS, the necessary tools, deployment scripts and resulting datasets are open-source and publicly available on GitHub.[7]

Testbeds & workloads While our production-scale deployments of SMARTWATTS cover both KUBERNETES and OPENSTACK clusters, for the purpose of this contribution, we chose to report on more standard benchmarks, like STRESS NG[8] and NASA's *NAS Parallel Benchmarks* (NPB) [2] to highlight the benefits of our approach.

Our setups are reproduced on the GRID5000 testbed infrastructure,[9] which provides multiple clusters composed of powerful nodes. In this evaluation, we use a Dell PowerEdge C6420 server having two Intel® Xeon® Gold 6130 Processors (Skylake) and 192 GB of memory (12 slots of 16 GB DDR4 2666MT/s RDIMMs). We are using the Ubuntu 18.04.3 LTS Linux distribution running with the 4.15.0-88-generic Kernel version, where only a minimal set of daemons is running in the background. As stated in 3.2.1, we are using the Cgroups to monitor the activity of the running processes independently. In the case of the

[7]https://github.com/powerapi-ng/smartwatts-formula
[8]https://launchpad.net/stress-ng
[9]https://www.grid5000.fr

system services managed by systemd and the services running in Docker containers, their Cgroups membership is automatically handled as part of their lifetime management.

For this host, the reported TDP for the CPU is 125 Watts and 26 Watts for the DRAM. Theses values were obtained from the `PKG_POWER_INFO` and `DRAM_POWER_INFO` *Model Specific Registers* (MSR). The energy and performance optimization features of the CPU—*i.e.*, *Hardware P-States* (HWP), *Hyper-Threading* (HT), *Turbo Boost* (TB) and *C-states*, are fully enabled and use the default configuration of the distribution. The default CPU scaling driver and governor for the distribution are *intel_pstate* and *powersave*.

In all our experiments, we configure SMARTWATTS to report power measurements twice a second ($2Hz$) with an error threshold of 5 Watts for the PKG and 1 Watt for the DRAM.

Objectives We evaluate SMARTWATTS with the following criteria:

- The *quality* of the power estimations when running sequential and parallel workloads,
- The *accuracy and stability* of the power models across different workloads,
- The *overhead* of the SMARTWATTS sensor component on the monitored host.

3.3.2 Experimental Results

Quality of estimations Figure 3.3 first reports on the PKG and DRAM power consumptions we obtained with SMARTWATTS. The first line (rapl) refers to the ground truth power measurements we sample for the PKG and the DRAM via the HWPC events `RAPL_ENERGY_PKG` and `RAPL_ENERGY_DRAM`, respectively. The second line (global) refers to the power measurements estimated by SMARTWATTS for the PKG and the DRAM components from `CPU_CLK_THREAD_UNHALTED:REF_P` and `CPU_CLK_THREAD_UNHALTED:THREAD_P` fixed counters, `INSTRUCTIONS_RETIRED`, and `LLC_MISSES` programmable counters. The list of events has been automatically selected by the sensor component as presenting the best correlation with RAPL samples, as described in Section 3.1.5. The error for each of the power models are further discussed in Figures 3.4 and 3.5.

The lines kernel and system isolate the power consumption induced by all kernel and system activities. Kernel activities include devices specific background operations, such as *Network interface controller* (NIC) and disks I/O processing queues, while system activities cover the different services, like the SSH server and Docker daemon, running on the node.

The remaining lines reports on individual power consumptions of a set of NPB benchmarks, which are executed in sequence (`lu`, `ep`, `ft`) or concurrently (`ft`, `cg`, `ep`, `lu`, `mg`) with variable number of cores (ranging from 8 to 32 cores). One can observe that SMARTWATTS

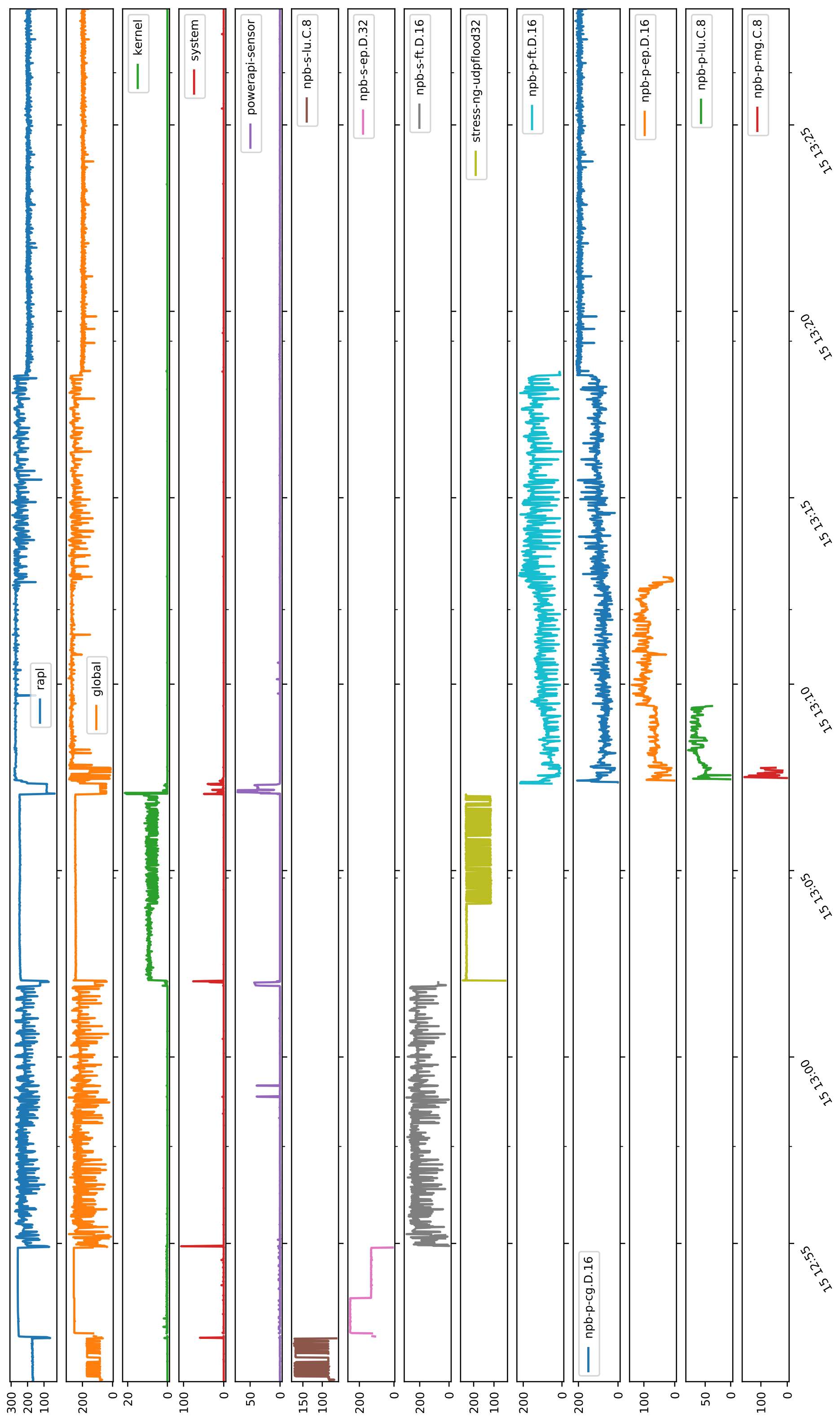

Fig. 3.3 Evolution of the aggregated PKG & DRAM power consumption (in Watts) along time and containers

supports the isolation of power consumptions at process-level by leveraging Linux Cgroups. This granularity allows SMARTWATTS to monitor indifferently processes, containers or virtual machines.

We also run `stress-ng` to observe potential side effects on the kernel activity by starting 32 workers that attempt to flood the host with UDP packets to random ports. While it remains negligible compared to the power consumption of the UDP flood process (3.26 W vs. 94.46 W on average), one can observe that this stress induces a lot of activity at the kernel to handle I/O, while the rest system is not severely impacted.

One can also observe that our sensor (`powerapi-sensor`) induces a negligible overhead (less than 0.2 Watts) with regards to the consumption of surrounding activities.

Estimation accuracy Figures 3.4 and 3.5 reports on the distribution of estimation errors we observed per frequency and globally (right part of the plots) for the above scenario. We also report on the number of estimations produced for each of the frequency (upper part of the plots). While the error threshold for CPU and DRAM is set to 5 Watts and 1 Watts, one can observe that SMARTWATTS succeeds to estimate the power consumption with less than 3 Watts and 0.5 Watt of error for the PKG and DRAM components, respectively. The only case where estimation error grows beyond this threshold refers to the frequency 1000 Hz of the CPU (cf. Figures 3.4).The frequency 1000 Hz refers to the idle frequency of the node and the sporadic triggering of activities in this frequency induces a chaotic workload which is more difficult to capture for SMARTWATTS given the limited number of samples acquired in this frequency (130 samples against 3555 samples for the frequency 2700 Hz).

The DRAM component, however, provides a more straightforward behavior to model with the selected HWPC events and therefore reports an excellent accuracy, no matter the operating frequency of the CPU package (cf. Figure 3.5).

The accuracy of the power models generated by SMARTWATTS are further detailed in Table 3.1. While our approach succeeds to deliver accurate estimations of the power consumption for both CPU and DRAM components, the maximum error refers to the bootstrapping phase of the sensor that requires to acquire a sufficiently representative number of samples in order to build a stable and accurate power model.

Model stability Beyond the capability to accurately estimate the power consumption of software containers, we are also interested in assessing the capability of SMARTWATTS to generate stable power models over time. Tables 3.2 and 3.3 therefore reports, for each frequency, on metrics about the stability of power models. In particular, we look at the number of correct estimations produced by the power models in a given frequency. Given

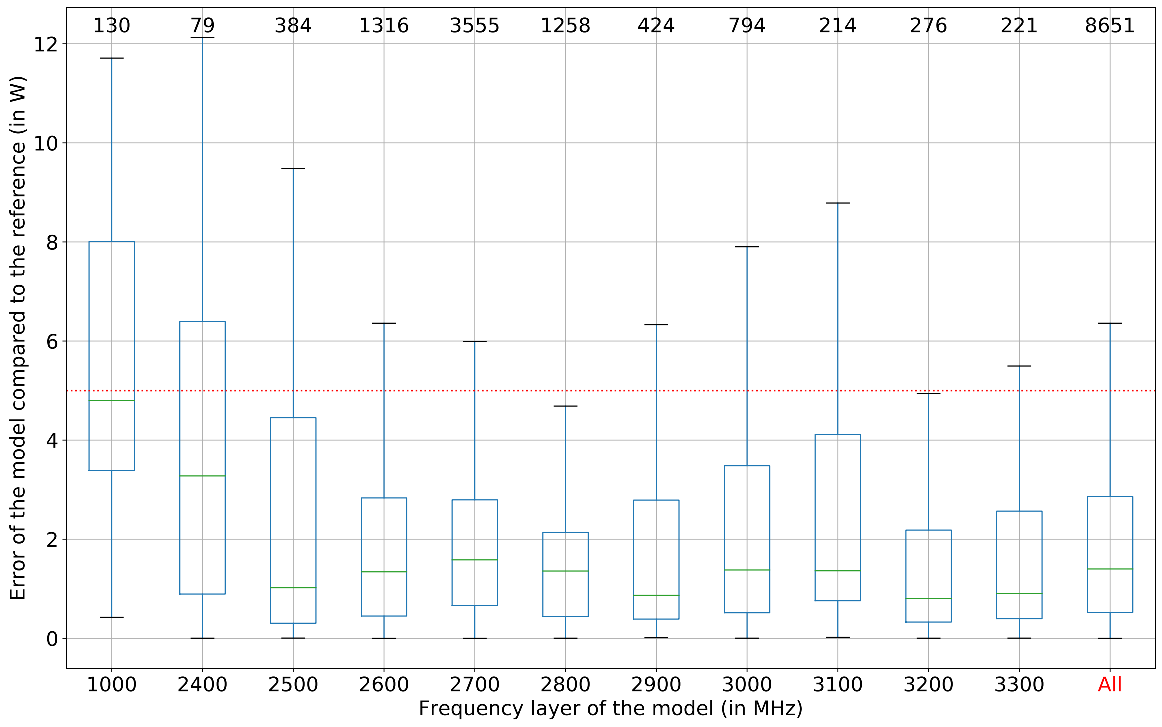

Fig. 3.4 Global & per-frequency error rate of the PKG power models

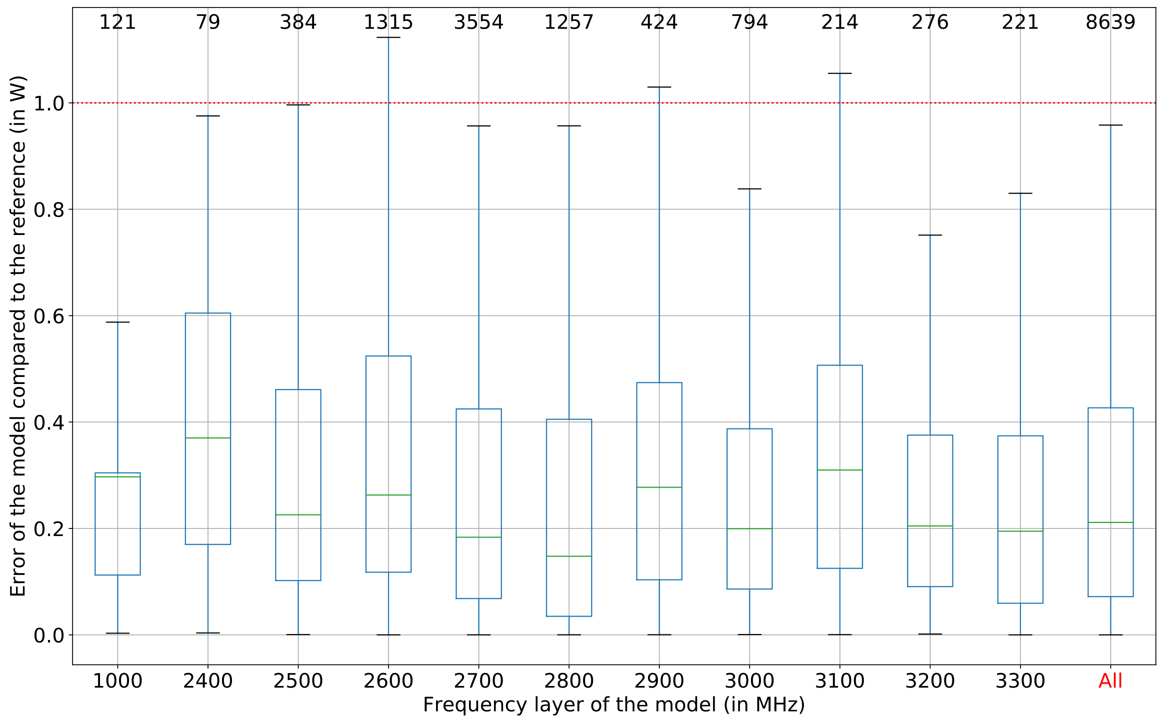

Fig. 3.5 Global & per-frequency error rate of the DRAM power models

Table 3.1 Per-socket PKG & DRAM power models accuracy

Resource	Socket	ε_{min}	ε_{max}	ε_{mean}	ε_{std}
PKG	0	0.000 W	117.563 W	2.625 W	5.548 W
	1	0.000 W	97.906 W	3.278 W	6.743 W
DRAM	0	0.000 W	26.114 W	0.465 W	0.839 W
	1	0.000 W	5.737 W	0.309 W	0.442 W

our input workloads, we can observe that SMARTWATTS succeeds to reuse a given power model up to 594 estimations, depending on frequencies. While we observed that the stability of our power models strongly depends on the sampling frequency, the error threshold, as well as the input workloads, one should note that the overhead for calibrating a power model in a given frequency does not take more than a couple milliseconds, which is perfectly acceptable when monitoring software systems in production.

Table 3.2 PKG power models stability per frequency

Frequency	models	total	min	max	mean	std
1000	130	130	1	13	1.710	1.881
2400	35	79	1	19	2.323	3.345
2500	72	384	1	58	3.764	10.226
2600	152	1316	1	417	6.045	29.904
2700	239	3555	1	358	8.610	27.591
2800	95	1258	1	294	9.530	39.320
2900	64	424	1	266	5.108	29.061
3000	134	794	1	121	4.783	13.581
3100	56	214	1	127	3.890	17.061
3200	53	276	1	115	5.307	17.817
3300	44	221	1	79	4.911	12.770

Monitoring overhead Regarding the runtime overhead of SMARTWATTS, one can observe in Figure 3.3 that the power consumption of SMARTWATTS is negligible compared to the hosted software containers. To estimate this overhead, we leverage the fact that the sensor component is running inside a software container, thus enabling SMARTWATTS to estimate its own power consumption. In particular, one can note in Table 3.4 that the sensor power consumption represents 0.333 Watts for the PKG and 0.030 Watts for the DRAM, on average, when running at a frequency of $2 Hz$. The usage of the *Hardware Performance Counters* (HwPC) is well known for its very low impact on the observed system, hence it does not

Table 3.3 DRAM power models stability per frequency

Frequency	models	total	min	max	mean	std
1000	21	121	1	58	6.050	14.009
2400	18	79	1	14	4.647	3.390
2500	82	384	1	129	4.425	14.082
2600	290	1315	1	70	3.887	7.961
2700	499	3554	1	594	4.980	25.637
2800	106	1257	1	300	7.618	37.541
2900	89	424	1	33	4.326	5.469
3000	137	794	1	60	5.293	8.270
3100	58	214	1	14	3.754	3.444
3200	50	276	1	62	5.520	9.361
3300	34	221	1	29	6.314	8.312

induce runtime performance penalties [12, 17, 35, 44]. Additionally, we carefully took care of the cost of sampling these HwPC events and executing as little as possible instructions on the monitored nodes.

Table 3.4 Per-component power consumption of the sensor

Power	min	max	mean	std
PKG	0.0 W	63.961 W	0.333 W	3.097 W
DRAM	0.0 W	8.365 W	0.030 W	0.283 W

By proposing a lightweight and packaged software solution that can be easily deployed across monitored hosts, we facilitate the integration of power monitoring in large-scale computing infrastructures. Furthermore, the modular architecture of SMARTWATTS can accommodate existing monitoring infrastructures, like KUBERNETES METRICS or OPENSTACK CEILOMETER, to report on the power consumption of applications. The following section therefore demonstrates this capability by deploying a distributed case study atop of a KUBERNETES cluster.

3.3.3 Tracking the Energy Consumption of Distributed Systems

To further illustrate the capabilities of SMARTWATTS, we take inspiration from [11] to deploy a distributed software systems that processes messages forwarded by IoT devices to a pipeline of processors connected by a KAFKA cluster to a CASSANDRA storage backend. Figure 3.6

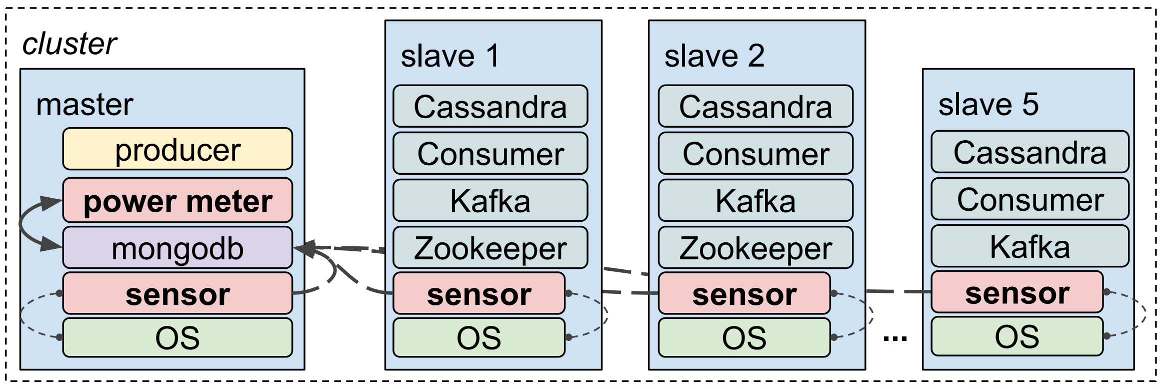

Fig. 3.6 Deployment of Kubernetes IoT backend services across 6 nodes

depicts the deployment of this distributed system on a KUBERNETES cluster composed of 1 master and 5 slave nodes of the same configuration as stated in 3.3.1. The input workload consists in a producer injecting messages in the cluster with a throughput ranging from 10 to 100 MB/s.

Figure 3.7 reports on the evolution of the power consumption per service while injecting the workload from the master node. One can observe that, when increasing the message throughput, the most impacted service is the Consumer, which requires extensive energy to process all the messages enqueued by the Kafka service. This saturation of the Consumer service seems to represent a core bottleneck in the application.

To further dive into this problem, we consider another perspective on the deployment in order to investigate the source of this efficiency limitation. While the execution of this workload requires 1.32 MJoules of energy to process the whole dataset, Figure 3.8 further dives inside the distribution of the energy consumption of individual pods along the PKG and DRAM components as a Sankey diagram [40]. This diagram builds on the capability of SMARTWATTS to aggregate power estimations along time to report on the energy consumption, as well as its capacity to track power consumption from software processes (on left-hand side) down to hardware components (on the right-hand side). This diagram can therefore be used to better understand how a distributed software system takes advantage of the underlying hardware components to execute a given workload. In particular, one can observe that 91 % of the energy is spent by the CPU package, while the Consumer service drains 65 % of the energy consumption of the monitored scenario. Interestingly, one can observe that this energy consumption is evenly distributed across the 5 slaves, thus fully benefiting from the pod replication support of KUBERNETES. The observed energy overhead is not due to the saturation of a single node, but rather seems to be distributed across the nodes, therefore highlighting an issue in the code of the Consumer service. This

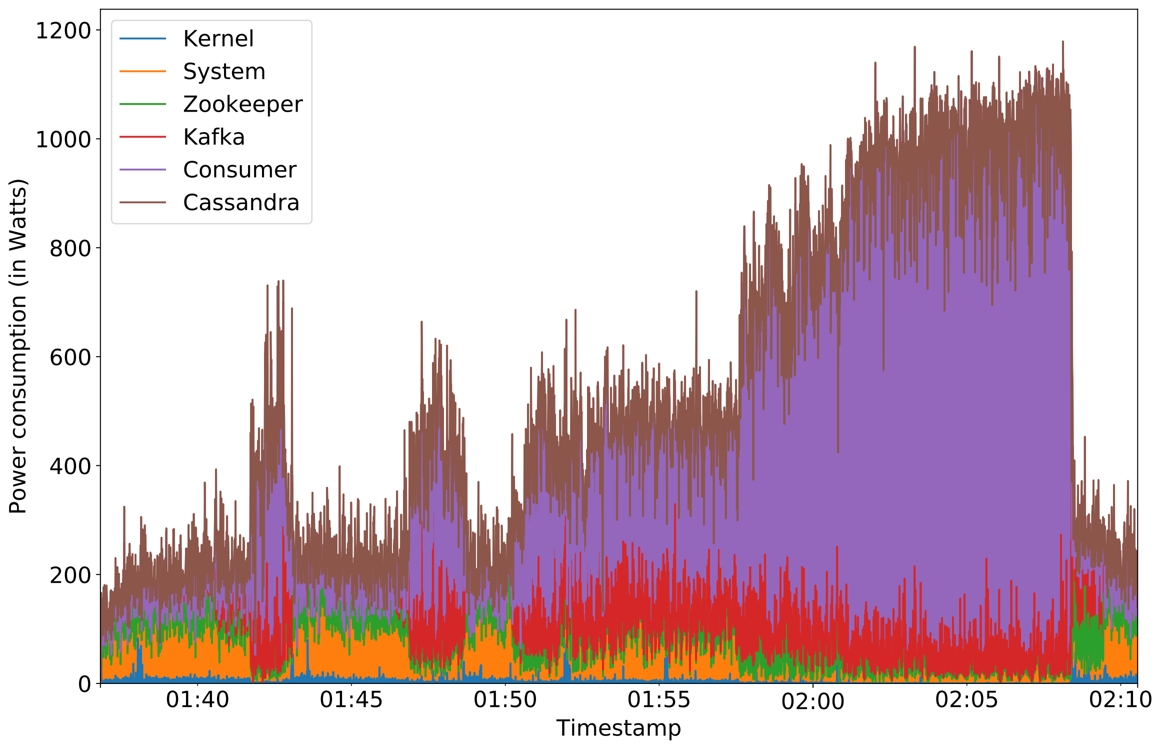

Fig. 3.7 Monitoring of service-level power consumptions

issue is related to the acknowledgement of write requests by the CASSANDRA service, which prevents the CONSUMER service to process pending messages.

We believe that, thanks to SMARTWATTS, system administrators and developers can collaborate on identifying energy hotspots in their deployment and adjusting the configuration accordingly.

3.4 Summary

Power consumption is a critical concern in modern computing infrastructures, from clusters to data centers. While the state of practice offers tools to monitor the power consumption at a coarse granularity (*e.g.*, nodes, sockets), the literature fails to propose generic power models, which can be used to estimate the power consumption of software artifacts.

In this chapter, we therefore presented a novel approach, named SMARTWATTS, to deliver per-container power estimations for the CPU and DRAM components. In particular, we propose to support self-calibrating power models to estimate the CPU and DRAM power consumption of software containers. Unlike static power models that are trained for a specific workload, our power models leverage sequential learning principles to be adjusted online in

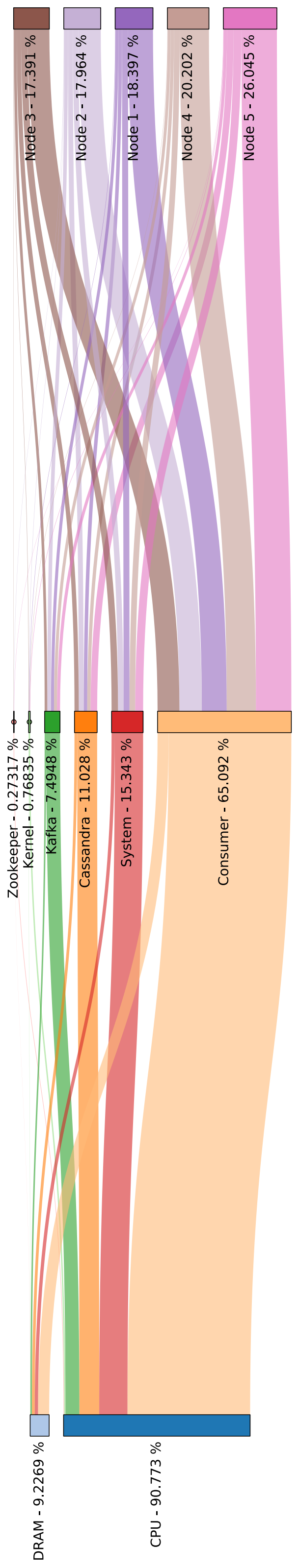

Fig. 3.8 Distribution of the energy consumption across nodes and resources

order to match unexpected workload evolutions and thus maximize the accuracy of power estimations.

While we demonstrate this approach using Intel RAPL and the Linux's *perf_events* interface, we strongly believe that it can be used as a solid basis and generalized to other architectures and system components. In particular, other CPUs implementing the RAPL interface, such as AMD Ryzen, can be supported by our approach.

Thanks to SMARTWATTS, system administrators and developers can monitor the power consumption of individual containers and identify potential optimizations to apply in the distributed system they manage. Instead of addressing performance issues by adding more resources, we believe that SMARTWATTS can favorably contribute to increase the energy efficiency of distributed software systems at large.

Chapter 4

SelfWatts: On-the-fly Selection of Performance Events for Power Meters

As stated in Section 2.2, while software-defined power meters offers a solution to support process-level power estimations, through the implementation of dedicated power models that leverage system-level metrics to estimate the power consumption at the granularity of software services [21, 11]. The design of the underlying power models that estimate the power consumption of the monitored software components keeps being a long and fragile process that remains tightly coupled to the host machine [53, 35, 13]. Furthermore, the deployment of such software-defined power meters remains a critical issue when facing the diversity of hardware settings in the wild, which prevents a wider adoption by the industry. In particular, the proposed power models either cannot be exploited because of the unavailability of required metrics or, at best, deliver incorrect power estimations of hosted software services [56, 7].

This key limitation, therefore, calls for more adaptive approaches that can adjust and optimize the power model to the hardware constraints of a given deployment target. More specifically, this contribution addresses the self-optimization of power models by *i)* automatically selecting the most relevant set of hardware performance events and *ii)* continuously inferring the best power model for the target architecture. The proposed approach ensures that our self-optimized power models keep delivering accurate power estimations while fitting to the hardware constraints imposed by the deployment. Our approach is made available as a software solution, named SELFWATTS, that can be quickly deployed at the scale of a data center to monitor the power consumption of software containers or *virtual machines* (VM), with negligible overhead. Once deployed, SELFWATTS keeps exploring the space of available hardware performance events to detect if unexplored events contribute more favorably to the accuracy of the power model.

Interestingly, we show that SELFWATTS delivers real-time power estimations that compete with the state-of-the-art software-defined power models while offering a plug-and-play solution to data center administrator for monitoring the power consumption of their infrastructure services, as well as reporting the energy consumption of their customers, no matter their hardware constraints. This contribution thus paves the way for more sustainable cloud services by exposing this key performance indicator to interested stakeholders (*e.g.*, cloud administrators and customers) and encourage them to reduce their environmental footprint.

As stated in Section 1.3, the code of SELFWATTS is freely available online as open-source software to encourage its deployment at scale and to leverage the adoption and reproduction of our results.

The key contributions of this chapter can therefore be summarized as follows:

1. a self-calibrating power modelling approach,
2. CPU & DRAM models supporting power states,
3. an open source implementation of our approach,
4. an assessment of the accuracy, runtime overhead and on container-based environments.

In the remainder of this chapter, we start by introducing our contribution (cf. Section 4.1). Then, we detail the implementation of SELFWATTS as an extension of the (already described in Section 3) SMARTWATTS software-defined power-meter (cf. Section 4.2) and we empirically assess its validity on three scenarios (cf. Section 4.3). We conclude and provide some perspectives for this work in Section 4.4.

4.1 Power Monitoring with SelfWatts

4.1.1 Approach Overview

To learn power models that continuously deliver the best accuracy, no matter the workload and the deployment target, we propose an approach that explores the space of performance events incrementally and evaluates the impact of available events on the energy consumption of the monitored host. We propose to apply this approach at runtime, while the software-defined power meter keeps running to challenge the current power model with alternative performance events. This online learning approach aims to ensure the convergence of the power model towards the best combination of performance events that characterize the power consumption of a given workload and target architecture. While this approach may require some time to converge, we consider that the context of cloud computing assumes long-term monitoring that can accommodate such a training phase, given the timescale of virtual machine deployments.

As introduced in Figure 4.1, our approach introduces a feedback loop between the inference of a power model and the monitoring of performance events. By doing so, the monitoring becomes aware of the relevance of the selected events and can adjust the exploration of performance events accordingly.

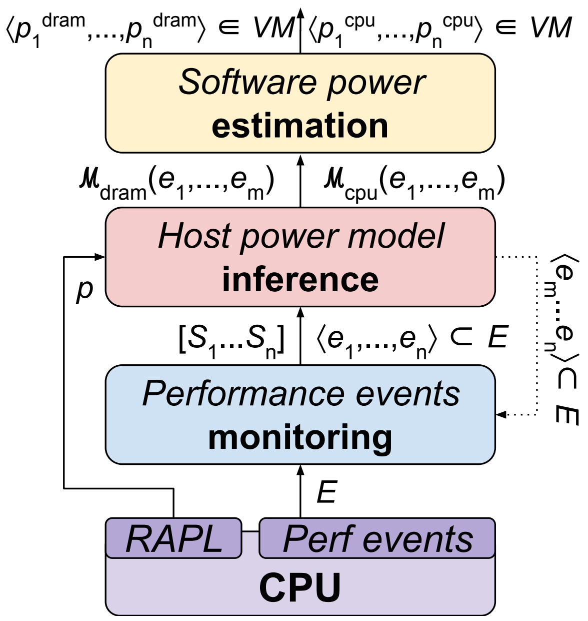
$\langle p_1^{dram},\ldots,p_n^{dram}\rangle \in VM$
$\langle p_1^{cpu},\ldots,p_n^{cpu}\rangle \in VM$
Software power estimation
$\mathcal{M}_{dram}(e_1,\ldots,e_m)$
$\mathcal{M}_{cpu}(e_1,\ldots,e_m)$
Host power model inference
p
$\langle e_m \ldots e_n\rangle \subset E$
$[S_1 \ldots S_n]$
$\langle e_1,\ldots,e_n\rangle \subset E$
Performance events monitoring
E
RAPL
Perf events
CPU

Fig. 4.1 Overview of the SELFWATTS approach.

More specifically, the monitoring component starts by selecting a subset of performance events $\langle e_1, \ldots, e_n \rangle \in E$ from the list of events E available on the target CPU (cf. Section 4.1.4). The cardinality of the selected set of events depends on the number of HWPC slots that can be used concurrently. The monitoring component forwards metrics samples for the selected events to the inference component, which applies a supervised machine learning algorithm to establish a relationship between the energy consumption p retrieved from the RAPL interface and the selected events (cf. Section 4.1.2). This model is further exploited by the estimation component to report on the power consumption of individual virtual machines or containers (cf. Section 4.1.3). The inference component also reports on the list of irrelevant performance events, which can be used by the monitoring component to select another set of performances events among E. In this approach, performance events used by the power model are kept by the monitoring component, which only replaces the irrelevant performance events. By doing so, the power model is intended to converge towards a model that retains the performance events delivering the most accurate power estimations.

The following sections dive into the specific challenges of each component, starting from the host power model inference (cf. Section 4.1.2), before explaining how software power estimation works (cf. Section 4.1.3) and finally, how the performance events monitoring explores the space of available events (cf. Section 4.1.4).

4.1.2 Host Power Model Inference

First, we consider that, for any hardware resource $res \in \{pkg, dram\}$ exposed by the RAPL interface, the associated power consumption p_{res}^{rapl} can be modelled as:

$$p_{res}^{rapl} = p_{res}^{static} + p_{res}^{dyn} \tag{4.1}$$

where p_{res}^{static} refers to the static power consumption of the monitored resource, and p_{res}^{dyn} reflects the dynamic power dissipated by the processor along the sampling period. By default, SELFWATTS consider p_{res}^{static} to be 0 and will spread the static consumption of the host across the active containers and virtual machines proportionally to theirs activity. However, other power accounting policies where the static consumption needs to be specifically handled can be implemented.

As previously introduced, the accuracy of a power model M_{res}^{f} strongly depends on *i)* the selection of relevant input features (performance events e_i) and *ii)* the acquisition of input samples that are evenly distributed along with the reference power consumption range. To better deal with the power features of hardware components, we group the input samples per operating frequency $f \in F$, being the set of all frequencies operated by the hardware resource. Thus, we learn frequency-specific power models, aiming to converge automatically to a stable and precise representation over time. By tagging the samples along with the frequency operated by the processor, SELFWATTS ensures that the learned power models do not overfit the current context of execution, which may lead to inaccurate power models. The sampling tuples S_k^f, containing the raw sampled performance events, are grouped into memory as frequency layers $L_{res}^f = [S_1^f, \dots, S_n^f]$, which are the input features we maintain to build M_{res}^f.

To classify the samples in the layer corresponding to the current frequency of the processor, SELFWATTS compute the average running frequency $\overline{f}$ as follows:

$$\overline{f} = f_{base} * \frac{\Delta\ \mathtt{APERF}}{\Delta\ \mathtt{MPERF}} \tag{4.2}$$

where f_{base} is the processor base frequency constant extracted from the field *Package Maximum Non-Turbo Ratio* of the `PLATFORM_INFO` *Model Specific Registers* (MSR). The `APERF` and `MPERF` variables are MSR-based counters that increment at the current and maximum frequencies, respectively. These counters are continuously updated, hence they report on a precise average frequency without consuming the limited HwPC slots. Interestingly, the performance power states, such as P-states and Turbo Boost, are covered by these counters as they act mainly on the frequency of the core to boost the performance. The idle optimization states (C-states) are also included, as they mainly reduce the average frequency of the core towards its *Max Efficiency Frequency* before being powered-down.

To filter out the irrelevant performance events for the power model, we compute a ranking of performance events using a *Recursive Feature Elimination* (RFE) and a cross-validated selection of the best number of features. This phase considers not only the raw input samples, but also transformed samples obtained from a fixed set of transformers, such as $Log, Exp, Sqrt, Cbrt, MinMaxScaler, StandardScaler, RobustScaler, Normalizer$, to boost the accuracy of the inferred models. The output of this phase is an ordered list of event and transformer combinations that can be used to infer a power model based on the subset of relevant performance events $E = \langle e_1, \ldots, e_m \rangle \subset \langle e_1, \ldots, e_n \rangle \subset E$. From these filtered events, we can infer a frequency-specific power model $M^f_{res} = [\gamma_1, \cdots, \gamma_m]$ that correlates, for a given frequency f, the dynamic power consumption ($\hat{p}^{dyn}_{res}$) to the raw samples for the frequency f that associated to set of relevant performance events E, $L^f_{res}(E)$:

$$\exists f \in F,\ \hat{p}^{dyn}_{res} = M^f_{res} \cdot L^f_{res}(E) \tag{4.3}$$

In SELFWATTS, we learn M^f_{res} from a *Lasso* regression applied over the past k samples filtered by E, $S^f_k = \langle p^{dyn}_{res}, e_1, \cdots, e_m \rangle$, with $p^{dyn}_{res} = p^{rapl}_{res} - p^{static}_{res}$.

To ensure that the power consumption of hosted virtual machines (or any application) is consistent with regards to the global power consumption of the host, we check that the intercept belongs to the range $[0, \text{TDP}]$ where TDP refers to the *Thermal Design Power* of the CPU. By comparing $p^{dyn}_{res} + p^{static}_{res}$ with p^{rapl}_{res}, we can continuously estimate the error $\varepsilon_{res} = \mid p^{dyn}_{res} - \hat{p}^{dyn}_{res} \mid$ from estimated values in order to monitor the accuracy of the power model M^f_{res}. This estimation error is then stored in a sliding window of k samples to keep track of the accuracy of the model over time:

$$\tilde{\varepsilon}^f_{\text{res}} = med(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) \tag{4.4}$$

Whenever the median error $\tilde{\varepsilon}^f_{\text{res}}$ of the window exceeds a given threshold α set by the administrator, we assumes that a new power model requires to be inferred for the frequency f by reasoning over the latest input samples forwarded by the monitoring component.

4.1.3 Software Power Estimation

Given that we learn the host power model M^f_{res} from aggregated relevant performance events samples, $L^f_{res} = \sum_{c\in C} L^f_{res}(c)$, we can predict the power consumption of any container or virtual machine $c \in C$ by applying the inferred power model M^f_{res} to the input samples associated to c, $L^f_{res}(c)$:

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

Then, we distribute the value of the intercept i that is included in the estimate $\hat{p}^{dyn}_{res}(c)$ proportionally to the dynamic part of the consumption of c

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

In theory, one can expect that $\hat{p}^{dyn}_{res} \overset{!}{=} p^{dyn}_{res}$ if the model perfectly estimates the dynamic power consumption but, in practice, the predicted value may introduce an error $\varepsilon_{res} =| p^{dyn}_{res} - \hat{p}^{dyn}_{res} |$. Therefore, we cap the power consumption of any container c as:

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

to ensure that $p^{dyn}_{res} = \sum_{c\in C} \lceil \tilde{p}^{dyn}_{res}(c) \rceil$, thus avoiding potential outliers. Thanks to this approach, we can also report on the confidence interval of the power consumption of containers by scaling down the observed global error:

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

4.1.4 Performance Events Monitoring

SELFWATTS aims to explore the space of available performance events to monitor the most relevant set of events E that accurately model the power consumption of the host.

To do so, the monitoring component lists all the available performance events E that can be monitored from the target architecture together with the number of HWPC slots s that can be used for monitoring these performance events without triggering multiplexing effects, which seriously impact the accuracy of the input samples. Then, it randomly picks a set of s performance events $\langle e_1 \ldots e_n \rangle \subset E$ and configures the HWPC slots accordingly. The resulting set of input samples $\overline{S_k^f}$ is forwarded to the power model inference component to be used whenever a new power model requires to be learned.

When picking the set of events to be monitored, the events that are included in the current power model are kept in the new set, while the previous subset of events that were tagged as irrelevant by the inference component is replaced by another set of unexplored performance events taken from E.

One can note that, to speed up the convergence of power models and reduce the delay to produce accurate estimations, one can configure the monitoring component with a set of performance events that should be considered in priority. For example, this hint can be used to favour performance events, like *unhalted core cycles*, *unhalted ref cycles*, *instructions retired*, *llc misses/prefetch*, or *memory transactions cycles*, which are commonly adopted by the literature and then let SELFWATTS evaluate the relevance of these events in the deployment context, possibly identifying alternative performance events that better fit the power consumption of the target host.

4.2 Implementation Details

SELFWATTS builds on the POWERAPI toolkit presented in Section 3.2.2 to implement the self-optimizing power modelling approach. Interestingly, POWERAPI was designed as a modular software system that can run atop a wide diversity of production environments. More specifically, SELFWATTS is an extension of the SMARTWATTS power meter presented in Chapter 3, that introduces three key components: `Controller`, `Sensor`, and `Formula`. `Controller` and `Sensor` are covering the monitoring phase of SELFWATTS, while the inference and estimation phases are implemented by the `Formula`.

4.2.1 A Sensor to Monitor Performance Events

This component, developed in C, is a lightweight software daemon that uses *Hardware Performance Counters* (HWPC) to monitor a given set of performance events for all the Cgroups available on the host. We monitor Linux's Kernel *Control Groups* (Cgroups), as they are widely used by software container (Docker, LXC, Kubernetes) and virtual machine (libvirt) technologies, thus offering the adequate granularity to deliver software power estimations. The `Sensor`, therefore, periodically reports on samples of performance events per Cgroup, with a frequency that can be configured upon start ($\beta = 2\,\text{Hz}$ by default). This component represents the minimal requirement to obtain power estimations from a target architecture and is carefully implemented to limit its impact on hardware resources (CPU, DRAM) and co-located processes (containers, virtual machines).

4.2.2 A Controller to Explore Performance Events

This component, developed in C, is in charge of controlling the `Sensor` by configuring it with the appropriate set of performance events to monitor. The `Controller` uses the *Libpfm4* library[1] to detect the available *Performance Monitoring Unit* (PMU) of the target architecture, the number of HwPC slots and list the associated performance events. In SelfWatts, the resulting set of performance events is randomly shuffled before starting the exploration. The `Controller` obtains the list of irrelevant performance events from the `Formula` and kills the active `Sensor` to replace it by another instance configured with a new selection of performance events. This new selection includes all the relevant performance events of the active power models (including CPU and DRAM power models) and completes the set with the next events consumed from the shuffled list. Once fully consumed, the list of available performance events is reset and shuffled with another random seed, thus resulting in a different combination of performance events to be explored by our approach.

4.2.3 A Formula to Optimize Power Models

As long as the median error of the active power models remains below the configured threshold ($\alpha = 5W$ by default), the `Formula` component delivers power estimation at the pace of forwarded samples (twice a second by default). If the median error exceeds this α threshold, then the `Formula` discards the active power model to infer a new power model from the new set of performance events. Given that the list of irrelevant events is forwarded to the `Controller` as soon as a new power model is computed, the `Controller` can anticipate by starting a new `Sensor` with a set of performance events. The list of relevant performance events will therefore be used to deliver power estimations as long as the power model is kept active, while the remaining performance events will be accumulated and consumed by a new power model will be requested, thus drastically reducing the delay to infer a new power model. The power model inference is implemented in Python and leverages *Scikit-learn*, which is the *de facto* standard Python library for general-purpose machine learning.[2]

[1]https://perfmon2.sourceforge.net/
[2]https://scikit-learn.org

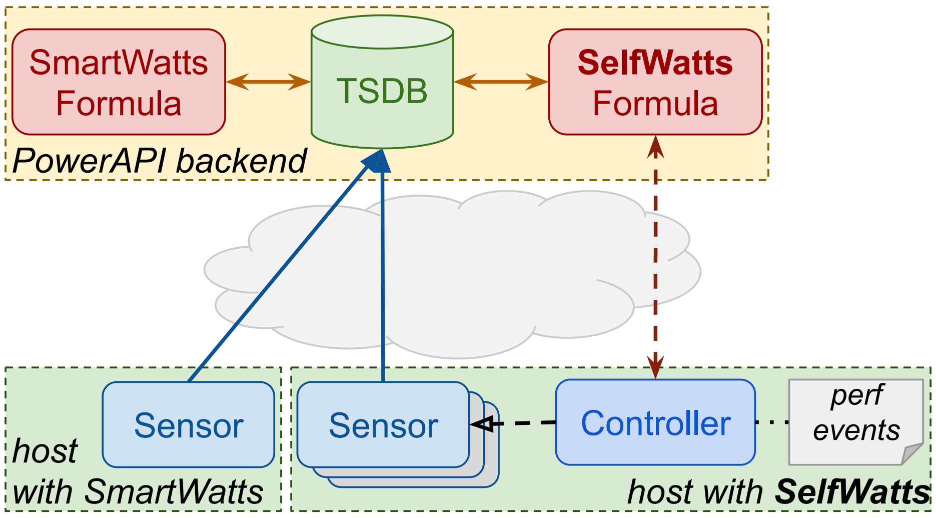

Fig. 4.2 Deployment of SELFWATTS, compared to SMARTWATTS.

4.2.4 Deployment of SELFWATTS

All the components of SELFWATTS are made to be deployed as Docker containers to ease their deployment and lifecycle management in container-based environments. The `Controller` embeds the `Sensor` and is deployed on all host machines. Optionally, one can use a MONGODB instance as a message queue to communicate input samples through a publish-subscribe pattern and as a *time series database* (TSDB) to store power estimations.

Figure 4.2 depicts an example of SELFWATTS configuration and compares it to the deployment of SMARTWATTS. In this configuration, the `Formula` component can be hosted by a remote virtual machine in charge of delivering the power estimations for all the monitored hosts and Cgroups. But, given the modularity of POWERAPI, SELFWATTS can also be deployed as a standalone solution where the components `Controller`/`Sensor` and `Formula` are co-located on the host machine. The latter deployment scheme is the one we consider in the following section to assess the accuracy and overhead of SELFWATTS in a "*worst-case*" configuration, which requires all the computations to be completed on the monitored host.

4.3 Empirical Evaluation

This section assesses the accuracy and the efficiency of SELFWATTS to select relevant performance events and to estimate the power consumption of hosted virtual machines with accuracy. More specifically, this section evaluate our contribution on the following criteria:

- How does SELFWATTS compare to SMARTWATTS in terms of accuracy?

- What is the runtime overhead of SELFWATTS?
- How does SELFWATTS adapt to different target architectures?

4.3.1 Evaluation Methodology

We follow the experimental guidelines reported by [55] to enforce the quality of our empirical results. In this case, the experiments are executed on real environments, we fully share the machine specifications and the version and configuration of the software used. The benchmarks used for the validation are standard and widely used in the evaluation of other contributions in the literature. [5, 23, 12, 27, 13] For the sake of reproducible research, SELFWATTS, the necessary tools, deployment scripts, and resulting datasets are open-source and publicly available on GitHub.[3]

Testbeds

Our setups can be reproduced on the GRID5000 testbed infrastructure,[4] which provides large clusters of machines for experiment-driven research. To assess the versatility of our approach, we consider several heterogeneous processor architectures that exhibit different characteristics and combinations of power-aware features, as reported in Table 4.1.

Table 4.1 Testbed hardware settings

Model	Dell PowerEdge C6420	Dell PowerEdge R730	Dell PowerEdge R630
CPU	Intel Xeon Gold 6130	Intel Xeon E5-2650 v4	Intel Xeon E5-2630 v3
Generation	Skylake	Broadwell	Haswell
Cores per-socket	16	12	8
Thread(s) per-core	32	24	16
Socket(s)	2	2	2
TDP	125 W	105 W	85 W
Memory	192 GiB	128 GiB	128 GiB
# Perf. Events	257	260	265
# HWPC slots	3 fixed / 4 generic	3 fixed / 4 generic	3 fixed / 4 generic

Both the host and virtual machines are using the Ubuntu 20.04.1 LTS Linux distribution with the `5.4.0-53-generic` Kernel version, where only a minimal set of daemons are

[3]https://github.com/powerapi-ng
[4]https://www.grid5000.fr

running in background. All the selected workloads run inside QEMU[5] virtual machines managed by libvirt.[6]

Workloads

Our workloads are based on standard benchmarks, like STRESS NG[7] and NASA's *NAS Parallel Benchmarks* (NPB 3) [2], to highlight the benefits of our approach. The experiment workload is split into two phases, SEQUENTIAL where these benchmarks will run one after the other, and PARALLEL where all the benchmarks will run concurrently.

Power meters

In all our experiments, we configure the `Controller/Sensor` components SELFWATTS to report on power estimations twice a second ($\beta = 2\,\text{Hz}$), and the FORMULA component with an error threshold of $\alpha = 5W$, which are the default parameters of SELFWATTS.

We evaluate and compare the following configurations:

1. SmartWatts refers to the configuration of the SMARTWATTS power meter as previously introduced in Section 3.3,
2. SELFWATTS (default) to the configuration of SELFWATTS with default settings,
3. SELFWATTS *with fixed events* starts SELFWATTS with the following `x86` performance events: `UNHALTED_REFERENCE_CYCLES`, `UNHALTED_CORE_CYCLES`, `INSTRUCTION_-RETIRED`, which consumes the 3 fixed HWPC slots and are commonly considered by the state of the art.

4.3.2 Experimental Results

We start by reporting in Figures 4.3 and 4.4 on the power estimations reported by the default configuration of SELFWATTS when running our sequential and parallel workloads, respectively. The cumulated execution of these workloads lasts for 30 minutes on a Dell PowerEdge C6420 machine and both figures report on 3 classes of power estimations. First, the kernel and system power profiles reflect the Linux kernel and all the operating system activities, respectively. One can observe that the power consumption of the kernel and system layers remains low in general, by reaching up to 2 W in the sequential phase, but may go above 25 W when dealing with the UDP stress in the parallel phase.

[5]https://www.qemu.org
[6]https://libvirt.org
[7]https://launchpad.net/stress-ng

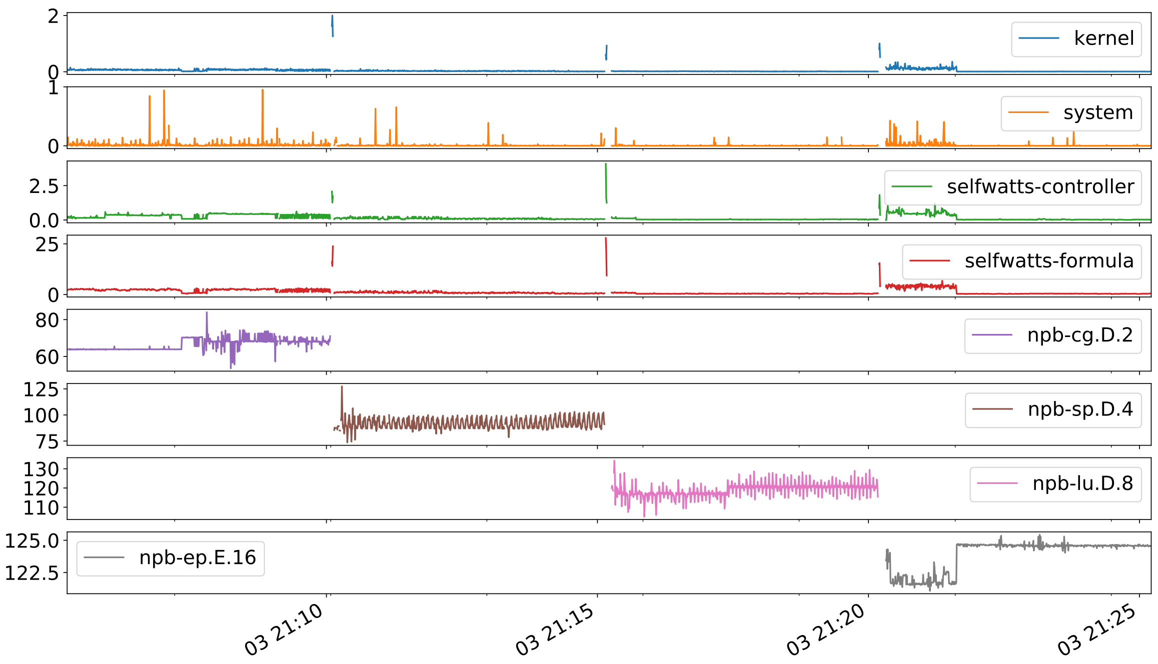

Fig. 4.3 Power estimations per VM for SEQUENTIAL phase

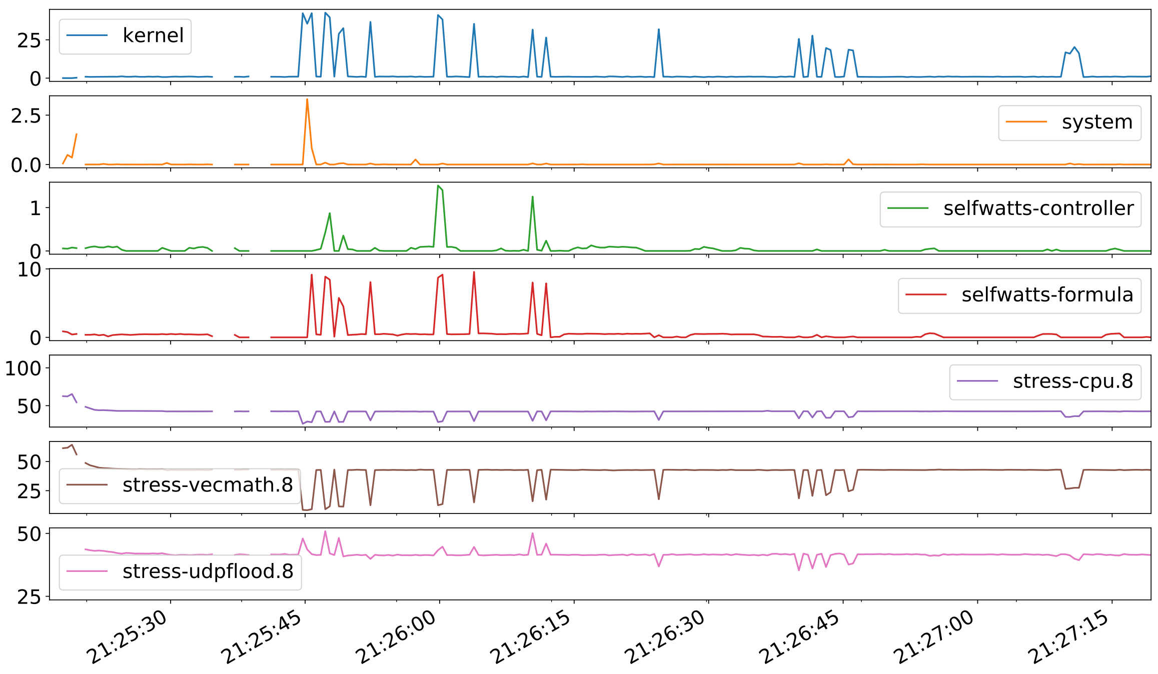

Fig. 4.4 Power estimations per VM for PARALLEL phase

Then, the selfwatts-controller and selfwatts-formula power profiles illustrate the activity of SELFWATTS along with the benchmark. This power profile highlights periods where SELFWATTS invests energy to find a new power model by exploring alternative performance events before reaching more stable period with a reduced power consumption that reflects the exploitation of a stable[8] and accurate power model.

Finally, the lower part of both figures depicts the individual power profiles of NPB and STRESS NG workloads, which are isolated in dedicated virtual machines. One can observe that, no matter the workload and the number of involved cores, SELFWATTS keeps delivering power estimations with accuracy and low overhead. We further investigate these claims in the remainder of this section.

Estimation accuracy

To assess the accuracy of SELFWATTS, we start by reporting on the error ε of the 3 configurations under study. The statistics reported in Figure 4.5 show that SELFWATTS succeeds to compete with SmartWatts in terms of accuracy by reaching the same error on average (2 W for the host estimation), far below the error threshold of $5W$ we used for all the configurations. Yet, SELFWATTS reports on more occurrences of larger errors, as its exploration phase may lead to inaccurate estimations for short periods, compared to SmartWatts which boots with an accurate model and never explores alternative performance events, essentially adjusting the coefficients of the power model M by computing a new *Ridge* regression, while SELFWATTS combines RSE and a *Lasso* regression to perform on-the-fly performance events selection and power model optimization. Our approach, therefore, goes one step beyond SmartWatts by adopting a more dynamic power modeling approach that takes the freedom to consider alternative performance events to optimize the power model.

To further investigate the errors reported by SELFWATTS, we compare the evolution of this error ε along with phases and steps of the workloads. We, therefore, split the SEQUENTIAL phase into 4 consecutive steps, which are aligned with the 4 NPB applications that are executed during this phase (cf. Figure 4.6). One can observe that, no matter the applications, both configurations of SELFWATTS reach a similar accuracy compared to SmartWatts.

These results can be further confirmed with the PARALLEL phase of our workload in Figure 4.7. Although SELFWATTS can be perceived as less accurate than SmartWatts, our manual investigations revealed that SmartWatts entered the PARALLEL phase with a power model that exhibited a high error ratio and triggered the inference of a much more accurate

[8]a power model is considered as stable as long as it keeps estimating host power consumption with a median error $\tilde{\varepsilon}$ below the configured threshold α.

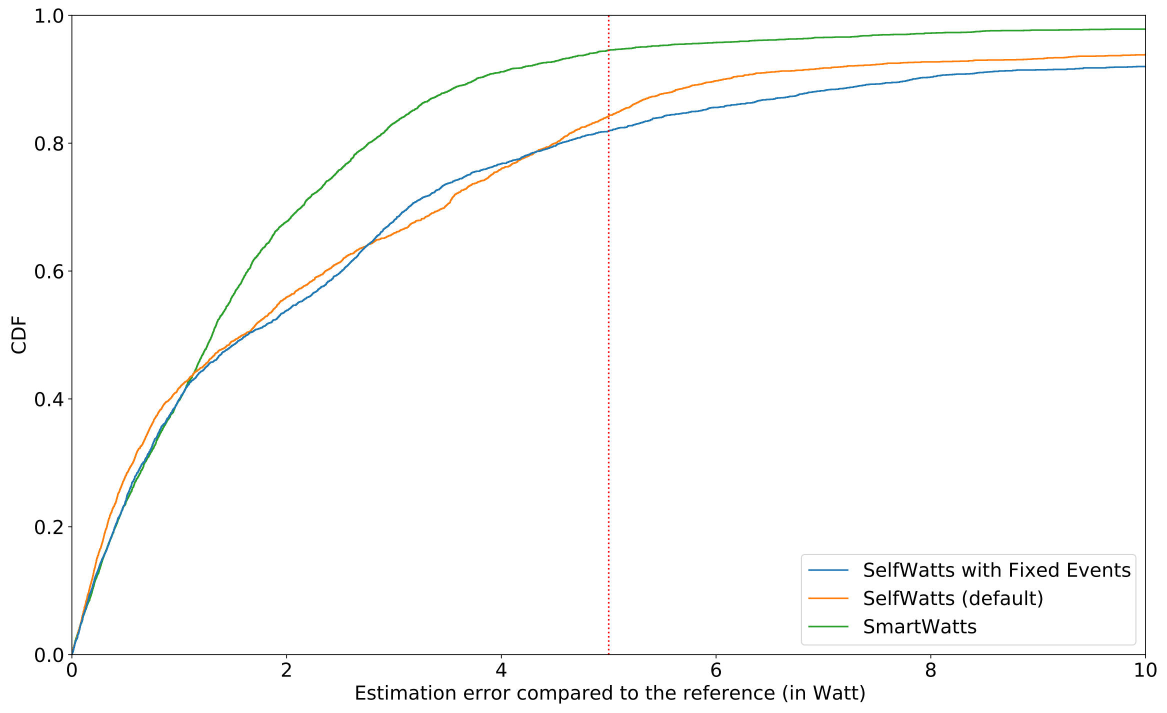

Fig. 4.5 Cumulative Distribution Function (CDF) of estimation errors ε for SELFWATTS and SmartWatts

power model, while the power models exploited by the two configurations of SELFWATTS turned out to be a stable, but a bit less accurate (still far below the configured error threshold).

Runtime overhead

To evaluate the runtime overhead of SELFWATTS, we explore the power consumption of its components in order to investigate the overhead imposed by our solution on the monitored host. Tables 4.2 and 4.3 more specifically report on the average power consumption of the `Controller/Sensor` and `Formula` components we implemented. The `Formula` component runs with the PYPY runtime for Python, version 7.3.3.[9] Interestingly, as both `Controller/Sensor` and `Formula` components are deployed as DOCKER containers, SELFWATTS can monitor and deliver fine-grained power estimations of its components in addition to the monitored VMs.

We also study the evolution of the power consumption of SELFWATTS when monitoring an increasing number of hosted virtual machines on a single node. Thus, Figures 4.8 and 4.9 report on the power consumptions of the DRAM and CPU hardware resources for both components when monitoring a growing number of hosted virtual machines that ranges from

[9]https://www.pypy.org

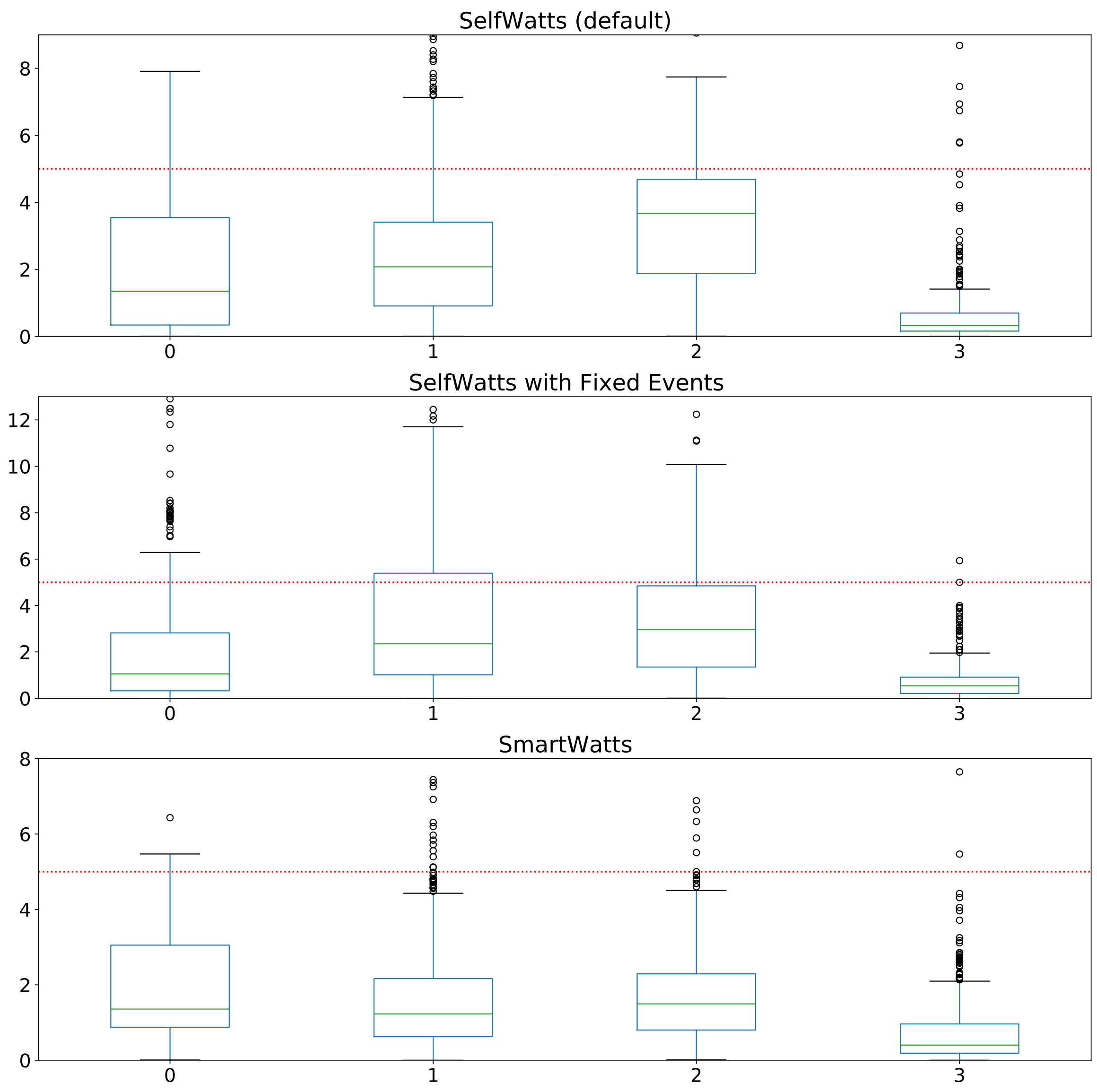

Fig. 4.6 Estimation errors ε per application involved in the SEQUENTIAL phase

Table 4.2 Power consumptions of the `Controller/Sensor` component

Configuration	Avg	Std	Energy
SELFWATTS (default)	0.26 W	0.75 W	600 J
SELFWATTS *with fixed events*	0.20 W	0.44 W	465 J
SmartWatts	0.16 W	0.20 W	379 J

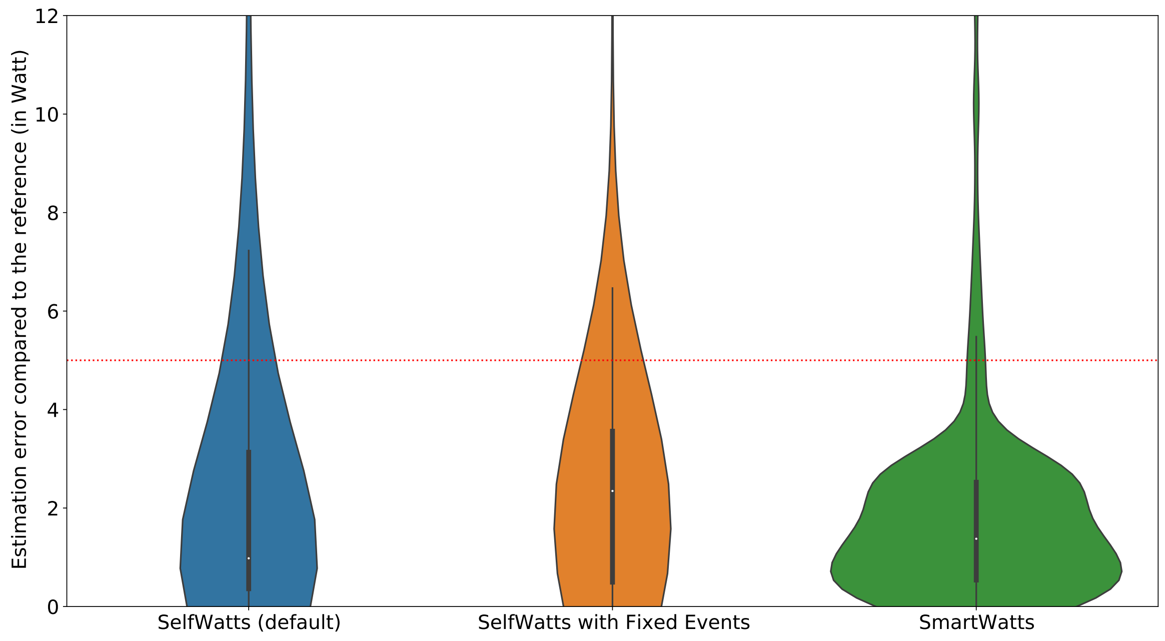

Fig. 4.7 Estimation errors ε for the PARALLEL phase

Table 4.3 Power consumptions of the `Formula` component

Configuration	Avg	Std	Energy
SELFWATTS (default)	0.64 W	1.53 W	1,458 J
SELFWATTS *with fixed events*	0.42 W	0.52 W	960 J
SmartWatts	0.33 W	1.65 W	817 J

0 to 200, by starting 10 new VMs every 10 seconds (vertical red lines refers to increments of 40 VMs). The consumption spikes observed in Figure 4.8 refer to periodic adjustments of the coefficients of the power model M when additional VMs are started, while the first consumption spike observed in Figure 4.8 refers to a change of the performance events used by the power model M. This is due to the fact that the node moves from 0 to 40 hosted VMs, which drastically change the execution context, but demonstrates that SELFWATTS succeeds to automatically adapt its power models. In the latter case, one can observe that SELFWATTS succeeds to free the memory associated to irrelevant performance events.

Interestingly, when reaching 200 VMs, the default configuration of SELFWATTS consumes $10W$ and $1.5W$ for the CPU and DRAM resources, on average, which represents a cost per VM of $0.06W$. Among the factors that can contribute to further reduce this overhead, one can mention the exploitation of fixed events (cf. Tables 4.2 & 4.3), the reduction of the monitoring rate β, or increasing the error threshold α.

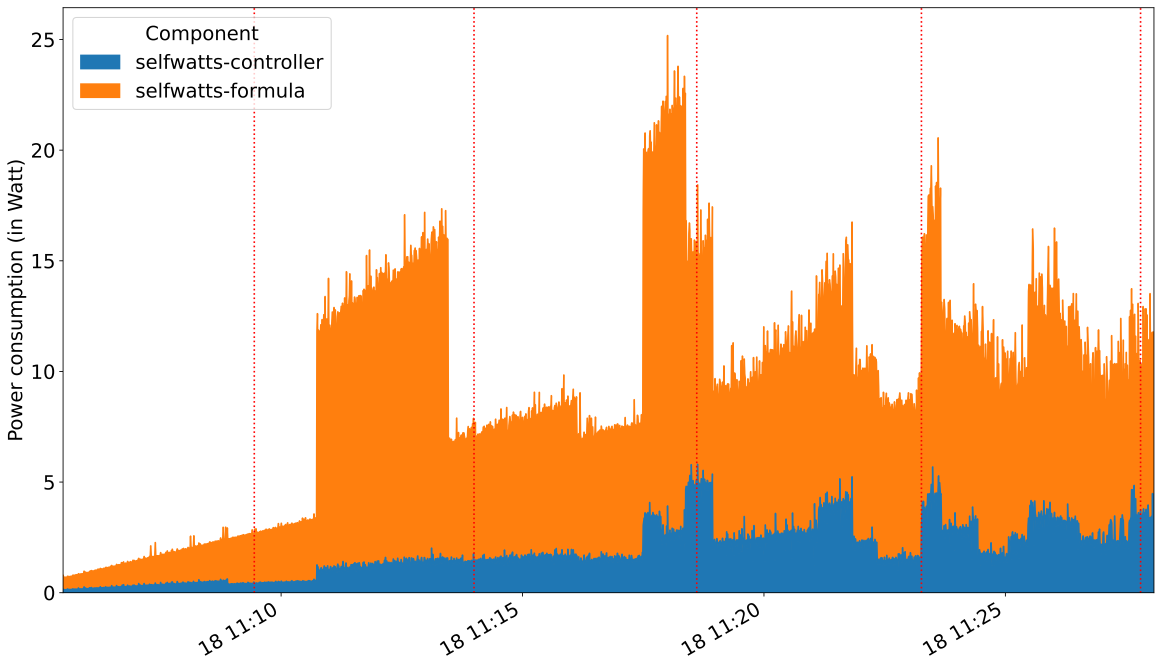

Fig. 4.8 CPU power consumption of the `Controller/Sensor` and `Formula` components

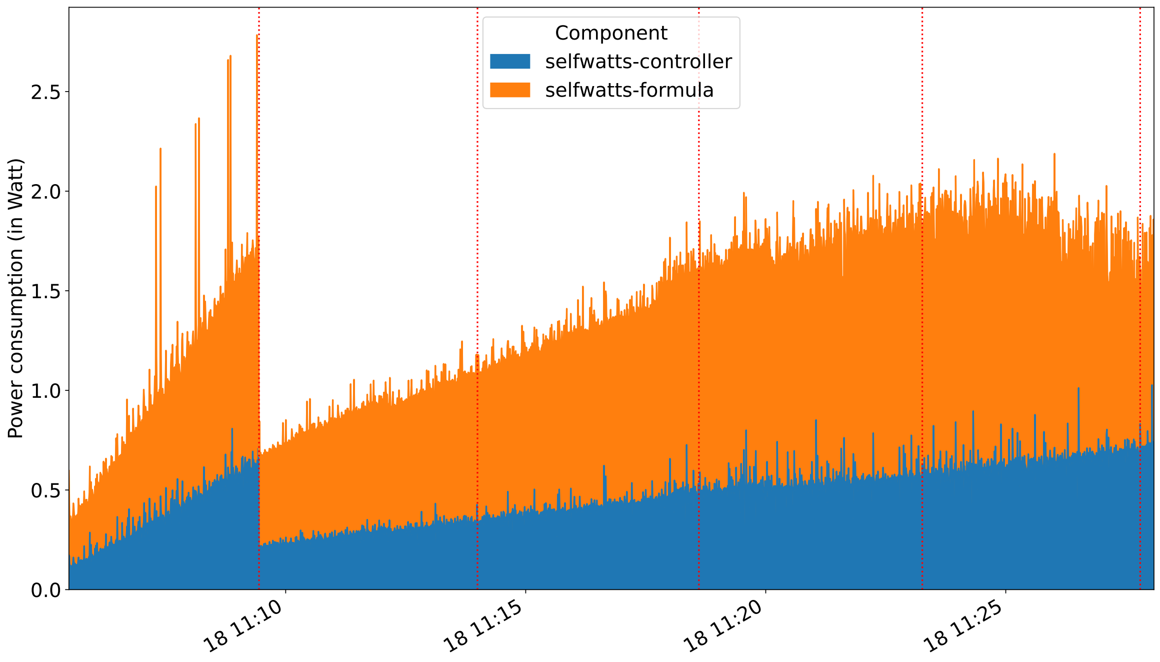

Fig. 4.9 DRAM power consumption of the `Controller/Sensor` and `Formula` components

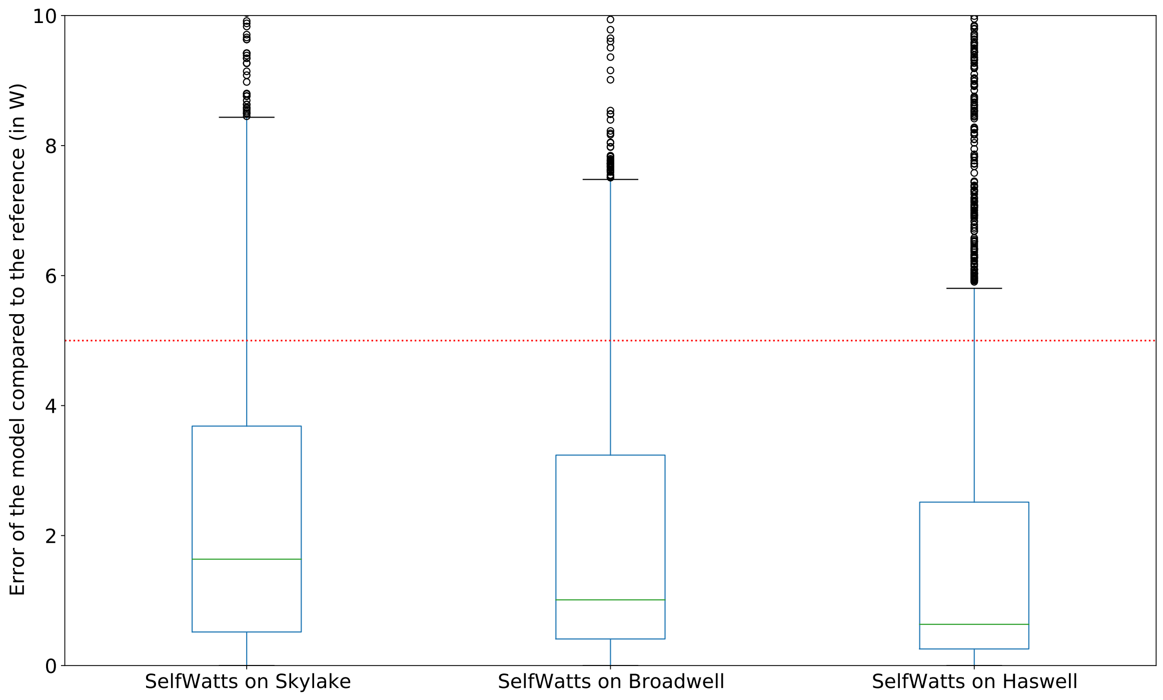

Fig. 4.10 Estimation errors ε for SELFWATTS on across different CPUs

With regards to the power consumption profiles of monitored applications, ranging from $25W$ to $125W$ (cf. Figures 4.3 & 4.4), we can conclude that SELFWATTS offers a lightweight solution to monitor the power consumption of virtual machines and containers.

Self-optimization

Finally, to assess the adaptability of SELFWATTS, we deployed the default configuration on three target architectures that exhibit a different set of performance events (cf. Table 4.1). In this mode, SELFWATTS offers a zero-configuration solution to automatically converge towards accurate power models for any target architecture, as reported in Figure 4.10. Our results show in particular that, no matter the target architecture, SELFWATTS succeeds to estimate the power consumption of the host machine with high accuracy, which contributes to estimate the power consumption at the scale of virtual machines and containers with higher confidence. We believe that this accuracy is particularly critical when further estimating the power consumption of guest software systems, as proposed by [12].

4.3.3 Lessons Learned & Perspectives

Beyond the high accuracy and the low overhead exhibited by SELFWATTS, we would like to share the lessons we learned from designing and implementing such a self-optimizing middleware solution.

First, the monitoring of performance events requires carefully considering the pitfalls related to the limited number of HWPC slots. While SELFWATTS detects the number of available HWPC slots that can be used to explore the relevant performance events, nothing prevents another co-located software system to monitor other performance events and incidentally impact the accuracy of SELFWATTS by triggering multiplexing at the level of the hardware performance counters.

Surprisingly, the randomization process introduced by the `Controller` component of SELFWATTS conducted to the inference of power models exploiting performance events that are different from the ones commonly identified by the community, yet achieving similar accuracy. Furthermore, due to the stochastic nature of the monitored environment, nothing prevents SELFWATTS to exploit power models based on a different set of relevant performance events when running the same workload on the same target architecture. Nevertheless, both our accuracy and overhead investigations show that the lack of convergence towards a single power model is not a limitation of our solution, contrary to state-of-the-art claims about the fact only a limited number of performance events can accurately capture the power consumption of a target architecture.

Unfortunately, the introduction of input feature transformers in the `Formula` component of SELFWATTS did not lead to expected results, as most of the inferred power models we manually analyzed rather exploit the raw input samples after completing the *Lasso* regression. We, nonetheless, believe that this negative result deserves to be mentioned as part of the lessons we learned to open discussions for the relevance of such methods commonly adopted in machine learning in the specific-case of online supervised training based on performance events.

Through the definition of an error threshold, our approach enforces a trade-off between the stability of the inferred power models and their accuracy (cf. Figure 4.7). This parameter α, therefore, indirectly controls the decision process of SELFWATTS to balance the exploration of a more accurate power model versus the exploitation of an acceptable power model, yet not optimal. While this problem is commonly known in reinforcement learning communities, we believe that it is particularly critical as our experimentation has shown that the exploration phases inevitably induce a power consumption overhead.

Finally, while SELFWATTS ambitions to support any target architecture, our experiments with an AMD EPYC 7301 (32 cores / 64 threads) has shown that, although the RAPL interface is being supported on latest Linux kernel versions, *i)* the RAPL support for the DRAM is still lacking and *ii)* the support for performance events remains immature compared to Intel-based processors. We nonetheless believe that future developments of AMD-related libraries will fix this limit in a near future.

Finally, beyond the specific case of power modeling we explore in the context of this contribution, we believe that the proposed architecture could also benefit to other case studies that require to downscale ground truth observations (here RAPL measurements) to the scale of individual Cgroups, by automatically correlating causal connections between global observations and more fine-grained activities. For example, the identification and monitoring of side-channel attacks in the domain of security could leverage our contribution.

4.4 Summary

Power consumption is a critical concern in modern distributed computing infrastructures, from HPC to data centers, which more and more aim to implement sustainable solutions to cope with environmental challenges raised by the massive deployment of software services. While current practices leverage tools to monitor the power consumption at a coarse granularity (*e.g.*, nodes, sockets), the literature still fails to propose generic power models, which can be easily deployed and used to estimate the power consumption of software artifacts in production with accuracy. This failure can be explained not only by the prohibitive cost of some solutions and models deployed on monitored hosts, but also the consideration of a static set of input features (*e.g.*, performance events) that may not be available on a specific target architecture, thus compromising the deployment of the monitoring solution.

In this chapter, we reported on a novel zero-configuration power meter, named SELFWATTS, that automatically selects the relevant performance events and continuously self-optimize the power models that can be used to deliver real-time power estimations with accuracy. Interestingly, we demonstrate that, no matter the target architecture, SELFWATTS does not require a prohibitive offline calibration phase to maintain a power model that can report the power consumption of software containers or *virtual machines* (VM). The experimental results we conducted highlights that SELFWATTS can estimate the power consumption of an unknown host with an average error of $2W$ (1.6 % of the TDP) for a monitoring cost of $0.06W$ per monitored VM.

Thanks to SELFWATTS, data center operators and users can easily deploy a software power meter that can be used to monitor the power consumption of their software containers, no matter their hardware constraints. This allows to removes barriers to the deployment of power-aware software solutions, in which the power consumption of software components is continuously monitored and optimized.

Chapter 5

*x*PUE: Extending Power Usage Effectiveness Metrics for Cloud Infrastructures

5.1 Introduction

As demonstrated in Section 2.3, while existing Energy Efficiency metrics are able to provide a global view of the energy consumption of a cloud infrastructure, they fail to support the energy profiling at a finer granularity: dealing with the software services that are distributed across such infrastructures. Modern *Data Centers* (DC) are continuously trying to maximize the *Power Usage Effectiveness* (PUE) of their infrastructure to reduce their operating costs, and eventually their carbon emission [22]. In the context of cloud providers, PUE is increasingly adopted and communicated as a *Key Performance Indicator* (KPI) reflecting the energy efficiency of the delivered solution. For years, reducing the PUE of data centers has become an active research area where actors compete to propose the most efficient cooling techniques, consider renewable energies as part of their electricity mix, and improve the utilization of IT resources. While this metric is mostly used to broadly communicate on the general efficiency of an infrastructure, it does not provide any insight on the energy efficiency of the software services that are deployed on top of it.

In this chapter, we introduce *x*PUE, an extension of the well known *Power Usage Effectiveness* (PUE) metric that allows operators and users to assess the energy efficiency of their infrastructure. Our approach, based on the combination of multiple energy efficiency metrics that are tailored to the different layers of the infrastructure, can be used at will depending on the required granularity of the analysis. The availability of the various metrics depends on the availability of the power measurements at the level of abstraction requested. For example, estimating the energy efficiency of a machine at least requires access to the energy consumption of the machine itself. However, we designed *x*PUE to be able to work by incorporating the existing efficiency data published by the data center operators, which is often the case for the PUE and CUE metrics.

We show that *x*PUE success to provide an in-depth view of the energy efficiency of the various hardware and software components of a cloud infrastructure. This contribution allows future work on the energy efficiency of cloud infrastructures, and more generally of distributed infrastructures, by providing a fine-grained view of the energy consumption of the software services that are deployed to help identify potential inefficiencies.

As stated in Section 1.3, the tools used by *x*PUE are freely available as open-source software to encourage the deployment at scale and to leverage the adoption and reproduction of our results. The key contributions can therefore be summarized as follows:

1. an extension of the *Power Usage Effectiveness* (PUE) metric to support a top-down analysis of energy hotspots on an infrastructure,
2. an implementation of the proposed extensions as instrumented approach completely based on open-source components,
3. an assessment of the proposed metrics on various hardware and software platforms under different contexts.

In the remainder of this chapter, we start by introducing this contribution (cf. Section 5.2). Then, we detail the implementation of *x*PUE as a real-time energy effectiveness metric for Cloud infrastructures (cf. Section 5.3), and we empirically assess its validity on multiple scenarios (cf. Section 5.4). We conclude and provide some perspectives for this work in Section 5.5.

5.2 Contributions

*x*PUE groups a family of PUE-related metrics that can be easily adopted by cloud providers, and more generally service providers, to estimate the power usage effectiveness of their infrastructure in the deep, including the software platforms they may operate. *x*PUE extends the state-of-the-art PUE and SPUE coverage by delivering insights beyond the power supply of physical servers. All *x*PUE metrics can be explored *post mortem* to audit a given architecture on a given period (*e.g.*, 1 year of operation), or in real-time to monitor the impact of operational conditions. Real-time monitoring of *x*PUE metrics allows cloud operators to quickly understand the conditions under which these indicators are optimal, or critical, hence offering them actionable feedback to immediately identify the most efficient levers to consider and optimize.

5.2.1 Overview of *x*PUE

In this chapter, we specifically leverage the SPUE introduced by Barroso *et al.* [4] to introduce the VPUE metrics and highlight resource usage effectiveness at the scales of hardware and virtual layers, respectively. Figure 5.1 depicts the complementarity of *x*PUE metrics with state-of-the-art metrics, including PUE, SPUE, CUE and WUE. One can observe that, while PUE, WUE, and CUE are delivering an in-breath coverage of the resource usage effectiveness of DC along different perspectives—energy, water, and carbon, respectively—SPUE and *x*PUE brings more in-depth insights by investigating how these resources are consumed within servers and ultimately cloud services. As the energy consumption of the hardware servers and software services is proportional to their usage, they can exhibit significant variability when it comes to energy efficiency. In the context of this chapter, we are therefore interested in investigating the resource usage effectiveness of cloud services, by covering hardware and software layers. By doing so, we intend to expose the real effectiveness of cloud infrastructures and to contribute to a more transparent exposure of how much energy is consumed by cloud providers for each functional unit, whenever operating a *Metal-as-a-Service* (MaaS), *Infrastructure-as-a-Service* (IaaS), or a *Container-as-a-Service* (CaaS) offer. By focusing on the hardware and software layers that are not covered by existing metrics, we intend to raise new environmental challenges for cloud infrastructures, by encouraging the ecosystem to maximize the end-to-end power usage effectiveness of cloud services, hence going beyond the sole effectiveness of DC buildings, as covered by the PUE.

In the remainder of this section, we first remind the SPUE (Section 5.2.2) and formalize the VPUE metrics (Section 5.2.3), before illustrating how they can be combined in a compound metric CPUE (Section 5.2.4) and applied to state-of-the-art resource usage metrics (Section 5.2.5).

5.2.2 SPUE: Assessing Cloud Servers Power Usage Effectiveness

The metric SPUE (standing for "*server PUE*") aims to estimate the power usage effectiveness at the scale of a server [4]. This metric is particularly relevant in the cloud when operating a *Metal-as-a-Service* (MaaS, also known as *bare-metal*) offer, which consists in delivering a hardware server to the customer. In this context, the cloud provider is in charge of hosting and eventually assembling, the delivered server. While the PUE stops at the power outlet of hosted servers, the SPUE metric intends to dive into the integrated components to capture

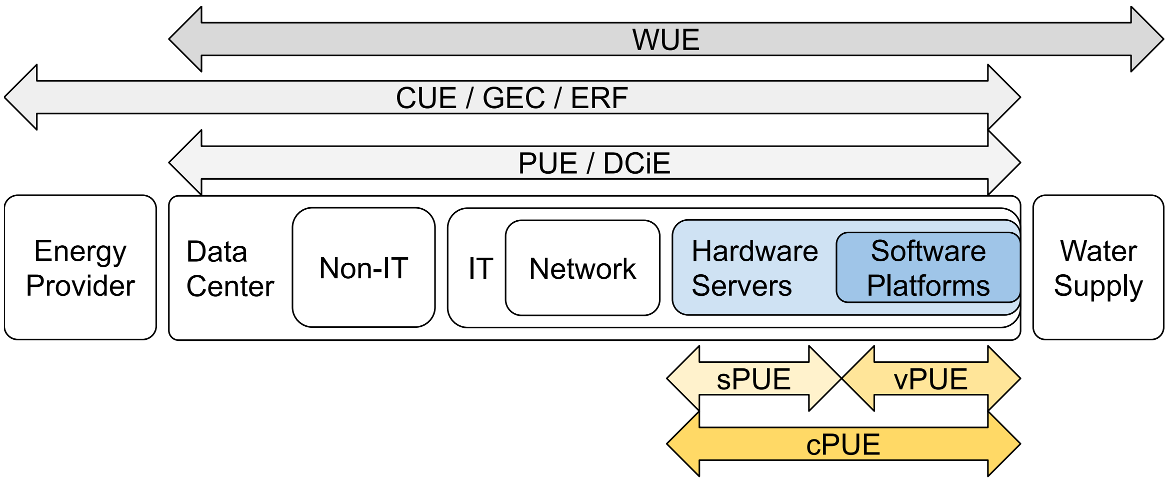

Fig. 5.1 Coverage of *x*PUE metrics.

the overhead imposed by the cooling systems and power supplies inside the server. To do so, we compute the ratio between the energy consumed by the IT equipment and the energy directly consumed by the server components (incl. CPU, GPU, memory, disk, controllers, etc.), as follows:

$$\text{sPUE} = \frac{\sum Energy(\mathsf{IT})}{\sum Energy(\mathsf{servers})} \tag{5.1}$$

This metric indicates how much energy is consumed by the physical server for each unit of computation delivered by the CPU and related components. By formalizing the sPUE, we are interested in highlighting the efforts spent by cloud providers to deliver energy-proportional bare-metal solutions—*i.e.*, limiting the cost of cooling components and optimizing the supply of direct current. We believe that this additional dimension is important to capture, as there is no guarantee that the PUE reflects this overhead.[1]

sPUE covers general-purpose computations that can be delivered to cloud customers, but also exploited internally to operate a cloud platform. However, no matter their nature—being IaaS, CaaS, PaaS, etc.—these cloud platforms impose a control plane to orchestrate the deployment of cloud services, which is another resource overhead that we further capture with the vPUE.

5.2.3 vPUE: Assessing Cloud Services Power Usage Effectiveness

The metric vPUE (standing for "*virtual PUE*") dives into the software layers of cloud infrastructures by investigating the cost of operating large and complex software platforms, such

[1]The PUE might eventually reflect, indirect, side-effects of server emissions.

as OPENSTACK[2] for a IaaS or KUBERNETES[3] for a CaaS. These cloud solutions generally share two key concepts: *virtualization techniques* to control access to the computational resources, and *control planes* to deploy and manage the hosted services. In this context, we are interested in investigating how much energy is consumed by the cloud infrastructure for each unit of computation delivered to the hosted services, no matter their nature—*i.e.*, virtual machines or containers. To capture this information, we define the VPUE as follows:

$$\text{VPUE} = \frac{\sum Energy(\mathsf{servers})}{\sum Energy(\mathsf{services})} \tag{5.2}$$

where servers refer to all the server components required to operate the services hosted by the considered cloud platform. The resulting ratio is intended to measure the overhead imposed by the virtualization techniques and the associated control planes, depending on the current usage.

One should also keep in mind that, unlike hardware layers, software layers can be stacked by cloud providers and/or their customers. For example, the deployment of a KUBERNETES cluster atop virtual machines hosted by an IaaS infrastructure is commonly adopted by practitioners to offer more flexibility when it comes to adjusting resource usage.

5.2.4 CPUE: Applying *x*PUE Metrics to Cloud Infrastructures

The metrics SPUE and VPUE can then be combined in the metric CPUE (referring to "*cloud PUE*"), which is intended to capture the end-to-end power usage effectiveness of cloud infrastructures. To compute this compound metric, one needs to compute the product of *x*PUE metrics, depending on the spectrum of the deployed infrastructure, as follows:

$$\text{CPUE} = \prod_{x \in L} x\text{PUE with } L \text{ the selected cloud layers} \tag{5.3}$$

The list of selected layers, L, depends on the context and the owner of the infrastructure. For example, a IaaS provider operating an OPENSTACK cloud platform will compute $\text{CPUE} = \text{SPUE} \times \text{VPUE}(\mathsf{OpenStack})$. Then, a KUBERNETES platform hosted on premise will rather be reported as $\text{CPUE} = \text{SPUE} \times \text{VPUE}(\mathsf{Kubernetes})$. Finally, because of the recursive property of virtualization techniques, one can imagine computing $\text{CPUE} = \text{SPUE} \times \text{VPUE}(\mathsf{OpenStack}) \times \text{VPUE}(\mathsf{Kubernetes})$ to capture the end-to-end PUE a multi-layers service platform leveraging several cloud technologies.

[2]https://www.openstack.org
[3]https://kubernetes.io

Beyond being a product of *x*PUE metrics, any CPUE metric can also be multiplied by the PUE of the DC hosting the platform services as follows:

$$\text{GPUE} = \text{CPUE} \times \text{PUE} \tag{5.4}$$

which is reported as the "*global PUE*" (GPUE) revealing, for each unit of computation performed by a cloud service, how much energy is effectively consumed by the whole DC hosting this service. Given that *x*PUE metrics share the same properties as the PUE—*i.e.*, the ideal value is 1.0—one can observe that any waste of resource in any of the covered layers may severely impact the global PUE of the infrastructure, thus challenging the cloud ecosystem to pay attention to the effectiveness of their solutions.

5.2.5 Revisiting State-of-the-Art Metrics with CPUE

Beyond the GPUE, one can also revisit the state-of-the-art metrics to consider their global impact, beyond the DC building optimizations. For example, the GDCiE metric can be extended with the CPUE as follows:

$$\text{GDCiE} = \frac{\text{DCiE}}{\text{CPUE}} \tag{5.5}$$

Similarly, the *Carbon Usage Effectiveness* (CUE) and *Water Usage Effectiveness* (WUE) metrics can be more accurately reported by cloud infrastructures as follows:

$$\text{GCUE} = \text{CPUE} \times \text{CUE} \tag{5.6}$$

$$\text{GWUE} = \text{CPUE} \times \text{WUE} \tag{5.7}$$

Overall, we claim that cloud providers and, more generally, owners of software service infrastructures should more systematically compute and share the CPUE of their solution to demonstrate that they do not waste the resources saved by an efficient PUE. Additionally, the GCUE and GWUE are providing a complementary and important perception on the emissions of cloud infrastructures, by sharing more accurate information about their environmental impact.

In the following sections, we report on different experiments we deployed to illustrate the critical impact of hardware and software layers in the cloud ecosystem, hence calling for the development of more energy-efficient cloud platforms, going beyond the advertisement of their sole PUE.

5.3 Implementation Details

The implementation of the *x*PUE metrics we propose requires reporting power measurements at a finer granularity than a power outlet. To do so, we leverage the POWERAPI toolkit presented in Section 3.2.2 to propose runtime support for our *x*PUE metrics.

For the global energy measurements required by the SPUE—*i.e.*, $\sum Energy(\mathsf{IT})$ in Equations 5.1—we leverage hardware power meters plugged into the power supply and *Intelligent Platform Management Interface* (IPMI) for the global power measurements of the servers. IPMI measurements have a low refresh rate and a low accuracy, it is preferable to use hardware power meters whenever possible. When the server combines multiple power supplies, the power measurements of all active power supplies are aggregated.

For hardware-specific measurements (CPU, DRAM, GPU...) required by the SPUE and VPUE—*i.e.*, $\sum Energy(\mathsf{servers})$ in Equations 5.1 and 5.2—we use the energy monitoring interface(s) exposed by the hardware.

Regarding software power measurements—*i.e.*, $\sum Energy(\mathsf{services})$ in Equation 5.2—we use the SMARTWATTS power meter presented in Chapter 3, which infers automatically power models from hardware measurements and disaggregates power consumption among software processes. SMARTWATTS supports power estimation both at the granularities of virtual machines and containers as well as it succeeds in estimating CPU and DRAM power consumptions in real-time with high accuracy.

To differentiate the hosted cloud services from the control plane services, we can select and tag pre-defined groups of services. For example, in a KUBERNETES cluster, some services are directly related to the *infrastructure*, like the container runtime, control plane, networking, and monitoring services. Subgroups can also be defined for the services: one can compute dedicated VPUE for network and monitoring services to further analyze their energy efficiency.

5.3.1 Implementing the *x*PUE Formulas

We implemented each *x*PUE metric as a Python function that can be integrated as a dedicated formula in the open-source library POWERAPI. The POWERAPI toolkit exposes a software agent that consumes input measurements from various sources (databases, message queues), processes the samples, and produces estimations through the same or another database(s). It is built to be extensible and allows to easily fit the deployment environment. We also leverage the *de facto* standard libraries in Python: PANDAS[4] for the samples manipulation and analysis, as well as SCIPY[5] for the computation of the *x*PUE estimations.

This results in the development of a family of dedicated *x*PUE formulas atop POWERAPI that continuously estimate SPUE and VPUE indicators and can even be further combined to report on the CPUE and GPUE compound metrics in real-time, as depicted in Figure 5.2. As *x*PUE metrics rely on power measurements coming from multiple sources (Intel RAPL, IPMI, hardware, and software power meters) that are not synchronized, we resample the power measurements to a common time base. By default, *x*PUE formula detects the shortest time window possible to resample the PANDAS *DataFrame*. Finally, our implementation of *x*PUE leverages a database to store the power measurements received from hardware and software power meters. We choose to leverage the publisher-subscriber pattern to handle large-scale deployments. Thus, all power measurements are sent to a MONGODB capped collection. These measurements are then handled by the formula components and *x*PUE metrics are then stored in the database for further analysis.

5.3.2 Deploying the *x*PUE Metrics

To simplify the deployment process, *x*PUE is available as containers, which provide an environment-agnostic deployment and ease the lifecycle management of its related services. The formula components can be deployed on any host of the cluster, or a remote server, as it only requires access to the power measurements through a message queue to work. For example, one can use a MONGODB instance as a message queue to communicate the power-meters measurements through a publish-subscribe pattern and then store the *x*PUE metrics into an INFLUXDB *Time Series DataBase* (TSDB). The *x*PUE metrics are exposed as GRAFANA dashboards[6] for environment/service-specific metrics reporting and real-time/offline analysis.

5.4 Empirical Validation

This section builds on our implementation of *x*PUE to study the factors that contribute to improving or degrading the *x*PUE metrics we introduced. We start by investigating the impact of SPUE on different hardware configurations (cf. Section 5.4.2), then exploring the VPUE in the context of a IaaS infrastructure (based on OPENSTACK) and a CaaS infrastructure

[4]https://pandas.pydata.org
[5]https://www.scipy.org
[6]https://grafana.com

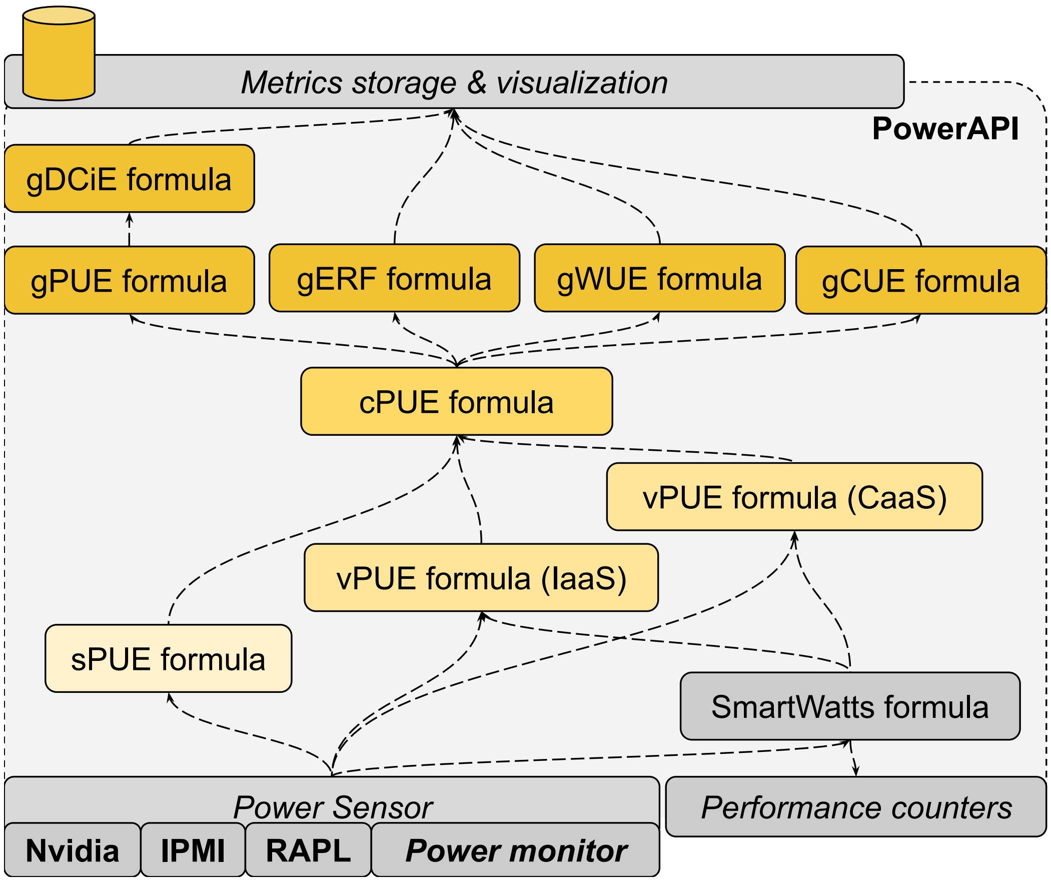

Fig. 5.2 Deployment of *x*PUE

Table 5.1 Testbed hardware settings

Provider	Grid'5000			OVHcloud	
Model	Dell PowerEdge R640	Dell PowerEdge R7525	Dell PowerEdge R640	Intel bare-metal server	AMD bare-metal server
CPU	Intel Xeon Gold 5220	AMD EPYC 7352	Intel Xeon Gold 5218	Intel Xeon Silver 4214R	AMD EPYC 7413
Generation	Cascade Lake	Zen 2	Cascade Lake	Cascade Lake	Zen 3
Socket(s)	1	1	2	2	1
Cores per socket	18	24	16	12	24
Threads per socket	36	48	32	24	48
Memory	96 GiB	128 GiB	384 GiB	32 GiB	64 GiB
TDP	125 W	155 W	125 W	100 W	180 W
Cooling	Air			Water	

(based on Kubernetes). We conclude by illustrating the cPUE in the context of a CaaS deployed in a IaaS, a widely-adopted architecture in the cloud industry (cf. Section 5.4.4).

In this section, we assess the accuracy of *x*PUE to evaluate the *Power Usage Effectiveness* into deeper layers of Kubernetes and OpenStack based cloud infrastructures.

5.4.1 Evaluation Methodology

We follow the experimental guidelines reported by [55] to enforce the quality of our empirical results. In this case, the experiments are executed on real environments, we fully share the machine specifications and the version and configuration of the software used. We choose to evaluate our contribution on the OpenStack and Kubernetes platforms, which are widely used by the industry. For the sake of reproducible research, *x*PUE, the necessary tools, deployment scripts, and resulting datasets are open-source and publicly available on GitHub.[7]

Hardware Settings

Most of our experiments are deployed on the Grid'5000 testbeds infrastructure [3], which is a large-scale and flexible testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing—including cloud, HPC, Big Data, and AI. We deploy our experimental infrastructure on 5 nodes of the cluster gros located in the site of Nancy. The description of the considered servers (cf. 2 first columns in Table 5.1). This cluster is particularly interesting as each node has its power supply monitored by a hardware power meter to monitor the power consumption of the node with high accuracy and at a high frequency.

We also consider the provisioning of additional bare-metal servers from a production-scale cloud infrastructure provided by OVHcloud.[8] We chose this cloud provider as all the

[7]https://github.com/powerapi-ng
[8]https://www.ovhcloud.com/

hosted servers are cooled using an advanced water cooling system, which may contribute favorably to the SPUE, compared to traditional air cooling.[9] Furthermore, OVHcloud proposes a MaaS offer with a wide diversity of hardware architectures (including Intel and AMD processors) and access to server-scale power measurements.

Software Settings

All machines of our experiment are using the Ubuntu 20.04 LTS Linux distribution with a kernel version `5.4.0-121-generic`, where only a minimal set of daemons are running in the background.

For the KUBERNETES cluster, the deployment is done using kubeadm and the version deployed is the 1.21. The container runtime is Containerd version 1.6.6 and the *Container Network Interface* (CNI) deployed is FLANNEL[10] version 0.18.1.

For the OPENSTACK cluster, the deployment is done using MICROSTACK,[11] which deploys OPENSTACK version Ussuri through the SNAP[12] package manager. This allows us to quickly deploy an OPENSTACK instance in self-contained packages that support individual monitoring through Linux Cgroups. This choice considerably eases the deployment and the reproducibility of our results, while remaining representative of real-world deployments.

Input Workloads

For both deployments, we use a state-of-the-art benchmark tool for Linux, STRESS-NG,[13] to simulate various resource-intensive workloads that stress various parts of the system. Therefore, during each experiment, containers and VMs will be started with a random resource workload that will be maintained until it is stopped. This allows us to generate a base workload on the infrastructure and to stress multiple parts of the software services and underlying hardware.

Power Meters

Hardware measurements. To monitor the power consumption of the servers, we use the available hardware power meters attached to the input of the power supply of every machine in the cluster. The measurements are automatically reported to an INFLUXDB at a frequency 50 measurements per second, which is more than required for our experiments.

[9]https://blog.ovhcloud.com/water-cooling-from-innovation-to-disruption-part-i/
[10]https://github.com/flannel-io/flannel
[11]https://microstack.run
[12]https://snapcraft.io
[13]https://launchpad.net/stress-ng

Software measurements. *SmartWatts* requires the deployment of two components, the `Sensor` who monitors the *Hardware Performance Counters HwPC)* and needs to be deployed on every node, and the `Formula` component who computes the power estimations. In all our experiments, we configure the `Sensor` components to report on power estimations once per second ($\beta = 1\,\text{Hz}$), and the FORMULA component with an error threshold of $\alpha = 5W$ at the scale of the CPU package (`PKG`).[14]

5.4.2 sPUE Experiments

Server setting. To evaluate the relevance of sPUE, we run our benchmark designed to stress twice the maximum CPU load of the Intel Xeon Gold server. This aims to investigate the behavior of the hardware infrastructure in a situation of resource over-commitment. Figure 5.3 reports on the evolution of the sPUE over time, when increasing CPU workload. One can first observe that idle servers seriously degrade the energy efficiency of cloud infrastructures, with an sPUE > 4. Beyond this first observation, the sPUE tends to decrease with higher CPU usage, no matter the number of concurrent containers, as acknowledged by Figure 5.4. With an average value of 3.1 throughout experimentation, one can observe that the optimal and lowest value reached for the sPUE of such a standard server caps at 2.7, which is high above the values commonly communicated for PUE. As the sPUE should be combined with the PUE of any cloud provider to report the GPUE reflecting the physical power consumption induced by any Joule consumed by the computing units (cf. Equation 5.4), it can only degrade the value of the standard PUE. These observations thus challenge cloud infrastructures and MaaS providers to deliver more energy-efficient servers by investing in energy-proportional techniques to power linearly a server with the triggered activities. This is particularly critical as DC and cloud providers are known to be under-utilized, which let us expect higher sPUE in production than the one we measured in our testbed.

GPU impact. Unfortunately, none of the server configurations of our testbed did include any GPU (or other hardware accelerators) to investigate the impact of such computing units. Nonetheless, our *x*PUE models can apply to any computing units, as long as one can obtain energy measurements at the required granularity (*e.g.*, CPU and DRAM measurements provided Intel RAPL). Regarding the GPU, one can use `nvidia-smi` to collect energy measurements from NVIDIA cards and add the reported values to the servers part of Equation 5.1. This challenge technology providers to implement and expose fine-grained energy

[14]https://01.org/blogs/2014/running-average-power-limit-\T1\textendash-rapl

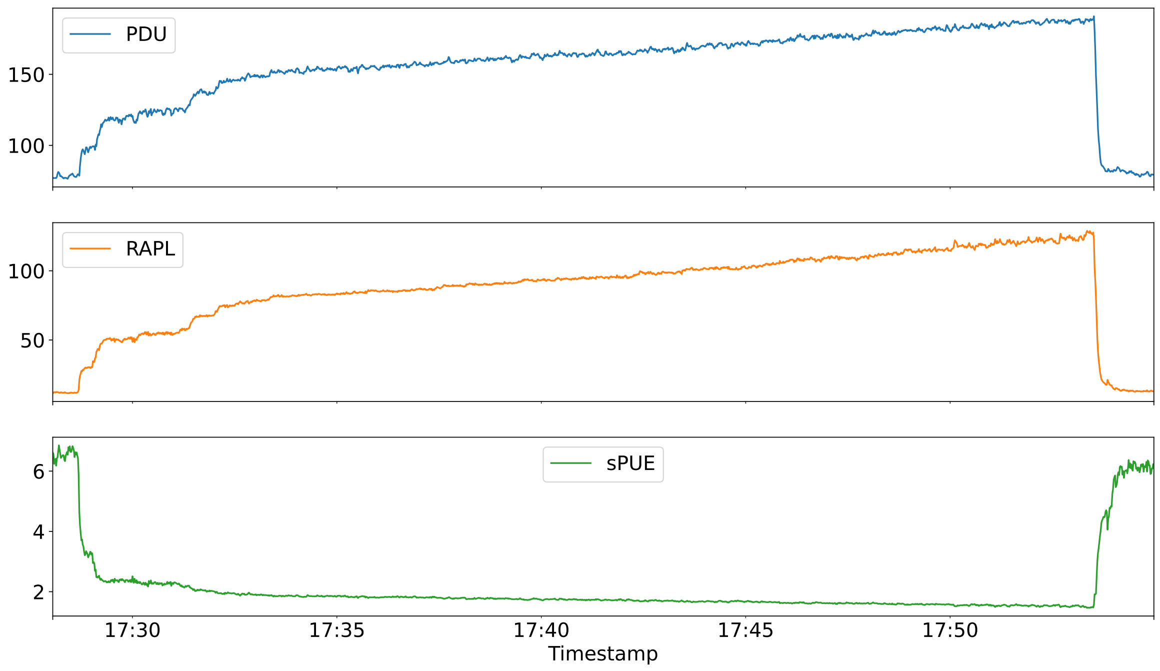

Fig. 5.3 Evolution of the sPUE over time and increasing workload

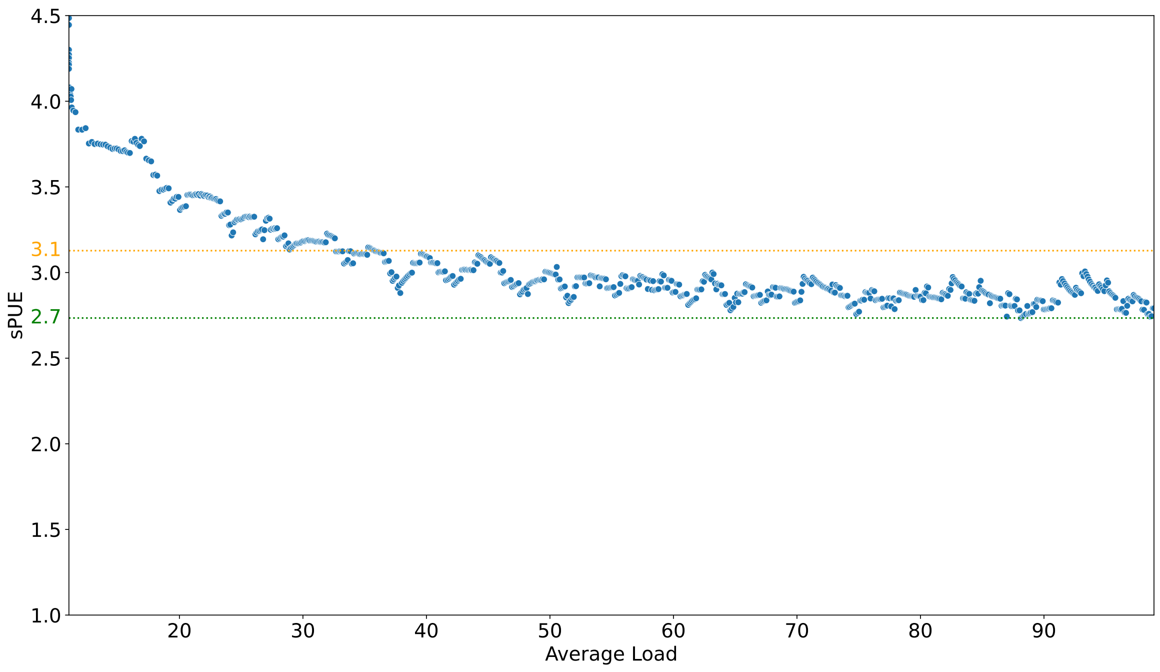

Fig. 5.4 Correlation of the sPUE and the CPU average load

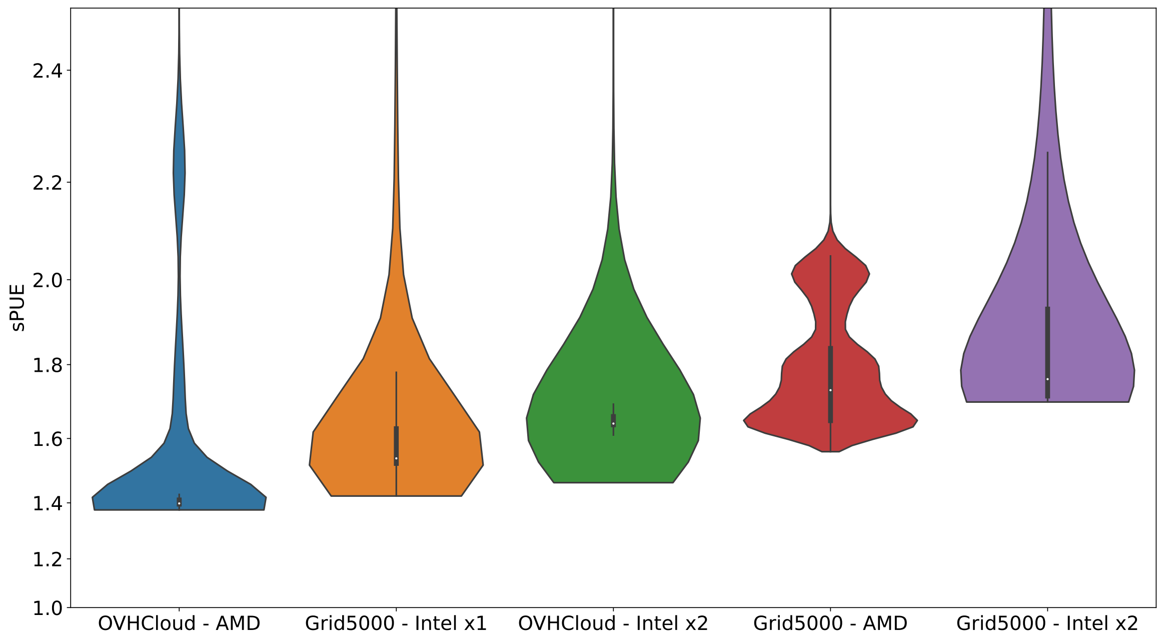

Fig. 5.5 Comparing the sPUE of all the hardware configurations under-test.

interfaces—like Intel, AMD, and NVIDIA recently did—in order to help the cloud ecosystem to better diagnose the causes of energy wastes and related opportunities for improving the efficiency of cloud infrastructures.

Hardware impact. To further investigate this hardware overhead of servers, we hypothesize like [4] that the sPUE might be influenced by the CPU architecture and the air cooling system of the server. Therefore, using the same input workload as in the previous experiment, we estimate the sPUE of Intel and AMD servers that are deployed in Grid'5000 (based on air cooling) and OVHcloud (based on water cooling) infrastructures (cf. Table 5.1). While water cooling systems directly contribute to improving the PUE of the DC, they may also indirectly impact the sPUE by removing the CPU/GPU fans from the server frame, hence saving energy consumption required by the equipment embedded in most of the servers. Figure 5.5 thus depicts the distribution of sPUE for each hardware configuration of Table 5.1 as a violin plot. Each configuration is stressed in the same conditions, by executing a workload of twice the maximum CPU load of each configuration. One can observe that the major factor contributing to effectively improving the sPUE is related to the adoption of a water cooling system, while the CPU architecture has a lower impact on the sPUE.

Other impacts. While we could not change the AC-DC power supply of our testbed or include alternative supply designs, we believe that the sPUE can also capture the efficiency

Table 5.2 Cluster-wide sPUE statistics.

Platform	**min**	**max**	**mean**	**median**
KUBERNETES	2.9	22.6	5	3.8
OPENSTACK	2.9	27.5	6.5	4.4

of this hardware component and contribute to adopting a more energy-efficient solution. We are also confident that this separation of concerns could be captured by a partial sPUE (pHPUE), inspired by the *partial PUE* (pPUE) [22], to isolate the overhead imposed by this hardware component.

One can therefore observe that optimizing the sPUE does not only require adopting energy-saving strategies to power and cool down server components but also maximizing the utilization of provisioned resources. This observation thus challenges the hardware configuration of servers to be carefully sized to the closest number of CPU threads, and other hardware components, which are required to support a target workload. In this context, elasticity mechanisms should be deployed at the hardware level by cloud infrastructure to implement energy proportionality and deal with the variability of workloads, hence preventing the over-provisioning of resources that require to be kept always on.

Cluster setting. Beyond single node deployments, cloud infrastructures are often considered to provision a cluster of several nodes that are then assembled to deploy a IaaS (*e.g.*, OPENSTACK) or CaaS (*e.g.*, KUBERNETES) platform. These clusters are typically composed of, at least, a control node and several worker nodes. We, therefore, consider the injection of a cluster-wide input workload to both OPENSTACK and KUBERNETES platforms, deployed atop 5 Intel Xeon Gold 5220 servers, to study the average sPUE at the scale of a cluster. Table 5.2 and Figure 5.6 compare the sPUE distribution of all the nodes involved in the cluster for both experiments. Similarly to the case of idle servers, idle platforms may be the root cause of a critical sPUE observed at the scale of a cluster, with observed factors above 20. Such situations typically witness the over-provisioning, and under-allocation, of a cloud infrastructure that allocates much more computing resources than required. While the optimal sPUE observed for a cluster of 5 nodes reaches 2.9, one can observe that even a stressful scenario like one of our scenario results in an average sPUE of 5 to 6.5—depending on platforms—thus highlighting the critical impact of the power usage efficiency of the nodes composing the cluster required to host a cloud infrastructure.

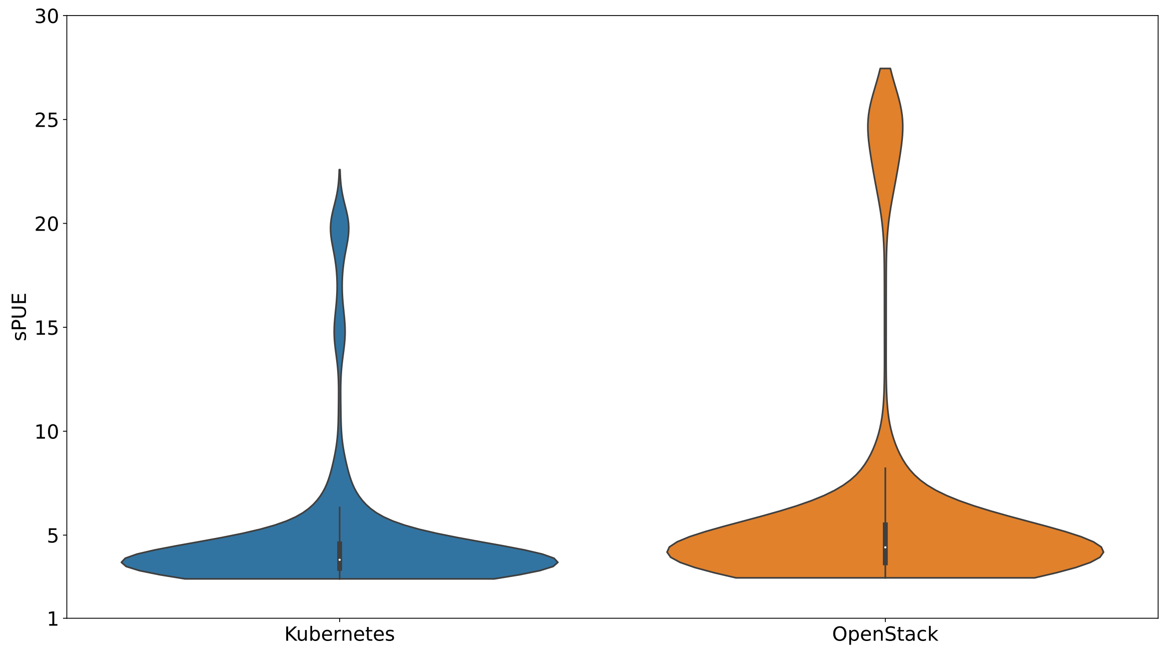

Fig. 5.6 Comparing the sPUE of a cluster used to host OpenStack and Kubernetes platforms.

Network impact. In the context of a distributed setting, like a cluster, the question of including network equipment might arise. As part of our experiment, we consider the impact of the network—*i.e.*, a single *Top of Rack* (ToR) switch in our context—as part of this cluster setting, assuming that a switch is required to connect several nodes.[15] However, the power consumption of most hardware network equipment (switch, routers, etc.) is known to be stable, no matter the workload (115.3 W in the context of our ToR switch), thus we advocate including the energy consumption of network equipment within the IT part of the sPUE, and not as part of the servers (cf. Equation 5.1). The motivation for doing such is that sPUE could be easily reduced by adding more and more network equipment in the servers part, which would go against the objective of optimizing the energy efficiency of the overall cloud infrastructure.

Control plane impact. Then, we compute the sPUE of the control node and the worker nodes separately for each configuration (cf. Table 5.3 and Figure 5.7). One can observe that no matter the cloud platform, the sPUE of the control node is always higher than the sPUE of the worker nodes, as the control plane is consuming fewer resources than the worker nodes, on average. One can nonetheless observe that an OpenStack control reports a slightly better sPUE on average than its worker nodes. This can be explained by the consumption

[15] Production-scale cloud infrastructures may involve multiple ToR switches to ensure failover.

Table 5.3 Control plane impact on SPUE.

Node	Platform	min	max	mean	median
control node	KUBERNETES	2.9	4.4	4	4
	OPENSTACK	2	6.2	3.3	2.9
worker nodes	KUBERNETES	1.4	6.7	2.1	1.8
	OPENSTACK	1.7	14.9	3.5	2.8

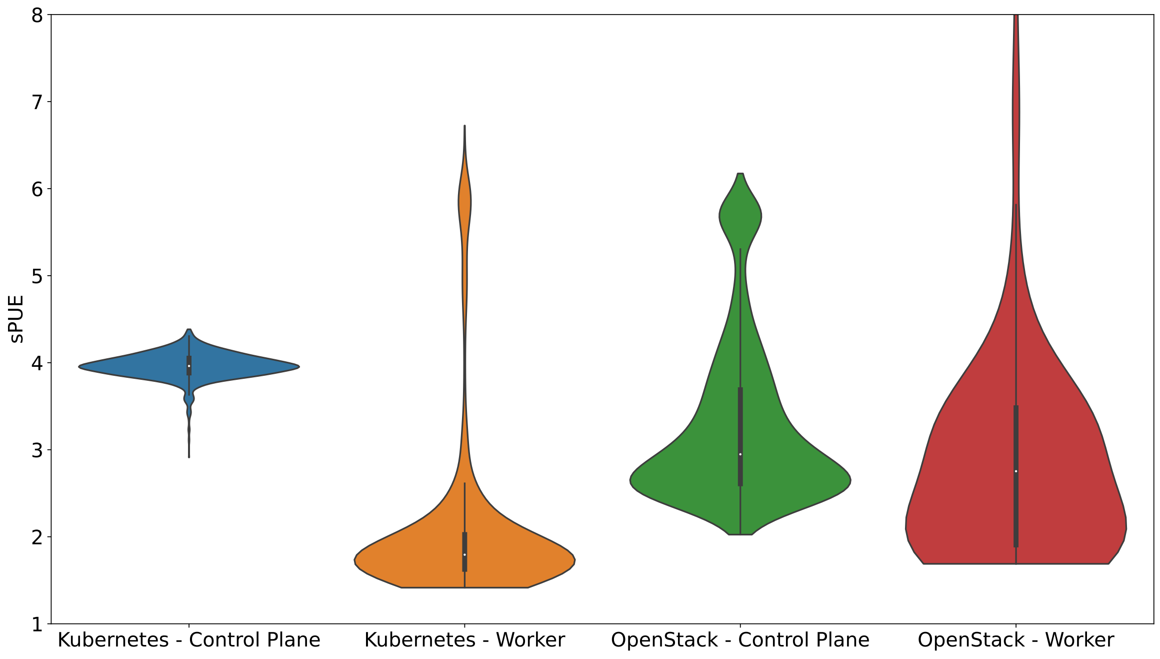

Fig. 5.7 Comparing the SPUE of control & workers nodes for OPENSTACK & KUBERNETES platforms.

of this control node, which tends to be relatively high compared to other nodes, due to the number of control services that are involved in the OPENSTACK platform. To reduce the impact of the control plane on the SPUE, one should therefore consider maximizing the number of active worker nodes and consider the deployment of carefully sized control nodes, involving potentially smaller servers.

One can also observe that the SPUE of KUBERNETES and OPENSTACK platforms seem to differ, which highlights that both technologies are not stressing the hardware components in the same way. To better understand the root cause of these differences, we further explore the VPUE of both platforms in the following section.

5.4.3 vPUE Experiments

We then move to the study of vPUE, first in the context of an IaaS platform, based on OpenStack, and then on a CaaS platform, based on Kubernetes. Through the following experiments, we intend to study the resource overhead imposed by the software platform and all the services it provides to deploy and control VM and/or containers. We believe that such software layers represent another key efficiency factor to carefully consider when delivering cloud infrastructures. Furthermore, unlike hardware layers, software layers can be embedded to deliver, for example, a CaaS platform atop IaaS. Thus, considering the impact of such common practices is also another insightful feedback that we intend to cover thanks to the vPUE indicator.

Estimating the vPUE of OpenStack

Platform setting. To estimate the vPUE of OpenStack, we used the same input workload as in the sPUE experiments, namely, we run a benchmark designed to start twice the maximum CPU load of the provisioned cluster. This aims to investigate the efficiency of the cloud infrastructure ranging from an idle state to a situation of resource over-commitment. We use a separate control node for hosting cluster-wide OpenStack services in addition to services that are required to be deployed in the worker nodes. The CPU and DRAM overcommitment parameters are kept to default: `16:1` and `1.5:1`, respectively. Each allocated VM uses a profile `m1.exp`, which consists of 1 vCPU and 256 MB of DRAM. Given that we use 4 worker nodes, summing to 144 threads and about 384 GB of DRAM, we can allocate—in theory—up to $2,304$ vCPUs and 576 GB of vRAM, which represent about $2,000$ VMs with profile `m1.exp`. However, we stop our benchmark when reaching 288 VMs, which represent twice the physical threads made available at the scale of the cluster.

The vPUE is computed as the ratio of the total energy consumption of all the cluster services (including VM) to the energy consumption of hosted VM. As introduced in Section 5.3, we estimate the energy consumption of individual VM by using the SmartWatts software-defined power meter.

Overcommitment impact. Figure 5.8 illustrates the evolution of the vPUE over time when increasing the number of hosted VM. One can observe that the more VM, the better vPUE, as one could expect.

More precisely, Figure 5.9 shows that the vPUE converges towards its optimal value when reaching more than 100 hosted VMs, which roughly corresponds to the total amount of physical resources (CPU & memory) available in the cluster we provisioned. This average

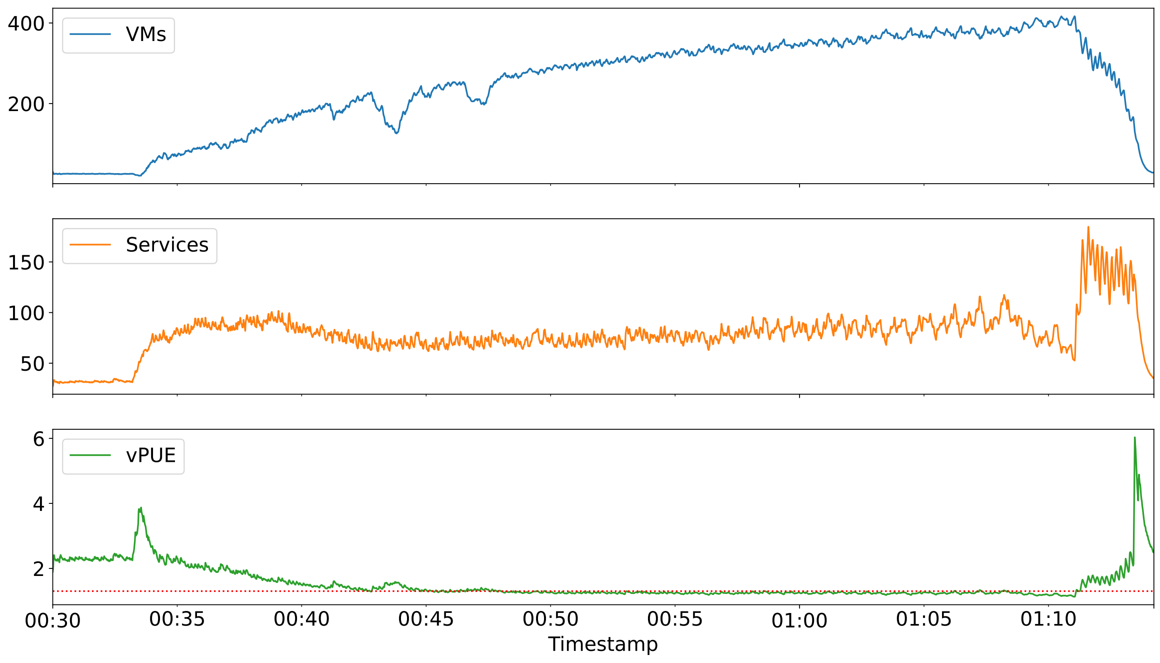

Fig. 5.8 Evolution of the vPUE of OPENSTACK when increasing the number of hosted VMs.

value is estimated to 1.3 in the context of our deployment of OPENSTACK, involving on 1 control node and 4 worker nodes. This first part of the experiment demonstrates that, as does the SPUE, the optimal VPUE is reached when fully loading the worker nodes, which encourages the optimization of the overcommitment parameters as a way to maximize the resource utilization of cloud infrastructures, hence favorably contributing to both indicators.

Control plane impact. Regarding the control plane of OPENSTACK, one can observe in Figure 5.10 that the major power-consuming service is neutron-api, which is the central service for managing virtual machines in OPENSTACK. In our testbed, NEUTRON-API consumed 5$\times$ more than the average power consumption of a hosted VM (labelled as VMs (Reference) in Figure 5.10), which can be partially explained by the benchmark we run to stress the OPENSTACK platform.

Yet, when considering the overall energy consumption for the scenario we executed, one can observe in Figure 5.11 that almost 80% of the energy is consumed by the virtual machines, and 20% are imposed by the OPENSTACK services, resulting in an aggregated VPUE of 1.25.

We believe that such detailed reports offered by the SMARTWATTS software-defined power meter can be further exploited by the operators of OPENSTACK platforms to highlight

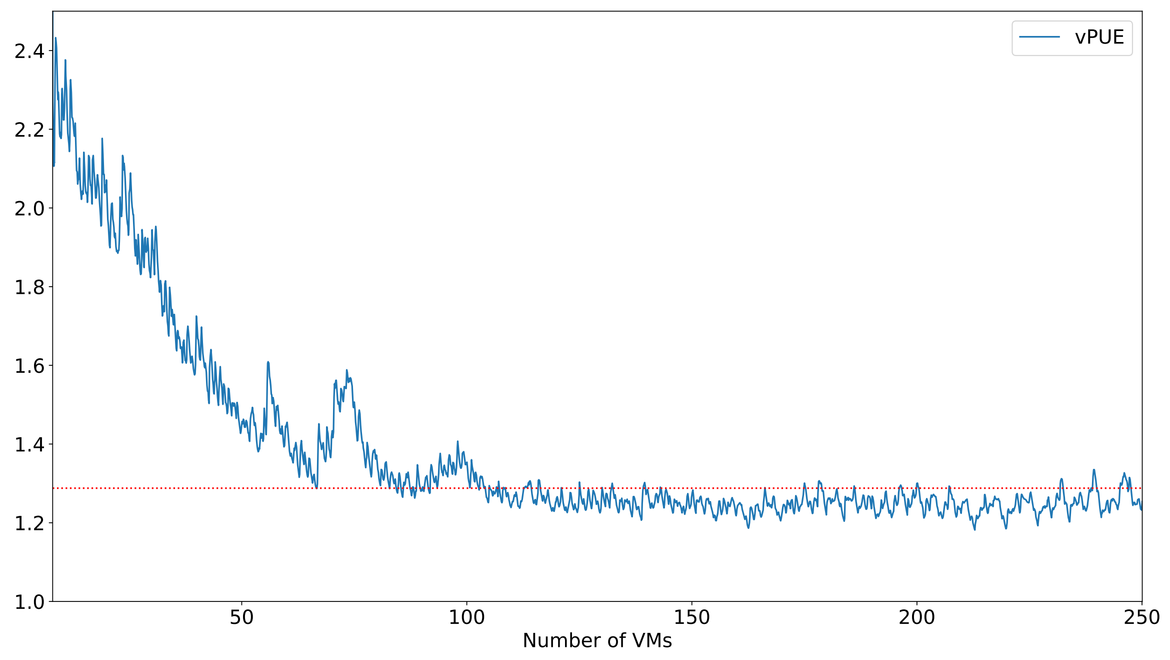

Fig. 5.9 Evolution of the vPUE for the amount of VMs

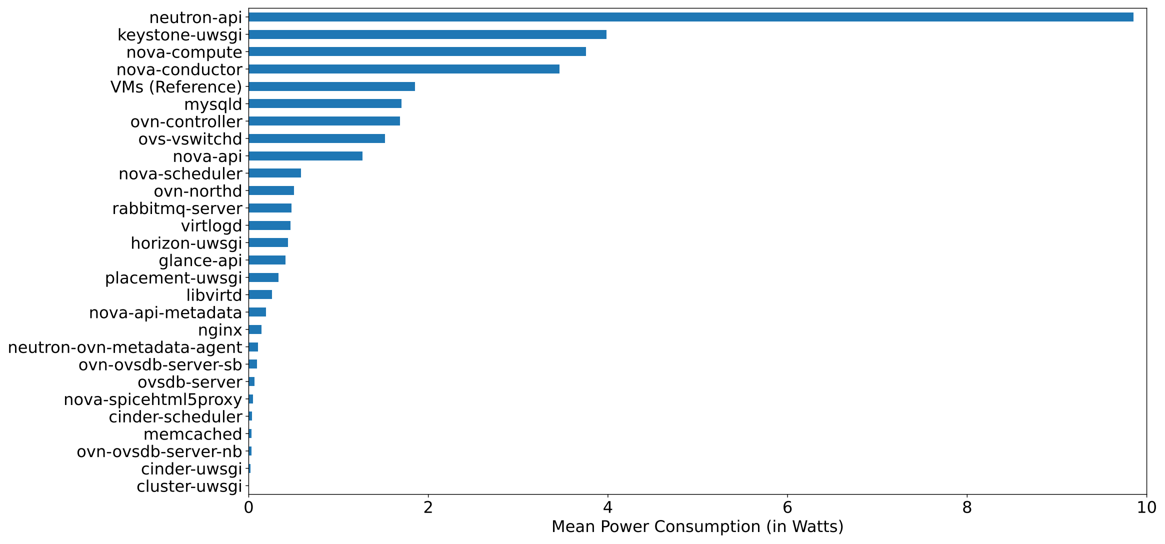

Fig. 5.10 Mean power consumption of the OPENSTACK services.

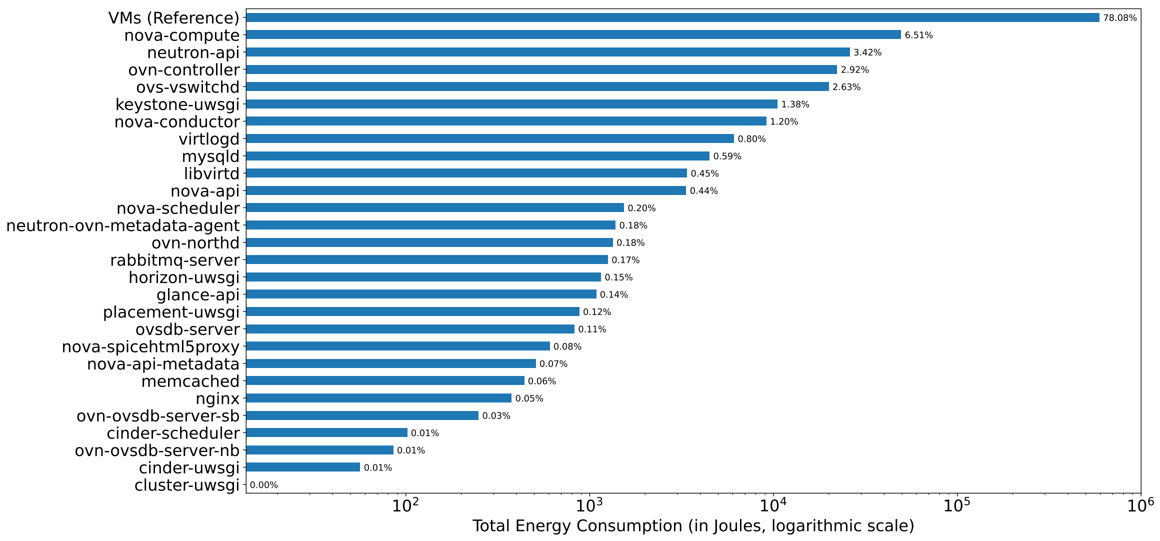

Fig. 5.11 Total energy consumption of the OpenStack services.

platform energy hotspots that require to be carefully considered, hence challenging the relevance and benefit of deployed services, beyond the standard configurations.

One should also note that, while the vPUE reports a lower value (1.25) than the sPUE (2.7, in Figure 5.6), it keeps representing a factor that needs to be combined with other indicators—*i.e,* cPUE $= 2.7 \times 1.25 = 3.375$ in this experiment—to reason upon global indicators and not partial ones.

To deepen our analysis, we then explore the relation between the energy efficiency of the infrastructure and its amount of workers. To do so, we ran a fixed workload that fully loads a node (in this case, 36 vCPUs) on clusters with a different amount of worker nodes. Figure 5.12 shows the energy usage proportion for the infrastructure services and the virtual machines deployed in OpenStack clusters with various amounts of workers nodes. As expected, the proportion of the energy used for infrastructure-related services decreases when we increase the amount of worker nodes. For a single worker cluster, infrastructure services consume 24.56% of the total energy consumed, and 74.44% is consumed by the virtual machines. This highlight the significant energy impact of the infrastructure, hence recommending to maximize the number of active nodes in a cluster to reduce its vPUE.

Estimating the vPUE of Kubernetes

Beyond the specific case of OpenStack, we also investigate the vPUE of a Kubernetes platform in this section.

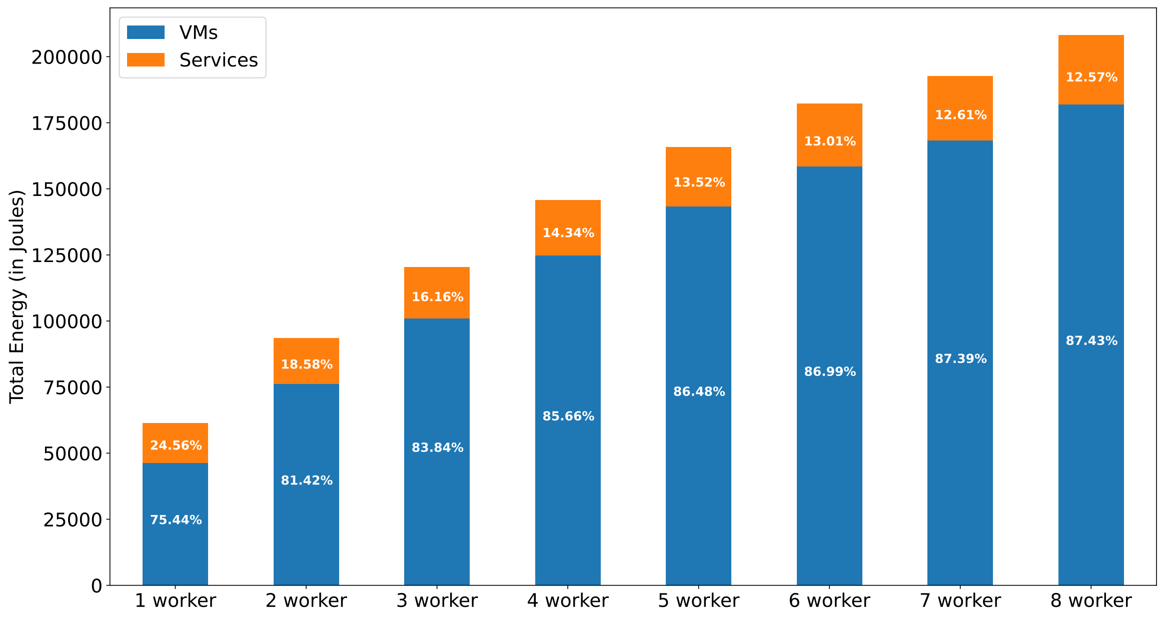

Fig. 5.12 Total energy consumption of the OPENSTACK cluster.

Table 5.4 Energy consumption of KUBERNETES and OPENSTACK services.

Node	Platform	Energy (kJ)	diff (kJ)
control services	KUBERNETES	15.6	
	OPENSTACK	**166.7**	**+151.1**
hosted jobs	KUBERNETES	484.3	
	OPENSTACK	**593.9**	**+109.6**

Platform setting. We keep using our benchmark designed to start twice the maximum CPU load of the cluster. To keep the configuration of the KUBERNETES cluster as close as possible to a production environment, we follow the best practices and do not allow the scheduling of containers to the control plane. This means that, in a cluster composed of 4 worker nodes, there are 144 available threads.

Figure 5.13 depicts the evolution over the time of the VPUE of this KUBERNETES cluster when deploying more than 400 containers across the 4 worker nodes.

During this experimentation, we observe an average VPUE of 1.3 with an optimal value of 1.1. Interestingly, one can observe that the control plane of KUBERNETES consumes less energy than the one of OPENSTACK to execute the same workload.

One can observe in Table 5.4 that the services composing the control plane of OPEN-STACK impose an overhead of 151.1 *kJ* (910%) compared to the control plane of KUBER-

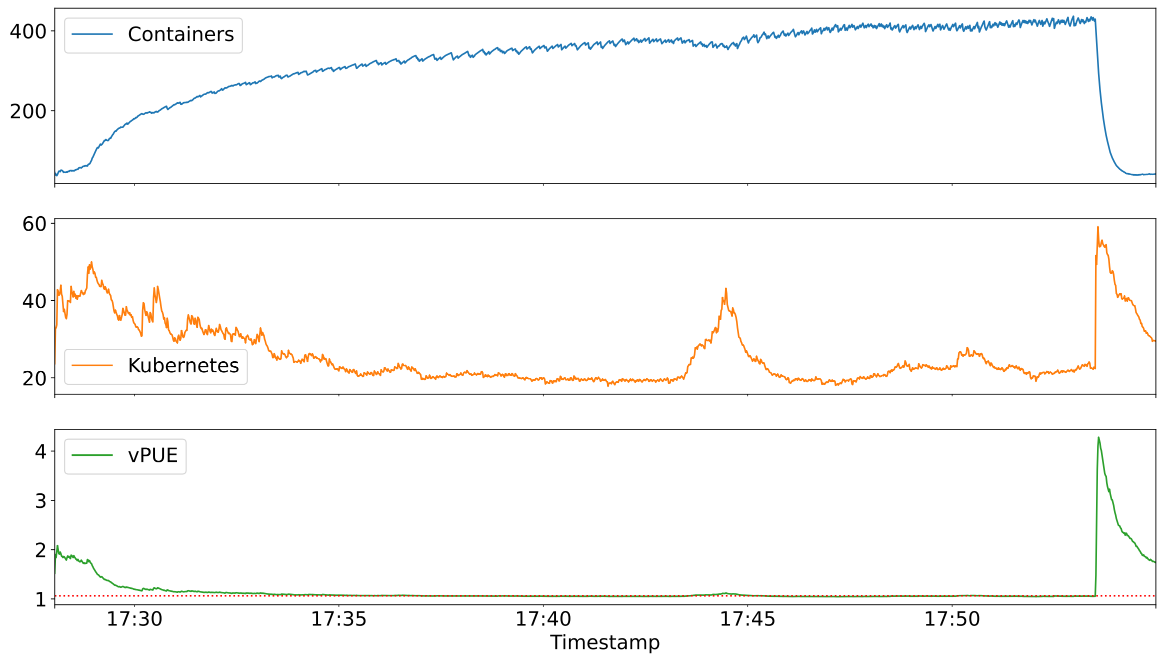

Fig. 5.13 Evolution of the vPUE of Kubernetes over the time.

netes, while the expectable overhead imposed by virtual machines over containers is limited to 109.6 kJ (23%).

The control plane of Kubernetes thus leaves 96.88% of the total energy consumed for the execution of hosted containers (cf. Figure 5.14), compared to 78.08% in the case of OpenStack.

As previously done for OpenStack, we ran a fixed workload that fully loads a node (in this case, 36 threads) on clusters with a different amount of worker nodes. Figure 5.15 shows the energy usage proportion for the infrastructure services and application containers deployed in Kubernetes clusters with various amounts of workers nodes. As expected, the proportion of the energy used for infrastructure-related services decreases when we increase the amount of worker nodes. For a single worker cluster, infrastructure services consume 14.07% of the total energy consumed, compared to 24.56% for an OpenStack cluster. One can therefore observe that this factor strongly impacts the vPUE of the cluster, hence recommending to maximize the number of active nodes in a cluster to reduce its vPUE.

5.4.4 cPUE & gPUE Experiments

As previously mentioned, Kubernetes clusters can be provisioned atop a set of VM hosted by a IaaS. In such a situation, the vPUE of the IaaS has to be combined with the one of Kubernetes to better reflect the resulting efficiency of the platform.

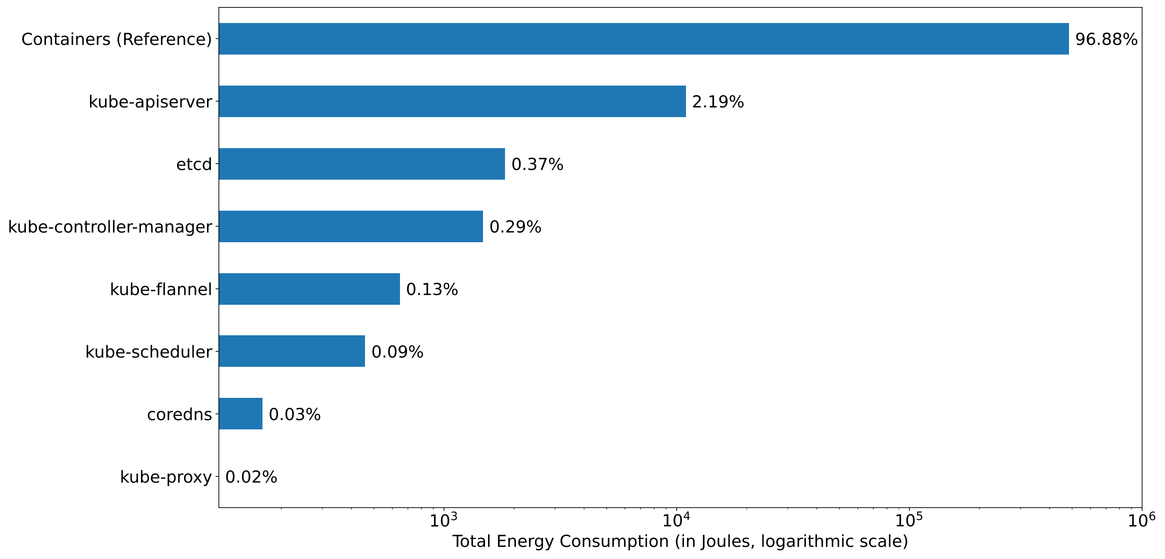

Fig. 5.14 Total energy consumption of the KUBERNETES services.

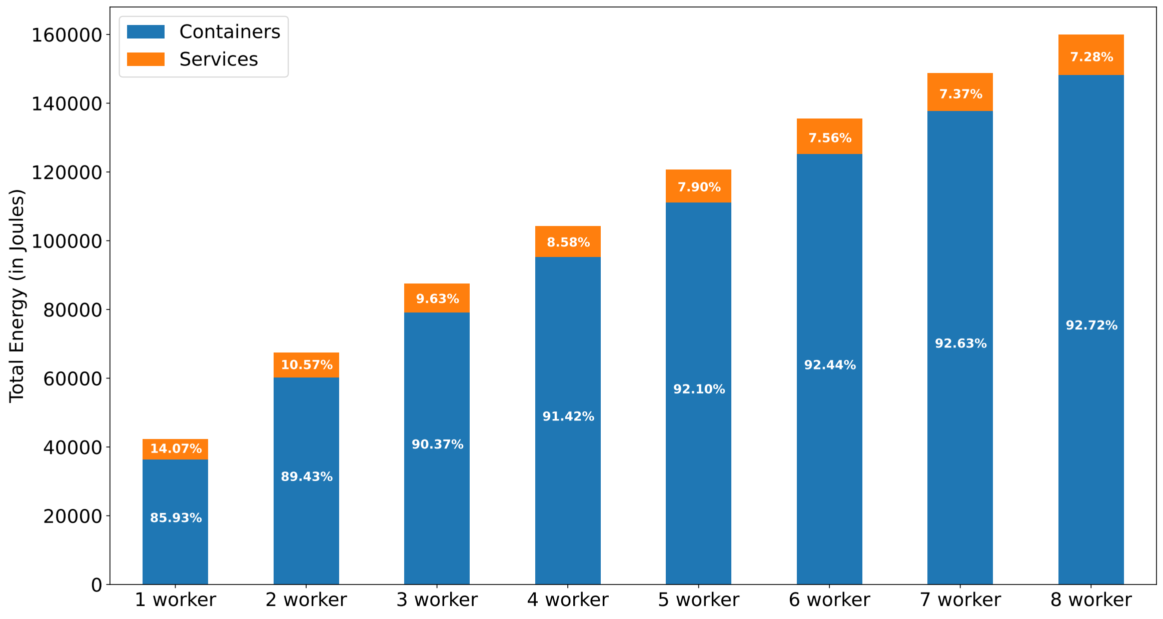

Fig. 5.15 Total energy consumption of the KUBERNETES cluster per amount of workers.

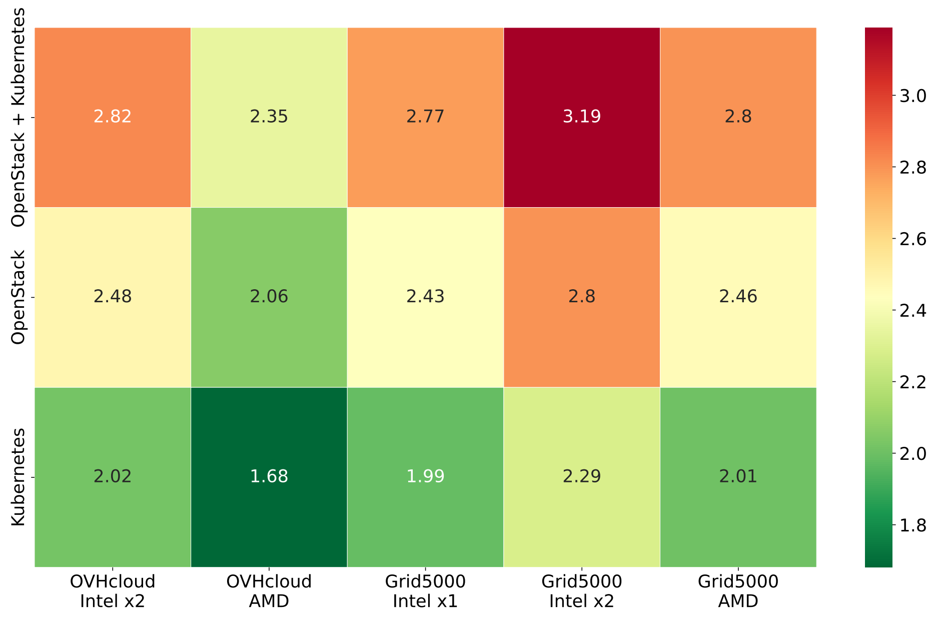

Fig. 5.16 Comparing the CPUE of hardware/software configurations.

To evaluate the ability of *x*PUE to assess the energy efficiency of a KUBERNETES infrastructure provisioned in an OPENSTACK cluster, we explore different configurations of IaaS/CaaS technologies and compute the associated CPUE. The estimated CPUE metrics are reported in Figure 5.16. By combining SPUE and VPUE metrics, one can observe that both the hardware server and the software stack can have a strong influence on the energy consumption of cloud infrastructures. In our setup, one can observe that the CPUE can range from 1.68 for a bare-metal KUBERNETES cluster hosted by AMD servers provisioned by OVHCLOUD to 3.19 for a virtualized KUBERNETES cluster provisioned atop a set of OPENSTACK virtual machines deployed in a private IaaS (Grid5000). This observation strengthens our claim that the optimization of energy efficiency of cloud infrastructure requires a holistic approach covering all hardware and software layers, beyond the sole optimization of the data center and its PUE.

To reveal the concrete energy footprint of a cloud service, one should therefore combine the CPUE of the cloud infrastructure with the PUE of the data center hosting the servers. Figure 5.17 estimates the global PUE (GPUE) of several server deployments hosted by OVHCLOUD, according to the selected data center. We use the PUE publicly published as reference for the data centers built by OVHCLOUD (*i.e.*, Gravelines in France, Beauharnois in Canada and Limburg in Germany). As for the data centers leased to OVHCLOUD (*i.e.*, Big

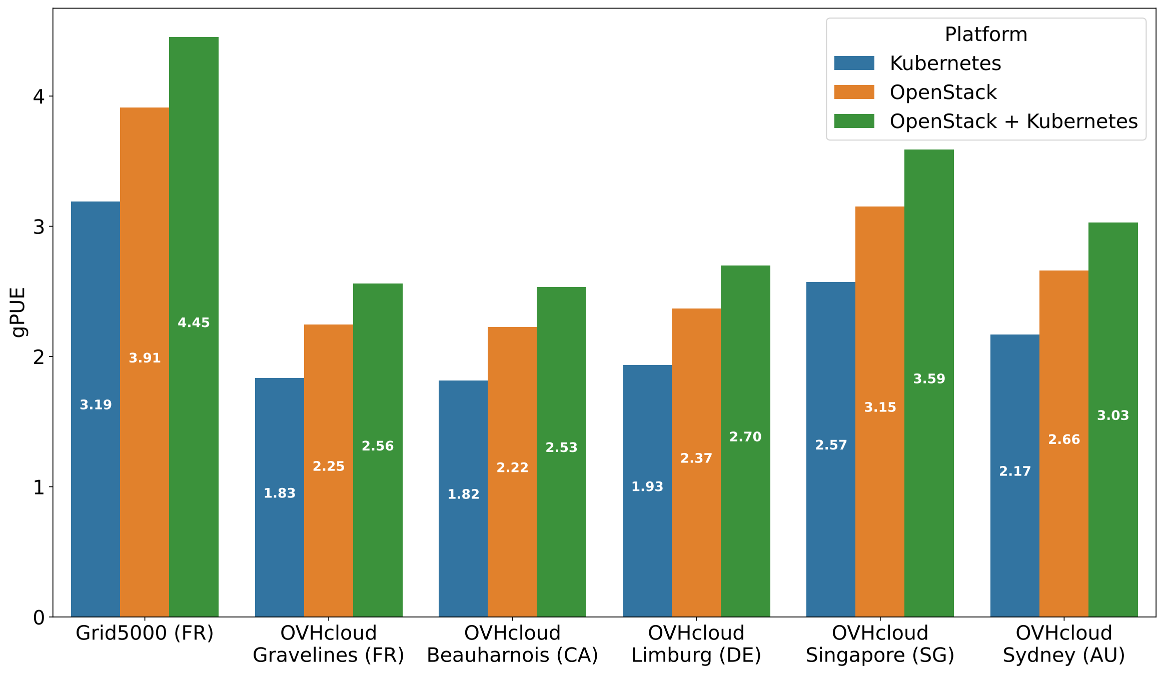

Fig. 5.17 Comparing the GPUE of DC/Platform configurations.

Data Exchange[16] (previously Telstra) for Singapore and NextDC[17] for Sydney in Australia), we use the data publicly published by the operators as reference.

Figure 5.18 estimate the global Carbon Usage Effectiveness (CUE) (GCUE) of several data center of OVHCLOUD across the world. As there is no data officially published by OVHCLOUD at the data center granularity for the CUE, we choose to use the data published by ELECTRICITY MAP[18] as reference for the Carbon Dioxide Emission Factor (CEF) of the countries in which is based each data center.

5.5 Summary

The energy efficiency of cloud infrastructure is a critical concern for modern deployments and a lot of work has been made in order to accurately evaluate this efficiency. While there are multiple indicators that aim to assess the efficiency of data centers, none of them take into account the efficiency of the running software. They mostly treat the running software as a black-box and give a yearly feedback about the global efficiency of the infrastructure, which does not allow to evaluate individually specific parts of the infrastructure.

[16]https://www.bdxworld.com/
[17]https://www.nextdc.com/
[18]https://www.electricitymaps.com/

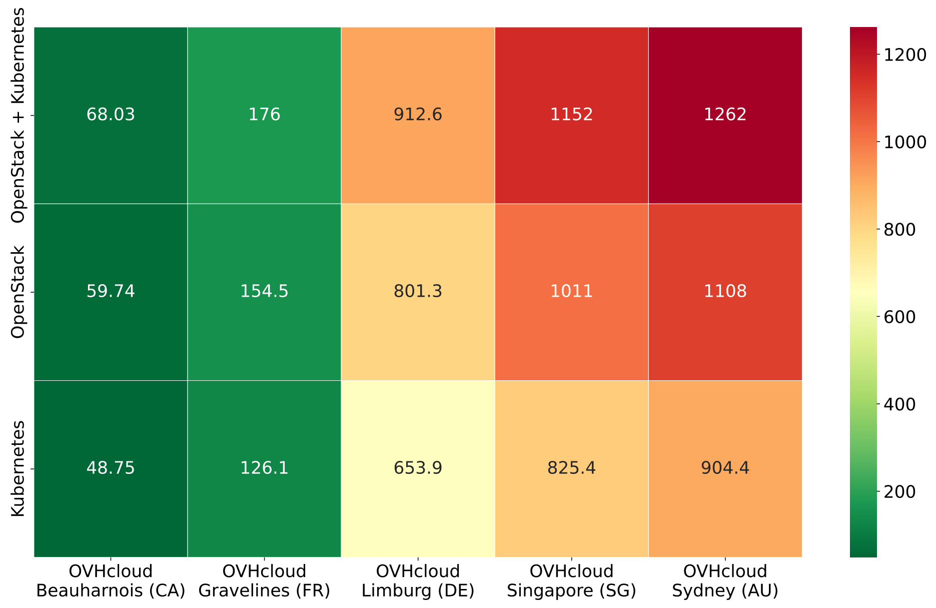

Fig. 5.18 Comparing the GCUE (gCO_2/kWh) of OVHcloud data center of various countries.

In this chapter, we presented *x*PUE, an extension of the *Power Usage Effectiveness* (PUE) metric that allows operators and users to assess the energy efficiency of their infrastructure. We mainly focus on the energy efficiency of the computing hardware and theirs underlying software services composing the infrastructure, and we show that *x*PUE success to provide an in-depth view of the energy efficiency of the various hardware and software components of a cloud infrastructure.

While we demonstrate our approach on *Kubernetes* and *OpenStack* test clusters, we strongly believe that it can be deployed and used across a large variety of infrastructure. We took care to allow a high flexibility to the users in order to easily adapt our solution to their specific infrastructure control plane and tools. The experimental results obtained showed the ability of our approach to flexibly adapt to the underlying infrastructure, and to provide a deep insight of the energy efficiency of the multiple layers composing the infrastructure in real-time.

Thanks to *x*PUE, cloud infrastructure operators can assess in real-time the efficiency of their infrastructure, from the hardware level and down to the software level. This allows for more experiments on energy management policies, and a faster feedback about its short, middle and long term behavior and efficiency for the cloud operator. Instead of relying on very complicated metrics that have a very long delay (mostly yearly) before getting feedback, we believe that *x*PUE provides a reliable indicator of the energy efficiency of the different parts of a cloud infrastructure while allowing more flexibility for the cloud operator.

Chapter 6

Conclusion & Perspectives

In this chapter, we summarize the contributions of our thesis and we discuss our short and long-term perspectives.

6.1 Summary of Contributions

In this thesis, we presented three contributions to the research in the field of software energy consumption.

In Chapter 3, we presented SmartWatts which is a self-adaptive power model for software containers based on the Hardware Performance Counters of the CPUs for the activity, and the *Running Average Power Limit (RAPL)* interface for the CPU energy measurements. This contribution provides tools to accurately estimate the power consumption of the software containers deployed on a machine, thus allowing operators and developers to better analyze the energy efficiency of their software systems.

In Chapter 4, we presented SelfWatts which is a self-calibrating power model that allows to automatically configure software power meters in heterogeneous environments. This contribution provides tools to easily deploy software power meters in heterogeneous environments, without needing specific configuration and specialized knowledge about the underlying hardware architecture of the machine.

In Chapter 5, we presented xPUE which is an extension of the *Power Usage Effectiveness* (PUE) indicator to evaluate the software energy efficiency of the various levels of the computing infrastructures. This contribution extends the scope the the PUE indicator to the software level, thus allowing to evaluate the energy efficiency of the software systems deployed across an infrastructure.

All contributions have been implemented and validated on a real-world testbed, and the results have been published in international conferences and journals. These contributions provide usable approaches and tools to accurately measure the energy consumption of software deployed at different levels of the infrastructure. To foster the adoption of our contributions, all tools are open-source and freely available on GitHub (cf. Section 1.3 for the links to the repositories).

6.2 Short-Term Perspectives

In this section we present the short-term perspectives of this thesis. As mentioned in the previous chapters, many perspectives can be considered throughout our presentation of the contributions and their implications.

6.2.1 Extends Software Power Meters to more Processing Units

In Chapter 3 and Chapter 4, we presented software-defined power meters that are able to measure the CPU and DRAM energy consumption of software containers running on heterogeneous hardware configurations. Market trends are pushing the industry to develop more heterogeneous architectures, with more processing units (e.g. GPUs, FPGAs, etc.) and more complex power management policies. While we designed our approaches to be easily extensible to other processing units, we did not have the opportunity to validate them on such architectures.

Central Processing Units.

CPUs architecture are greatly evolving and the industry is pushing for more flexible, performant and energy efficient architectures.

Recently, Intel pushed for heterogeneous multi-core architectures for the first time in its Lakefield micro-architecture, where big—more powerful and power-hungry—core(s) along with small—slower and energy-efficient—cores are integrated on the same die. This concept, borrowed from ARM big.LITTLE architecture, is called *Hybrid Cores* and is a promising approach to improve the energy efficiency of the CPUs [34].

ARM is also becoming more and more present in the data center and *High Performance Computing* (HPC) spaces in addition to be the major player in the mobile space. To this effect, major cloud providers such as Amazon Web Services (AWS), Microsoft Azure and *Google Cloud Platform* (GCP) introduced their own ARM-based servers, where better efficiency often reduces the price of such services.

RISC-V is an open-source *Instruction Set Architecture* (ISA) that is gaining traction in the industry. It is mostly dedicated to embedded systems and the *Internet of Things* (IoT) devices, where energy efficiency is a major concern because of the limited battery life of such environment.

Unfortunately, the current software-defined power meters are not able to measure the energy consumption of software containers on these architectures. While many software power meters rely on the *Running Average Power Limit* (RAPL) feature on Intel and AMD CPUs, there is no equivalent feature on ARM and RISC-V architectures. Keller *et al.* [33] proposed a RISC-V *System-on-Chip* (SoC) with an integrated power-management unit that can measure system state and actuate changes to core voltage and frequency, which allows to measure the energy consumption of the software. For ARM architectures, this feature is currently available on specific development boards where an energy probe is required, and it is not proposed on server SoC of the market.

Specialized Processing Units.

As *Machine Learning* (ML) and *Artificial Intelligence* (AI) are becoming more and more important, the industry is pushing for more specialized processing units to accelerate the execution of ML and AI algorithms. Hardware dedicated to improving the performance of such algorithms are called *Accelerators* and are often based on *Graphics Processing Units* (GPUs), *Tensor Processing Units* (TPU) or *Field-Programmable Gate Arrays* (FPGAs).

Cryptocurrencies mining is another good example of this trend, where the industry is pushing for more specialized processing units to accelerate the execution of the algorithms used to mine cryptocurrencies, and thus increase the profit of the miners. In this case, the processing units are often based on *Application-Specific Integrated Circuits* (ASICs) or *Graphics Processing Units* (GPUs).

Concerning the energy consumption of *Graphics Processing Units* (GPUs), there is a lot of work in the literature to measure the energy consumption of these devices, and to optimize the energy efficiency of the algorithms executed on them. Most GPUs vendors provide tools to measure the energy consumption of their devices, and there are also open-source tools such as Nvidia Management Library (NVML) and ROCm that are able to measure the energy consumption of GPUs. However, the equivalent for *Application-Specific Integrated Circuits* (ASICs) and *Field-Programmable Gate Arrays* (FPGAs) is far more sporadic, which greatly hinders the energy profiling of such devices.

6.2.2 Intelligent Application-Level Power Budgeting

Nowadays, resources intensive applications have access to many solutions to accelerate their execution, such as *Accelerators* (e.g. GPUs, TPUs, FPGAs, etc.), *Cloud Computing* (e.g.

AWS, Azure, GCP, etc.), *High Performance Computing* (HPC) clusters, etc. In Chapter 5, we presented *x*PUE which is an extension of the *Power Usage Effectiveness* (PUE) indicator to evaluate the software energy efficiency of the various hardware and software levels of computing infrastructures. While we showed the importance of using efficient hardware and software architectures and to choose appropriately the data center location, we did not address the energy efficiency of the applications themselves, especially when accelerators were used.

In this perspective, multiple approaches can be used to improve the energy efficiency of applications. First, the application can be optimized to reduce its own energy consumption under a set of user-defined constraints. For example, in Enes *et al.* [18], we proposed a platform based on our work presented in Chapter 3 to automatically distribute and enforce a power budget among users and applications. This platform allows users to set a power budget on their containerized applications and have it enforced in real-time and in a dynamic way. However, we did not explore the possibility to dynamically manage the power budget of applications using accelerators.

Secondly, one can explore the energy efficiency of its application/infrastructure on various hardware configurations, and plan accordingly to deploy the application on the most energy efficient configuration. For example, the authors of HEATS [49] took this approach to exploit the hardware diversity of the underlying infrastructures in order to make energy-aware scheduling decisions and improves the energy efficiency of the tasks deployed.

6.3 Long-Term Perspectives

In this section, we present the long-term perspectives of this thesis.

6.3.1 Distributed Application Energy Efficiency

In Chapter 5, we presented an analysis of the energy efficiency of the various levels of the computing infrastructures. However, we mainly assessed the energy efficiency of Cloud Computing architectures, where workloads are deployed on shared IT resources hosted in a data center.

Edge Computing is a computing paradigm where the data is processed at the edge of the network, close to the source of the data, instead of being processed in the cloud. This paradigm is used to reduce the latency of the applications, and to reduce the energy consumption of the applications by reducing the amount of data that needs to be transferred over the network. However, as stated by Jiang *et al.* [30], the energy consumption of the applications is still a major concern in this paradigm, and there is a lot of work to do to improve the energy efficiency of the applications deployed in such environments.

As the popularity of Edge Computing is increasing, we believe there is a need to develop new approaches to measure the energy consumption and efficiency of the infrastructures and applications deployed in such environments.

6.3.2 Application-Driven Hardware Power Management

Currently, the power management of the hardware is mostly driven by the operating system or the hardware itself. However, the operating system is not aware of the energy consumption of the applications, and thus cannot adapt the power management policies to the energy consumption of the applications. Such optimization mechanics are mostly based on heuristic and will always lack application specific knowledge.

Some efforts have been made to allow the operating system to adapt its power management policies to the energy consumption of the applications. In the Linux kernel, the *Energy Model* (EM) framework is an interface between drivers knowing the power consumed by various devices, and the kernel subsystems willing to use that information to make energy-aware decisions.[1] While this is a big step enabling more efficient scheduling of tasks on heterogeneous CPU topologies, it is still not enough to adapt the power management policies to a specific application.

Another approach to improve the energy efficiency is to have the hardware power management policy tailored to the workload of the application. Wright *et al.* [57] presents a dual-core RISC-V *System-on-Chip* (SoC) with integrated fine-grain power management, which allows to adapt the power management policies to the workload of the machine. This technique provides a good example of the potential of such approach, but it is still limited to a specific hardware architecture.

We believe there is still space for application-agnostic power management policies that can be automatically adapted to the workload of the application, and thus collaborate actively with the hardware energy optimization mechanism in order to improve the efficiency of the whole system.

6.3.3 Energy Efficient Trusted Execution Environments (e-TEE)

Trusted Execution Environments (TEE) are a class of hardware that provides a secure environment to execute sensitive applications. Cloud infrastructures supporting secure environments are emerging, such as Microsoft Azure confidential computing, allowing safe resources to

[1] https://docs.kernel.org/power/energy-model.html

be shared across multiple stakeholders. A Linux Foundation Consortium has been founded by Microsoft, Intel, AMD, and others actors to develop a standard for TEEs, called the *Confidential Computing Consortium* (CCC) [45].

In this category, Intel SGX is an approach to safely isolate software processes sharing the same resources (CPU, RAM) of a given host. Arnautov *et al.* [1] provides an approach to protect Docker containers from external tampering using Intel's SGX enclaves. However, such security guarantees come at a price and Gjerdrum *et al.* [24] evaluates the performance impact imposed by Intel SGX enclaves. An increase of the energy consumption for the application is also observed by Göttel *et al.* [26], especially when the application do not respect the memory constraints imposed by the enclave.

Another case of such application is the *Homomorphic Encryption* (HE) which is a technique to perform computations on encrypted data. In some cases, applications rely on *Near-Memory Processing* (NMP) and *Computing-in-Memory* (CiM) paradigms, where computation is done within the memory boundaries, to reduce latency and energy associated with data transfers in data-intensive applications [47].

While the TEEs are a promising technology to protect sensitive data, we believe there is still a lot of work to do to improve the energy efficiency of the applications running in such environments.

References

[1] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 689–703, USA. USENIX Association.

[2] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., and Schreiber, R. S. (1991). The nas parallel benchmarks. *IJHPCA*.

[3] Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., and Sarzyniec, L. (2013). Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing.

[4] Barroso, L. A., Clidaras, J., and Hölzle, U. (2013). The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154.

[5] Bedard, D., Lim, M. Y., Fowler, R., and Porterfield, A. (2010). Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*.

[6] Bellosa, F. (2000). The benefits of event: Driven energy accounting in power-sensitive systems. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*.

[7] Bertran, R., González, M., Martorell, X., Navarro, N., and Ayguadé, E. (2010). Decomposable and responsive power models for multicore processors using performance counters. In *ICS*, pages 147–158. ACM.

[8] Bircher, W. and John, L. (2007). Complete system power estimation: A trickle-down approach based on performance events. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS '07.

[9] Bircher, W. L., Valluri, M., Law, J., and John, L. K. (2005). Runtime identification of microprocessor energy saving opportunities. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

[10] Brady, G. A., Kapur, N., Summers, J. L., and Thompson, H. M. (2013). A case study and critical assessment in calculating power usage effectiveness for a data centre. *Energy Conversion and Management*, 76:155–161.

[11] Colmant, M., Felber, P., Rouvoy, R., and Seinturier, L. (2017). Wattskit: Software-defined power monitoring of distributed systems. In *CCGrid*, pages 514–523. IEEE / ACM.

[12] Colmant, M., Kurpicz, M., Felber, P., Huertas, L., Rouvoy, R., and Sobe, A. (2015). Process-level power estimation in vm-based systems. In *EuroSys*, pages 1–14. ACM, ACM.

[13] Colmant, M., Rouvoy, R., Kurpicz, M., Sobe, A., Felber, P., and Seinturier, L. (2018). The next 700 CPU power models. *JSS*, 144:382–396.

[14] Contreras, G. and Martonosi, M. (2005). Power Prediction for Intel XScale® Processors Using Performance Monitoring Unit Events. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

[15] Dayarathna, M., Wen, Y., and Fan, R. (2016). Data center energy consumption modeling: A survey. *IEEE Commun. Surv. Tutorials*, 18(1):732–794.

[16] Desrochers, S., Paradis, C., and Weaver, V. M. (2016). A validation of DRAM RAPL power measurements. In Jacob, B., editor, *MEMSYS*, pages 455–470. Association for Computing Machinery, ACM.

[17] Dolz, M. F., Kunkel, J., Chasapis, K., and Catalán, S. (2015). An analytical methodology to derive power models based on hardware and software metrics. *Computer Science - Research and Development*.

[18] Enes, J., Fieni, G., Expósito, R. R., Rouvoy, R., and Touriño, J. (2020). Power budgeting of big data applications in container-based clusters. In *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*, pages 281–287. IEEE.

[19] ETSI (2014). Access, Terminals, Transmission and Multiplexing (ATTM); Energy management; Global KPIs; Operational infrastructures; Part 2: Specific requirements; Sub-part 1: Data centres. Standard, ETSI, Sophia Antipolis CEDEX, FR.

[20] ETSI (2017). Access, Terminals, Transmission and Multiplexing (ATTM); Energy management; Global KPIs; Operational infrastructures; Part 3: Global KPIs for ICT sites. Standard, ETSI, Sophia Antipolis CEDEX, FR.

[21] Fieni, G., Rouvoy, R., and Seinturier, L. (2020). Smartwatts: Self-calibrating software-defined power meter for containers. In *CCGrid*, pages 479–488. IEEE.

[22] for Standardization, I. O. (2016). Information technology – Data centres – Key performance indicators – Part 2: Power usage effectiveness (PUE). Standard, International Organization for Standardization, Geneva, CH.

[23] Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., and Cameron, K. (2010). Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*.

[24] Gjerdrum, A. T., Pettersen, R., Johansen, H. D., and Johansen, D. (2017). Performance of Trusted Computing in Cloud Infrastructures with Intel SGX. In *7th International Conference on Cloud Computing and Services Science (CLOSER)*.

[25] Google (2021). Data centers efficiency. https://www.google.com/about/datacenters/efficiency/.

[26] Göttel, C., Pires, R., Rocha, I., Vaucher, S., Felber, P., Pasin, M., and Schiavoni, V. (2018). Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 133–142.

[27] Havet, A., Schiavoni, V., Felber, P., Colmant, M., Rouvoy, R., and Fetzer, C. (2017). GENPACK: A generational scheduler for cloud data centers. In *2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, pages 95–104. IEEE Computer Society.

[28] Insitute, U. (2020). Data center pues flat since 2013. https://journal.uptimeinstitute.com/data-center-pues-flat-since-2013/.

[29] Isci, C. and Martonosi, M. (2003). Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*.

[30] Jiang, C., Fan, T., Gao, H., Shi, W., Liu, L., Cérin, C., and Wan, J. (2020). Energy aware edge computing: A survey. *Computer Communications*, 151:556–580.

[31] Kamiya, G. (2022). Data centres and data transmission networks. Technical report, IEA: International Energy Agency.

[32] Kansal, A., Zhao, F., Liu, J., Kothari, N., and Bhattacharya, A. A. (2010). Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*.

[33] Keller, B., Cochet, M., Zimmer, B., Kwak, J., Puggelli, A., Lee, Y., Blagojević, M., Bailey, S., Chiu, P.-F., Dabbelt, P., Schmidt, C., Alon, E., Asanović, K., and Nikolić, B. (2017). A risc-v processor soc with integrated power management at submicrosecond timescales in 28 nm fd-soi. *IEEE Journal of Solid-State Circuits*, 52(7):1863–1875.

[34] Khushu, S. and Gomes, W. (2019). Lakefield: Hybrid cores in 3d package. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–20.

[35] Kurpicz, M., Orgerie, A., and Sobe, A. (2016). How much does a VM cost? energy-proportional accounting in vm-based environments. In *Euromicro*, pages 651–658. IEEE.

[36] Laros, J. H., Pokorny, P., and DeBonis, D. (2013). Powerinsight - a commodity power measurement capability. In *Green Computing Conference, 2013 International*.

[37] LeBeane, M., Ryoo, J. H., Panda, R., and John, L. K. (2015). Wattwatcher: Fine-grained power estimation for emerging workloads. In *SBAC-PAD*.

[38] Li, T. and John, L. K. (2003). Run-time modeling and estimation of operating system power consumption. *SIGMETRICS Perform. Eval. Rev.*

[39] Lim, M. Y., Porterfield, A., and Fowler, R. (2010). Softpower: Fine-grain power estimations using performance counters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*.

[40] Lupton, R. and Allwood, J. (2017). Hybrid sankey diagrams: Visual analysis of multidimensional data for understanding resource use. *Resources, Conservation and Recycling*, 124:141–151.

[41] McCullough, J. C., Agarwal, Y., Chandrashekar, J., Kuppuswamy, S., Snoeren, A. C., and Gupta, R. K. (2011). Evaluating the effectiveness of model-based power characterization. In *Proceedings of the USENIX Annual Technical Conference*.

[42] Noureddine, A., Rouvoy, R., and Seinturier, L. (2015). Monitoring energy hotspots in software - energy profiling of software code. *Autom. Softw. Eng.*

[43] Orgerie, A.-C., Dias de Assuncão, M., and Lefèvre, L. (2014). A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*

[44] Prekas, G., Primorac, M., Belay, A., Kozyrakis, C., and Bugnion, E. (2015). Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*.

[45] Rashid, F. Y. (2020). The rise of confidential computing: Big tech companies are adopting a new security model to protect data while it's in use - [news]. *IEEE Spectrum*, 57(6):8–9.

[46] Rashti, M., Sabin, G., Vansickle, D., and Norris, B. (2015). Wattprof: A flexible platform for fine-grained hpc power profiling. In *2015 IEEE International Conference on Cluster Computing*.

[47] Reis, D., Takeshita, J., Jung, T., Niemier, M., and Hu, X. S. (2020). Computing-in-memory for performance and energy-efficient homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(11):2300–2313.

[48] Rivoire, S., Ranganathan, P., and Kozyrakis, C. (2008). A comparison of high-level full-system power models. In *Proceedings of the Conference on Power Aware Computing and Systems*.

[49] Rocha, I., Göttel, C., Felber, P., Pasin, M., Rouvoy, R., and Schiavoni, V. (2019). Heats: Heterogeneity-and energy-aware task-based scheduling. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 400–405.

[50] Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., and Rajwan, D. (2012). Power-management architecture of the intel microarchitecture code-named sandy bridge. *Micro*.

[51] Schaeppi, B., Bogner, T., Schloesser, A., Stobbe, L., and de Asuncao, M. D. (2012). Metrics for energy efficiency assessment in data centers and server rooms. In *2012 Electronics Goes Green 2012+*, pages 1–6. IEEE.

[52] Shen, K., Shriraman, A., Dwarkadas, S., Zhang, X., and Chen, Z. (2013). Power containers: an OS facility for fine-grained power and energy management on multicore servers. In *ASPLOS*, ASPLOS '13, pages 65–76, New York, NY, USA. ACM.

[53] Snowdon, D. C., Sueur, E. L., Petters, S. M., and Heiser, G. (2009). Koala: a platform for os-level power management. In *EuroSys*, pages 289–302. ACM.

[54] Stoess, J., Lang, C., and Bellosa, F. (2007). Energy management for hypervisor-based virtual machines. In *Proc. of USENIX Annual Technical Conference*.

[55] van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C., and Heiser, G. (2018). Benchmarking crimes: An emerging threat in systems security. *CoRR*, abs/1801.02381.

[56] Versick, D., Waßmann, I., and Tavangarian, D. (2013). Power consumption estimation of cpu and peripheral components in virtual machines. *SIGAPP Appl. Comput. Rev.*, 13(3):17–25.

[57] Wright, J. C., Schmidt, C., Keller, B., Dabbelt, D. P., Kwak, J., Iyer, V., Mehta, N., Chiu, P.-F., Bailey, S., Asanović, K., and Nikolić, B. (2020). A dual-core risc-v vector processor with on-chip fine-grain power management in 28-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2721–2725.

[58] Yang, H., Zhao, Q., Luan, Z., and Qian, D. (2014). iMeter: An integrated {VM} power model based on performance profiling. *Future Generation Computer Systems*.

[59] Zamani, R. and Afsahi, A. (2012). A study of hardware performance monitoring counter selection in power modeling of computing systems. In *IGCC*, pages 1–10. IEEE Computer Society.

[60] Zhai, Y., Zhang, X., Eranian, S., Tang, L., and Mars, J. (2014). Happy: Hyperthread-aware power profiling dynamically. In *Proceedings of the USENIX Annual Technical Conference*.