UCLouvain

École polytechnique de Louvain

Energy-Aware Scheduling for ISA-Heterogeneous Clusters

From Process-Level Power Estimations to Cluster-Wide Energy Efficiency

Authors: **Simon Arys, Romain Carlier**
Supervisor: **Etienne Rivière**
Readers: **Tom Barbette, Nikita Tyunyayev**
Academic year 2023–2024
Master [120] in Computer Science

Acknowledgments

We would like to thank our supervisor Professor Etienne Rivière for his dedication, his constant availability and all the valuable advice he shared with us. We are grateful to Tom Barbette and Nikita Tyunyayev for accepting to be readers of our thesis. We also would like to address a special thank you to the INGI sysadmins for making sure we could have access to our dedicated machines throughout this thesis.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Another thank you to the Fond de la Recherche Scientifique de Belgique (F.R.S.-FNRS) for having funded and made available the specialised hardware without which our research would not have been possible.

Finally, we would like to express our deepest gratitude to our families and friends for their unwavering care and support throughout the past five years.

Abstract

Hardware heterogeneity, and more specifically CPU architecture heterogeneity, is becoming increasingly prevalent in cloud data centers. Recently, the Ampere Computing supplier released a new processor family for the Cloud Computing market, highlighting energy efficiency and sustainability, and offering a compelling alternative to Intel's performance-driven CPUs. The objective of this thesis is to leverage this source of heterogeneity for smarter and more energy-efficient resource management. More specifically, our work targets energy-aware scheduling for the Function as a Service (FaaS) deployment model, where cloud providers commonly rely on a Round-Robin or consolidation approach to decide on a function-to-machine placement. Our methodology utilises apriori process-level power consumption measurements to characterise the energy affinity of serverless functions towards our two heterogeneous CPUs. We designed two constraint programming static schedulers: an Energy-aware scheduler aiming at being the most energy-efficient based on these apriori measurements and a Cores-aware scheduler focusing on reducing the number of CPU cores required to run a given set of functions. However, collecting the energy consumption at the function level is not a trivial task. For this purpose, we rely on PowerAPI, a software toolchain to estimate the power consumption of individual processes, which we extended to support Ampere CPUs. To evaluate our schedulers, we designed benchmarks and workloads based on our implementation of 22 orchestrator-agnostic serverless functions covering multiple programming languages and behaviours. Our evaluation results show that our proposed Energy-aware scheduler reduces the total energy consumed by 2% to 5% on average, and up to 19% in specific configurations compared to common scheduling strategies and our Cores-aware implementation.

Contents

Introduction

Cloud computing provides scalable and flexible on-demand availability of computing resources over the internet for businesses and individual users. To achieve performance and cost-efficiency, cloud providers rely on smart resource management systems that come in multiple flavours. One of them, and the main subject of this thesis, is scheduling. The scheduling problem can be defined as the decision of a tasks-to-resources mapping aiming at improving one or multiple performance metrics such as latency, throughput or energy consumption. In this thesis, we focus on energy-efficient task placement strategies within machine clusters having heterogeneous Instruction Set Architectures (ISA). Indeed, data center hardware is becoming increasingly heterogeneous year after year. In 2019, the market share of ARM-based chips in data centers was below one percent and climbed up to over 7% in just 5 years [5]. Hardware, and more specifically CPU heterogeneity is gaining ground and presents an opportunity to improve data centers' energy efficiency. Our hypotheses are the following: a given task, running on two heterogeneous CPUs, neither have the same performance nor energy consumption. Conversely, two different tasks running on a single processor also do not expose the same performance and energy profiles. By apriori measuring and characterising such profiles, we believe that we can reduce the overall energy consumption of ISA-heterogeneous clusters by assigning each task to its most efficient machine.

(a) Intel Xeon Gold 5318Y CPU (b) Ampere Altra Max CPU

Figure 1: **CPUs illustration.** A visual representation of the Ampere Altra Max and Intel Xeon Gold 5318Y CPUs used in our experiments. Images © from vendors' websites.

In 2020, the Ampere CPU supplier released its novel Altra processor family targeting the Cloud Computing market. Its advertising focuses on "high performance, security isolation, extreme scalability and more importantly sustainability with

unmatched power efficiency" [3]. On the other hand, Intel started shipping its third-generation Xeon CPU in 2021 under the Ice Lake code name for the server segment. Unlike Ampere, Intel's selling arguments are not focused on energy efficiency but rather on performance. Recently, the Cloud and Large-Scale Computing Group at UCLouvain acquired multiple high-end servers, one equipped with an Ampere Altra Max CPU and eight dual-socket servers with two Intel Xeon CPUs each. These machines will be part of the Grid'5000 federated testbed [39] which offers access to diverse CPU models. Our work focuses on two ISA-heterogeneous rack-scale servers, respectively supplied with an Ampere Altra Max CPU and two Intel Xeon Gold 5318Y CPUs (dual-socket). Both machines embed 256 GB of RAM. A description of the characteristics of the two CPUs that we have at our disposal is provided in Table 1.

	2 x Intel Xeon Gold 5318Y	**Ampere Altra Max**
CPU	2 x 24 cores, 2 x 48 threads	128 cores
Architecture	x86 64-bit	ARM v8.2+ 64-bit
SMT	yes	no
CPU TDP	2 x 175 W	250 W
Max Clock	3.4 GHz	3 GHz
Base Clock	2.1 GHz	1 GHz
Lithography	10 nm	7 nm
L1	32KB L1i, 48KB L1d per core	64KB L1i, 64KB L1d per core
L2	1.25MB per core	1MB per core
L3	36 MB	/
SLC	/	16MB

Table 1: **CPUs hardware characteristics comparison.** The table compares the specifications of two Intel Xeon Gold 5318Y CPUs (dual-socket) and an Ampere Altra Max CPU, as described in the vendors' documentation. SMT stands for Simultaneous Multi-Threading and is a feature allowing the execution of multiple independent threads in a single core. Intel refers to this technology under the commercial name of Hyper-Threading. The SLC (System Level Cache) is a shared last-level cache on Ampere CPUs. It is similar to Intel's L3 cache but not equivalent [2].

In the realm of cloud computing, various paradigms exist to facilitate application development and deployment. Among these, the Function as a Service (FaaS) model, also known as the serverless model, is of growing interest. FaaS offers a fundamentally different approach compared to traditional cloud deployments, in which developers focus on writing small and self-contained pieces of code called functions. In this context, the infrastructure and deployment of function code is handled by the cloud operator. Functions are executed upon user request, which enables a pay-per-use billing scheme for customers. As the functions are self-contained and independent, they can scale automatically based on the load level. This frees developers from infrastructure management, allowing them to concentrate on code writing. Some use cases for that deployment method are image and video manipulation, machine learn-

ing inference or even Continuous Integration and Continuous Deployment (CI/CD) pipelines. Figure 2 shows an example of a FaaS deployment. The inherent nature of FaaS applications, composed of multiple functions that can be monitored independently, makes the model particularly well-suited to validate our hypotheses.

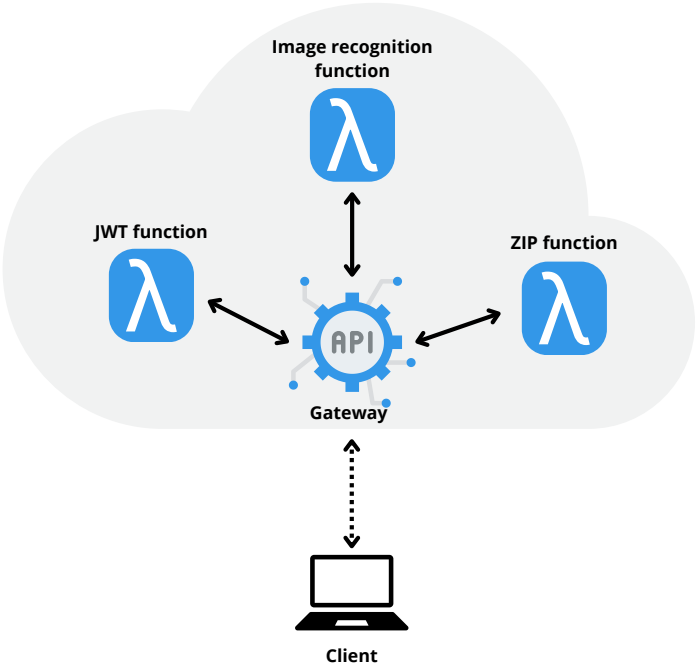


Figure 2: **An example FaaS deployment.** Clients send requests of function execution to the API gateway, which is in charge of calling functions and returning the results to the clients. Serverless functions can serve multiple purposes, in this case generating and signing JWT tokens [12], inferring a machine learning model and compressing files into a zip archive.

A significant part of our work involves identifying diversified and representative serverless workloads and benchmarks. In cloud data centers, serverless functions run inside orchestration frameworks that manage the deployment and network aspects. The biggest cloud providers, Google, Amazon and Microsoft all have their own serverless infrastructure. However, open-source solutions also exist for self-hosting, such as OpenWhisk [19] or OpenFaaS [18]. Several benchmark suites exist for these frameworks, providing valuable workloads and tools for evaluation. They are designed to run within a specific orchestrator and often include a small set of common functions. In this thesis, we propose a more comprehensive approach offering a larger number of functions with a greater diversity of characteristics. More precisely, we designed 22 serverless functions covering multiple programming languages (compiled and interpreted) and different behaviours (CPU intensive, file reading, etc.). Moreover, our approach being orchestrator-independent allows us to isolate and characterise the function performance alone, without measuring the energy consumed by the orchestrator itself.

Distinguishing the energy efficiency of our 22 functions requires us to actively monitor their energy consumption. However, measuring the energy consumption at the process level is not a trivial task. Software power meters are tools allowing to estimate individual processes' power usage. Most of them rely on the Running Average Power Limit (RAPL) interface [45], which embeds hardware counters reporting the energy consumption of the whole CPU. Based on these reports and other performance events measured per process such as CPU cycles and cache misses, tools like Kepler [37] or PowerAPI [40] distribute accurately the whole CPU power to individual processes. Unfortunately, RAPL is an Intel feature that is not present on Ampere CPUs. While Kepler claims that it also supports Ampere CPUs, we experimentally determined that this is not the case with our target ARM CPU for which the energy consumption estimations are way off the charts. To enable the support of process-level power monitoring on Ampere CPUs, we focus on the PowerAPI tool which is easier to extend thanks to its modularity. Our contributions include the PowerAPI support for Ampere CPUs.

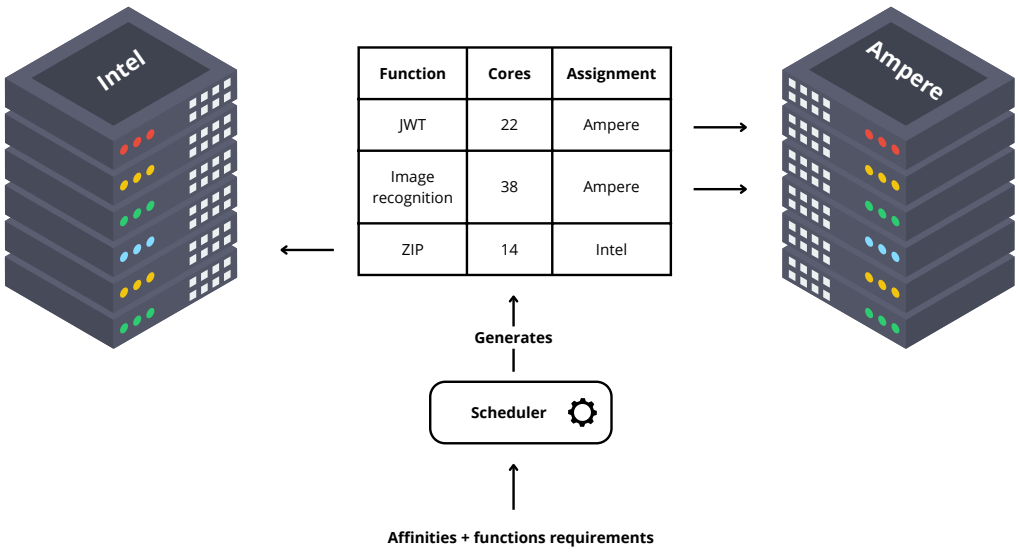


Figure 3: **Example of a scheduling decision taken by our energy-aware scheduler.** As input, it takes the energy CPU affinities and requirements for each function. Then, the scheduler finds an optimal function-to-machine mapping that minimises the total energy consumption. It also estimates the number of cores needed by each function to respect their requirements. The JWT function generates and signs a JSON Web Token. The Zip function compresses multiple files into a single archive. The Image recognition function identifies an object in a given image.

Our energy consumption measurements of our 22 serverless functions show that the Ampere CPU is the most energy-efficient for 21 of them in terms of number of executions per watt-hour. However, their percentage of affinity towards the ARM machine highly varies. To decide on an energy-efficient placement of our functions, we designed and implemented an energy-aware static scheduler. We use a constraint programming (CP) approach to model the different constraints, such as function

requirements or machine capacities. The CP solver finds an optimal assignment by minimising the energy required while respecting the requirements of all functions. Figure 3 shows an assignment example.

Data centers commonly use a Round-Robin or consolidation approach to choose the machines on which functions will be executed. Moreover, operators also strive to minimise the number of cores required to respect the functions' requirements. These scheduling strategies serve as a point of comparison to measure and evaluate the energy efficiency of our energy-aware scheduler. However, correctly comparing our scheduler to such implementations raised some concerns. Indeed, our approach to estimate the number of cores needed by each function to meet its requirements, expressed in requests per second, relies on apriori measurements. However, we demonstrate that such estimations can be erroneous due to resource contention. Consequently, as the function assignments change from one scheduler to the next, the total throughput observed might also differ. We tried different methodologies to cope with that variability, and we came up with a solution that highly reduces the gap in the number of executions performed. Our final solution involves both limiting the number of executions and restricting the duration of the experiments. Our results show that our energy-aware scheduler reduces the total energy consumed by 2 to 5% on average, and up to 19% in some specific configurations compared to common scheduling strategies found in cloud data centers.

Our work is divided into four chapters. Chapter 1 focuses on measuring energy consumption and describes our contribution for process-level power consumption estimations on Ampere CPUs. Chapter 2 continues with the design and implementation of our energy-aware scheduler. Chapter 3 describes our evaluation methodology and results. Finally, Chapter 4 concludes and gives some insights about future work possibilities.

Chapter 1

Measuring energy consumption

Nowadays, characterising and analysing energy consumption at various levels in the ICT sector is gaining more and more interest for economical and ecological reasons. But before understanding what composes our energy usage, we need to measure it. While coarse-grained tools and hardware devices such as power meters or connected outlets have existed for decades, there is a recent surge in interest for more fine-grained tools. With the growing enthusiasm for cloud technologies such as virtualisation and containerisation, measuring the consumption of each process, container, or application independently is a big challenge.

Process-level energy monitoring can be unnecessary for small devices running a single application. In these cases, measuring the CPU or machine power consumption can be a sufficiently precise metric of energy usage of processes if the devices run only a few of them. Nevertheless, the rise of data center servers equipped with hundreds of cores induces a shift towards massive application and service consolidation. In that case, process-level energy consumption monitoring becomes necessary to characterise each application independently. However, exactly measuring the energy consumption at the process level is an impossible task. Therefore, current tools rely on power usage estimations. These tools create power estimation models based on specific hardware characteristics of the CPU to infer individual tasks' power estimations from the global CPU power consumption. Developing such models requires insights into the CPUs' hardware specifications.

All the latest tools for process-level power consumption estimation support our Intel processor based on an x86 architecture. However, it appears that none of them can handle our Ampere Altra Max CPU which is built upon an ARM Neoverse-N1 architecture. Even though one tool claims to have the capability of reading the power consumption of Ampere CPUs through a Linux Kernel driver, its process-level and DRAM power estimations are still disastrous.

This chapter is organised as follows: Section 1 identifies and explains the latest frameworks and tools available for measuring energy consumption at different levels. Section 2 reports our inconclusive experiments with Kepler [37], a Kubernetes power-level exporter, while Section 3 describes our final methodology using the Power API

toolchain for estimating the power at the process level on both CPU architectures. This last section describes our contributions and includes our methodology for developing the support of PowerAPI on Ampere CPUs, together with the validation of our power model.

1 State of the art measurement tools

Energy consumption measurement and estimation are at the origin of numerous scientific papers and software tools. These tools cover energy usage at several granularities, from the power usage of the entire machine to estimations at the process level. We divide this section into two parts: coarse-grain tools allowing energy monitoring at the machine or whole-CPU level, and fine-grain tools enabling power estimations at the container/process level. Table 1.1 aggregates the various tools available for power consumption measurements and estimations at different granularities.

Tool	**Grain Level**	**Compatibility**	**Host**	**Export**
PDU	machine	any	physical	multiple
RAPL [45]	CPU DRAM	x86	bare metal	MSR sysfs
SMpro [4]	CPU DRAM	Ampere	bare metal	sysfs
Scaphandre [25]	processes	x86	bare metal VM	Prometheus
Kepler [37]	processes pods containers	x86 (Ampere)	bare metal VM	Prometheus
PowerAPI [40]	processes pods containers	x86	bare metal	multiple

Table 1.1: **Energy consumption exporter tools comparison.** Comparison of the tools available to measure/estimate power consumption at different grain levels. PDUs can be queried with multiple network protocols, depending on their compatibility and configuration. PowerAPI can export its estimations to various DBMS such as MongoDB or InfluxDB.

1.1 Coarse-grain tools

The highest-level tool that is available, in most data centers, is the connected Power Distribution Unit (PDU). These connected sockets track the global power usage of whatever is plugged in, such as computing servers in our case. Those connected PDUs are often accessible through network protocols, such as SNMP, to programmatically retrieve the current power usage, which can be used as a reliable ground truth. Of

course, this tool is not sufficiently precise to estimate the power consumption of single processes as it reports the whole machine's power consumption, including both the CPU and all other components.

At a lower level, the latest Intel CPUs are equipped with the Running Average Power Limit (RAPL) interface [45], providing two distinct functionalities for the various CPU components and the memory:

1. Limit or cap their power consumption.

2. Measure at a fine granularity and high sampling rate their power consumption.

RAPL supports multiple power domains, which are represented by Figure 1.1. Power domains refer to specific categories of components within the CPU that are monitored together, and the power exposed by a given domain includes the power of all its components. The package (PKG) domain evaluates the power usage of the entire socket, including both CPU cores and non-core components such as last-level caches and memory controllers. The two power planes (PP0 and PP1) measure respectively the consumption of all CPU cores and the consumption of the integrated graphics processor (GPU), if any. The DRAM domain, as its name suggests, tracks memory energy consumption. Finally, the Psys domain gauges the power of the entire System on Chip (SoC).

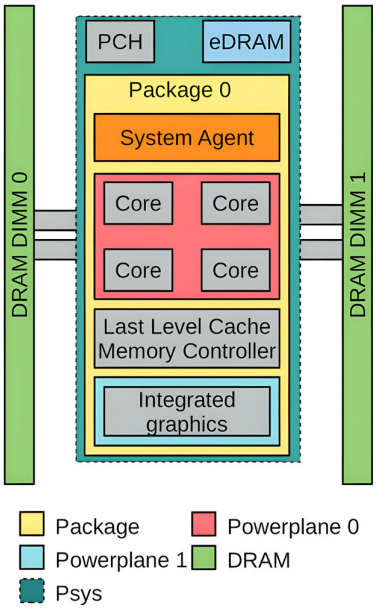


Figure 1.1: **Power domains of the RAPL interface**. Each power domain exports its own energy consumption independently. Illustration taken from [53].

For a given CPU, the available power domains depend on the processor's generation and might not all be present at the same time. The units of the different RAPL counters also vary according to the processor's generation. Fortunately, some

tools converts automatically the counter values into Joules, such as `perf` which is a performance analysing tool able to instrument CPU hardware counters, part of the Linux kernel since 2009. Desrochers *et al.* [47] and Khan *et al.* [53] respectively evaluated the accuracy of the RAPL DRAM domain and all RAPL domains showing a high correlation between RAPL readings and physical power measurements.

Ampere Altra CPUs expose similar counters to RAPL through capability registers, exposed via the System Management (SMpro) microcontroller [2]. Like RAPL, Ampere provides an interface to control power management functions using the Power Management (PMpro) microcontroller [2]. The power readings are also separated into three different domains, illustrated by Figure 1.2.

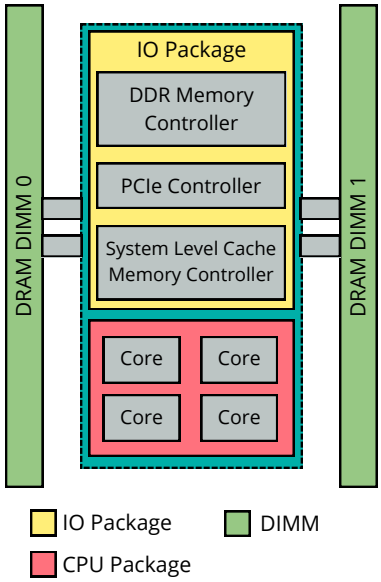


Figure 1.2: **Power domains of the Ampere SMpro interface.** Each power domain exports its own energy consumption independently. Illustration taken from [53] and adapted to the Ampere SMpro interface.

The CPU package reports the current power for all CPU cores. The measurements exported by the IO package contain the power of the SoC, PCIe controller, DDR memory controller and everything else labelled as non-core components. The DIMM package reports the DRAM power usage. The different counters expose measurements as the average power within the 5 last seconds, in mW. The reading of these counters is not yet implemented in the `perf` tool, but the `SMpro hwmon` [4] Linux kernel driver exposes them via the `sysfs` pseudo-filesystem. Unfortunately, no study has been yet carried out about the accuracy of these measurements.

1.2 Fine-grain tools

Using coarse-grain tools to measure the energy consumption of an entire machine or at the CPU level is sufficient for some use cases, such as monitoring the energy costs of server racks. Still, with the emergence of cloud technologies such as virtualisation or containerisation, one could want to monitor power usage at the Virtual Machine (VM) or process level. For instance, public cloud providers could benefit from tracking the power usage and thus the energy costs of individual users.

Accurately isolating the energy consumption of independent processes turns out to be non-trivial for two main reasons. First, it requires reliable system or CPU level ground truth measurements, or otherwise, it would distort the process-level estimations. This is not always available, especially for systems lacking the well-tested RAPL interface. Secondly, deriving the energy consumption of processes from the ground truth is not a straightforward one-dimensional regression task from processes' CPU utilisation only. Indeed, the power usage of the different processes often depends on a plethora of parameters including CPU clock cycles, cache misses and many others, that must be part of the regression to reliably estimate their consumption. We present three tools, Scaphandre [25], Kepler [37] and PowerAPI [40], that all use RAPL measurements as their ground truth for deriving per-process power consumption estimations.

Scaphandre [25] is a tool developed to monitor energy consumption at the process level. It accomplishes this task by analysing information located in `/proc/stat` and `/proc/PID/stat`, which contain detailed statistics about system processes, including data on CPU usage. Based on that information, it distributes parts of the whole CPU energy consumption to the different running processes. When the exporter requests an estimation of the power usage of the monitored processes, the sensor queries the RAPL interface, reads all `stat` files, and distributes the consumption according to CPU usage statistics found in `stat` such as the active user time per process. To compute the energy consumption of a whole application composed of multiple processes, they rely on Prometheus, a monitoring system and time series database [29]. Prometheus, having query language capabilities, allows aggregating measurements for different processes to export global metrics for the targeted application. Scaphandre's implementation details were not published in a conference or journal, and no evaluation of its accuracy is available in the online documentation. Scaphandre currently only supports Intel and AMD-based machines.

Kepler [37] (Kubernetes-based Efficient Power Level Exporter) is a framework to estimate the energy consumption of containerised applications. What differentiates at first Kepler from Scaphandre is what it collects and uses to estimate the energy consumption of each process. Kepler relies on an extended Berkeley Packet Filter (eBPF)-based approach to collect the hardware performance counters, such as CPU cycles, instructions or cache misses, for inferring per-process fine-grain estimations. With these exported metrics, it trains power models and infers the estimations in a self-calibrating and extensible way. Kepler claims to be compatible with Intel and AMD-based machines, but also Ampere Altra CPUs.

More details about our experiments and issues with Kepler are described in Section 2.

PowerAPI [40] is another toolchain that enables estimations of energy and power usage at the process level. Like Scaphandre and Kepler, it relies on the CPU energy metrics exposed by the RAPL interface. What differentiates PowerAPI from the two other tools is its modularity. The PowerAPI library itself is not what estimates the energy consumption, but rather plays the role of an intermediary. To be able to monitor the power usage of processes, one should plug a software sensor, responsible for collecting a pertinent set of raw events and metrics, and a formula, in charge of aggregating these metrics to infer energy usage at the process level. Together with the framework's core, the authors provide a sensor and a formula respectively named HwPC sensor [48] and Smartwatts [49]. The HwPC sensor collects RAPL metrics with other performance counter events such as core frequency while the formula trains one power model per layer of frequency. The formula performance evaluation shows that it has negligible monitoring overhead and estimates the energy consumption of the different processes with less than 3W and 0.5W of error for the CPU and DRAM respectively, compared to RAPL measurements. PowerAPI is compatible with Intel and AMD CPUs. The PowerAPI sensor and formula are more extensively detailed in Section 3.

All these tools that are compatible with Intel or AMD processors with an x86 architecture are reliant on the RAPL interface (except for Kepler which claims to support Ampere CPUs). Consequently, these tools only work with processors that have been produced after 2012. While some of those tools support virtual machines, most of the time the functionalities are limited and/or the metrics are less accurate, mainly due to the limited or lack of access to hardware metrics (through hypervisor).

2 Kepler

The first tool we found convincing at first was Kepler [37], a framework that offers a way to estimate power consumption at the process, container, and Kubernetes pod levels. Its main advantages are its pod-level precision of measurements and its claimed native support for Ampere CPUs. We were aware, nonetheless, that the paper presenting this tool was published only one month before we discovered it, and the fact that it was very recent also implied that it was somewhat immature which made it problematic to build and use.

2.1 Kepler framework

Kepler's operations to estimate the power consumption of containerised applications can be divided into two major parts: the collection of a bag of system power metrics followed by the training of a machine-learning power model based on the previously collected metrics to estimate the consumption of processes.

For optimal accuracy of the estimations, Kepler relies on bare-metal access to retrieve metrics from hardware components, such as the CPU and DRAM. On x86 machines, this is achieved through RAPL or the Advanced Configuration and Power Interface (ACPI). This approach allows Kepler to train a power model tailored to the specific hardware configuration, leading to the most precise power consumption estimates. However, Kepler can also function within virtual machines (VMs) commonly encountered in public cloud environments. In this case, a generic pre-trained power model is employed. While this approach is helpful in getting an insight into the energy efficiency of processes running within a VM, it suffers from less accurate results and limitations inherent to virtualised environments. For instance, the pre-trained model has no information about the number and size of virtual machines running in parallel on the CPU, which should be taken into account for better results.

Kepler collection of system power metrics

On a machine with bare-metal access, hardware counters are probed to assess resource utilisation, using CPU instructions to estimate CPU resource usage, collecting cache misses to estimate memory utilisation, and assessing Streaming Multiprocessor (SM) metrics to estimate GPU usage level (which is not used in our case). It gets access to those metrics through various APIs, such as Intel RAPL interface for CPU and DRAM power, NVIDIA Management Library (NVML) for GPU power, ACPI for platform power and more. Then, the metrics are stored in Prometheus. For the particular case of the Ampere Altra Max processor, there is an integrated piece of code able to retrieve the CPU and IO power consumption from a driver that is available in the Linux Kernel.

Kepler training of ML power model

The training phase begins with the retrieval of data from Prometheus using specific queries. It then leverages a multi-dimensional regression-based algorithm to train the model on both the energy consumption and the resource utilisation of system processes and containers. The model undergoes training iterations until it meets a satisfactory level of accuracy. Once this is done, the model can be queried to get an estimation of the power consumption of a process based on both the CPU's global power consumption and specific hardware performance events metrics.

2.2 Kepler support for Ampere

Kepler's DockerHub includes pre-built binaries for x86 CPUs that are fetched and run by Docker to start the Kepler framework. By default, no such binaries exist for ARM-based processors, making it not possible to be used out-of-the-box on our Ampere Altra Max CPU. Even though no helpful information resides in the project's documentation about ARM builds, we managed to use Kepler on ARM processors by building the project from source. We also designed the configurations and manifests for our Kind [13] cluster, a tool allowing the creation of Kubernetes clusters from Docker containers, and the Prometheus [29] exporter utilised by Kepler to estimate the power consumption of Pods present in our Kubernetes cluster. Kepler's energy

estimations are accessible through a Grafana dashboard which visually displays the Prometheus data.

Figure 1.3 shows the power values that were exported by Kepler on the machine embedding the Ampere Altra Max CPU. Those values are erroneous, the entire CPU package energy usage is estimated at a mean of 0.24W with a standard deviation of 0.21W which is invalid for a CPU with a minimum of 120W for the SoC power. Moreover, the power consumption of the "other" target, which includes all machine components except the CPU and DRAM obtained from ACPI, is unreasonably high as it reaches values above 10 billion watts. The DRAM is always showing 0 watts of power consumption because Kepler is not able to read those metrics on Ampere Altra CPUs.

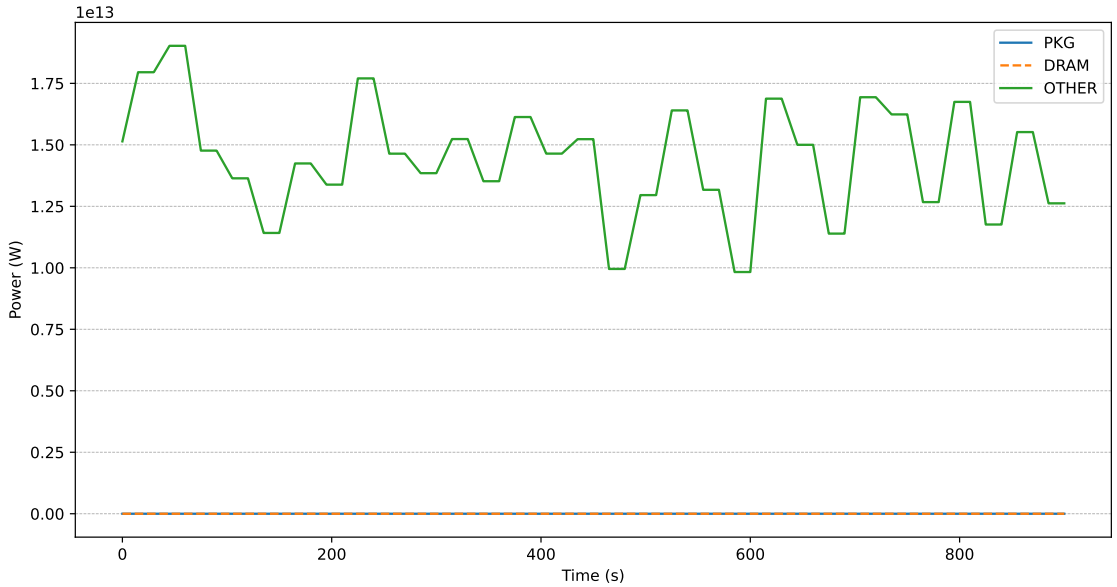


Figure 1.3: **Power consumption estimation of Kepler.** Erroneous energy consumption of the Ampere Altra Max CPU and machine's DRAM measured using Kepler in an idle state of the CPU.

We have multiple hypotheses for such faulty results. First, Kepler reads the SMpro Linux kernel driver but probably treats it as an equivalent value to RAPL, which is an incorrect assumption since RAPL and SMpro do not provide power in the same measurement units. Moreover, we suspect that Kepler's model is unable to learn and infer from utilisation metrics such as cache misses and CPU frequencies. Indeed, many Intel performance events are not available on Ampere, or available but at a different location under a different name, or not exactly measuring the same metrics or in the same measurement unit. Therefore, we contacted the Kepler team via a GitHub issue where we detailed our goal, our methodology to run Kepler and our results so far. We got a few ineffective answers and nothing to help us fix the erroneous results on Ampere Altra CPUs. Moreover, the recentness and constant evolution of the Kepler project were problematic for our experiments because we had to adapt our methodologies and configurations to the frequent changes made to

the project. To carry out our experiments we needed a more mature tool. From this point, we focused our attention on PowerAPI and implemented our own support for the Ampere Altra Max CPU.

3 PowerAPI

PowerAPI is a software library designed to monitor energy consumption at the process level. Unfortunately, due to its implementation being based on the RAPL interface and other Intel-based performance counters, it is not directly compatible with the Ampere CPU family. We decided to incorporate support for the Ampere Altra Max CPU and use PowerAPI for our measurement for two main reasons:

1. We had the chance and the opportunity to discuss and exchange with the developing team of the PowerAPI library from the University of Lille and Inria
2. Their infrastructure model was more mature and easier to extend than Kepler

We start by discussing the PowerAPI model and some implementation details. Then, we present our contribution: the extension and support of PowerAPI for Ampere Altra CPUs. We finish with multiple experiments to validate our implementation.

3.1 PowerAPI architecture

The PowerAPI model is composed of four modules, illustrated by Figure 1.4.

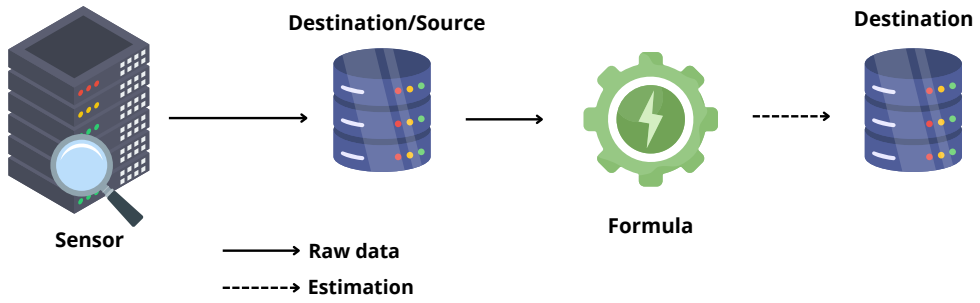


Figure 1.4: **PowerAPI architecture high-level view.** The sensor collects and exports raw data from diverse CPU hardware events to a database. The formula aggregates these data to estimate the energy consumption at the process level, exported to another database.

The four building blocks are the following:

- **Sensor**: responsible for collecting and exporting to the source the raw values needed to estimate the energy consumption at the process level. It must run on the same host and at the same time as the monitored processes. The other three modules can run on a separate machine.
- **Source**: a database service to store the data exported by the sensor.

- **Formula**: responsible for analysing the raw values exported by the sensor. From this data, it estimates the energy consumption of each monitored process and exports them as aggregated data to the destination.
- **Destination**: a database service to store the data exported by the formula.

3.2 The PowerAPI sensor

The sensor is the first important part of the PowerAPI framework. Its role is to collect the values needed by the formula to estimate the energy consumption of each process. The data collected by the sensor is aggregated in the form of HwPC reports and needs to match what the formula expects. For this thesis, we use the Hardware Performance Counters (HwPC) Sensor [48] developed by the PowerAPI team, but one could extend or implement another version of this sensor suited to its needs.

Sensor configuration

The HwPC sensor can export a wide variety of metrics, which must be specified in its JSON configuration file. The configuration file of the sensor accepts the following fields:

- `frequency`: the frequency (in ms) at which the sensor will export reports. The performance counters' values attached in the report correspond to their mean or accumulation over the last sampling period. We use the default value of 1000 ms.
- `output`: custom JSON object describing the destination where the HwPC reports are stored. PowerAPI supports many types of destinations such as MongoDB, CSV files, or InfluxDB. In our case, we work with MongoDB.
- `system`: the system field includes all the events/counters that one wants to monitor system-wide. Additionally to the counter names, one can specify in the configuration their monitoring type: only one CPU (core) per socket or all CPUs per socket. In our case, we collect RAPL power counters and core frequency counters.
- `container`: complementary to the system field, the container field includes the events that must be monitored per process.

The sensor's official documentation [21] provides a more detailed description of its configuration. The full configuration that we use for our sensor on Intel can be found in Appendix 1.1.

System events

The first category of events monitored by the sensor are system events. As explained above, the metrics are collected system-wide.

The first system event we export in our sensor configuration is the RAPL PKG energy counter. As this performance counter measures the energy consumption of the whole CPU socket and not single cores, we configured PowerAPI to monitor only one core per socket to avoid duplicates. It is worth noting that another counter called RAPL DRAM is also available to monitor memory energy consumption, but we do not use it in our case.

The two other exported events are the MPERF and APERF frequency counters, which are Model-Specific Registers (MSR) available on the latest x86 processors. They behave in the following way:

- MPERF increments at a rate equal to the maximum frequency of the CPU
- APERF increments at a rate equal to the actual frequency of the CPU

Since core frequencies can fluctuate significantly (e.g., only half might be active), we set the monitoring type to all cores per socket for these events. Otherwise, if only a few cores are active and are not part of the monitored ones, the average frequency could misleadingly reflect an idle state. Conversely, if the few monitored cores are idling and all the others are under load, the average frequency could misleadingly reflect a maximum load state.

Container events

Container events are the second type of events that are monitored and exported by the sensor. Those events can be any of the ones available for monitoring with the Linux `perf` tool. Like the system events, the container events are monitored globally, but also per process, allowing the formula to estimate the part that each process plays in the total energy consumption. For the formula to distribute the energy consumption as accurately as possible, the set of chosen events to be monitored must have the highest correlation with the monitored processes' power usage. We leveraged the events that the PowerAPI team has already discovered to have the highest correlation and best results for the x86 architecture. The events with their description according to Intel's documentation [10] are the following:

- `CPU_CLK_UNHALTED:REF_XCLK`: "counts core crystal clock cycles when the thread is unhalted".
- `CPU_CLK_UNHALTED:THREAD_P`: "counts the number of thread cycles while the thread is not in a halt state".
- `LLC_MISSES`: "counts core-originated cacheable requests that miss the L3 cache".
- `INSTRUCTIONS_RETIRED`: "counts the number of instructions retired".

Events and process monitoring

The reading of these events is performed by the `perf` API of the Linux kernel. To monitor the container events for each process individually, the sensor takes advantage of the Linux `cgroups v1` (control groups). Control groups is a Linux kernel

feature to isolate processes for resource allocation but also for fine-grain resource monitoring. The sensor monitors by default the `perf_event` control group, but one could configure another default cgroup path in the sensor configuration. The sensor also discovers at runtime the different Docker containers and Kubernetes pods to include their control groups in the events monitoring.

One must be aware that the number of events to be monitored simultaneously by the sensor is capped. Modern microprocessors have only a limited set of programmable hardware performance counters that can be used at the same time. If the number of available counters is exceeded, then the events will be monitored using *multiplexing*, in a round-robin fashion. Multiplexing can highly deteriorate the measurement accuracy of the different counters and consequently disturb the final energy consumption predictions at the process level. As the number of monitored events must be restricted, the importance of choosing the most relevant set of performance counters to monitor is even greater.

3.3 The PowerAPI formula

The formula is the second building block of the PowerAPI framework. It gathers all the reports published by the sensor to estimate the energy consumption of the monitored processes. We use the latest PowerAPI formula called SmartWatts [49]. Smartwatts is defined as a self-calibrating software-defined power meter, with the goal of providing fine-grain power consumption estimations at the container/process level.

Formula configuration

Just like the sensor, the Smartwatts formula needs to be configured in a JSON file and must include the following fields:

- `input`: custom JSON object describing the source where the formula will fetch HwPC reports to estimate energy consumption. This object must match the `output` field of the HwPC sensor configuration. We use MongoDB.
- `output`: custom JSON object describing the destination where the Power reports are exported. In our case, we use InfluxDB.
- `cpu-base-freq`: the CPU base frequency, in MHz. This is used in the estimation of the current core frequency.
- `cpu-tdp`: the CPU Thermal Design Power (TDP), in watts. This is used by the formula in the power consumption estimation.
- `cpu-error-threshold`: the acceptable error threshold for the estimation. That field is described more in detail in the next section.
- `sensor-reports-frequency`: the frequency at which the sensor exports its reports. This must match the sensor configuration.

The Smartwatts official documentation provides more details about the different configuration fields available [22]. Our configuration for the formula on Intel can be found in Appendix 1.1.

Smartwatts power consumption estimation

In order to estimate the power consumption of each process, the formula maintains at runtime a set of power models denoted M^f. The formula is able to maintain two such sets, one for the CPU consumption and one for the DRAM consumption, but we focus on the processor power usage. From these models, the Smartwatts formula estimates the global consumption of the host ($\hat{p}$) and the consumption of all containers/processes ($\hat{p}(c)$).

Smartwatts builds at runtime the set of power models M^f with f belonging to the set of frequency layers F. When the formula needs to estimate the current energy consumption of the host or a set of processes, it first computes the average running frequency as follows:

$$F_{avg} = F_{base} \cdot \frac{\Delta APERF}{\Delta MPERF} \tag{1.1}$$

where F_{base} is the CPU base frequency specified in the formula configuration. Every power model is built from an *Elastic Net* regression, which is a regular linear regression with combined L1 and L2 priors as regularisers. With a given set of raw performance events counters values $E^f = [e_0, \ldots, e_n]$, the estimated power consumption of the host is computed as follows:

$$\hat{p} = M^f\left(E^f\right) \quad \text{with } f \in F \tag{1.2}$$

Smartwatts takes as ground truth the power values exported by RAPL (p_{rapl}). Consequently, the estimation error ε is computed as:

$$\varepsilon = \mid p_{rapl} - \hat{p} \mid \tag{1.3}$$

When the computed error exceeds the given threshold in the formula configuration, the current power model for the frequency layer f is discarded and a new model is trained from the latest samples. Concerning the power estimation of the different containers or processes, more steps are needed to have an accurate estimation. First, we can predict the raw power consumption of any process c by applying the model M^f with the set of events collected at the process scale ($E(c)$):

$$\hat{p}(c) = M^f\left(E^f(c)\right) \quad \text{with } f \in F \tag{1.4}$$

As the intercept i of the model is included in the estimated power consumption of each process c, the formula distributes the intercept proportionally to the dynamic power usage part of c:

$$\tilde{p}(c) = \hat{p}(c) + i \cdot \left(\frac{\hat{p}(c) - i}{\hat{p} - i}\right) \tag{1.5}$$

Finally, the formula can compute a confidence interval on the estimation for every process c by computing the ratio over the global observed error:

$$\varepsilon(c) = \frac{\tilde{p}(c)}{\hat{p}} \cdot \varepsilon \tag{1.6}$$

Two differences between the original Smartwatts paper [49] and the current implementation must be noted:

1. Originally, the formula was able to monitor the set of events collected and to estimate their relevancy by computing the Pearson coefficient between the events and the global power consumption reported by RAPL. The most relevant performance counters were kept for training the different models. The PowerAPI team told us that they removed this feature as the set of relevant events was always the same on the set of machines on which they evaluated their formula.

2. Additionally, the formula was able to isolate the static (idle) power consumption and deduct it from the estimations. According to the developing team, this was removed for mainly two reasons. First, extracting the idle power consumption is a challenging task, which often leads to incorrect results. Secondly, removing the idle power from the estimations lacked interest from the users. Most of them consider that the static consumption must be distributed across every process as these processes are the reason why the machine is running and consuming that idle portion. In the current implementation, the idle power consumption is distributed across all processes.

3.4 Extending PowerAPI to support Ampere CPUs

PowerAPI only supports CPUs equipped with the RAPL interface because its estimations solely rely on its counters as a baseline for energy consumption. Fortunately, Ampere processors expose similar metrics, at least as it is described in their documentation. Our hypothesis for integrating Ampere Altra CPU support into PowerAPI is the following: if we manage to find equivalent events/counters to the ones exported by the HwPC sensor on Intel, then we could benefit from the already built Smartwatts formula to estimate the energy consumption at the process-level. We need to find three types of equivalences:

1. an equivalence to the RAPL counters;

2. a way to collect the average frequency per core;

3. a pertinent set of `perf` events, as much correlated with the reported CPU-level power as possible.

For the following sections and the validation of our findings, we will use the Linux `stress-ng` tool [15]. The tool can launch various workloads, but we focus on the matrix option that stresses the CPU with multiple levels of loads. Our methodology to validate the equivalences is the following:

- Run the `stress-ng` tool at different (increasing) load levels and export at the same time the readings of the counters (on Intel and Ampere).
- Compare the results of the counters used by PowerAPI on Intel with the results of supposedly equivalent counters on Ampere.

RAPL equivalence

The first step towards the PowerAPI support for Ampere CPU is to have a ground truth for deriving the energy consumption of each process. Fortunately, as stated in Section 1.1, Ampere Altra CPUs provide such hardware counters that in theory could correspond to the ones exposed by the RAPL interface. PowerAPI uses the RAPL PKG domain, which includes the consumption of the cores and non-core components. Our hypothesis, which we need to verify, is that the sum of the CPU and IO domains on the SMpro interface corresponds to the RAPL PKG domain.

To have a first validation of our hypothesis, we consider the metrics exported by RAPL on Intel, which have already been validated by multiple researchers, as our point of comparison. Our first experiment uses the `stress-ng` tool in matrix mode. It starts by loading the machine at 10%, and increases the load by 10% every 45 seconds, leaving 30 seconds of idle time between each benchmark, up to a load of 100%.

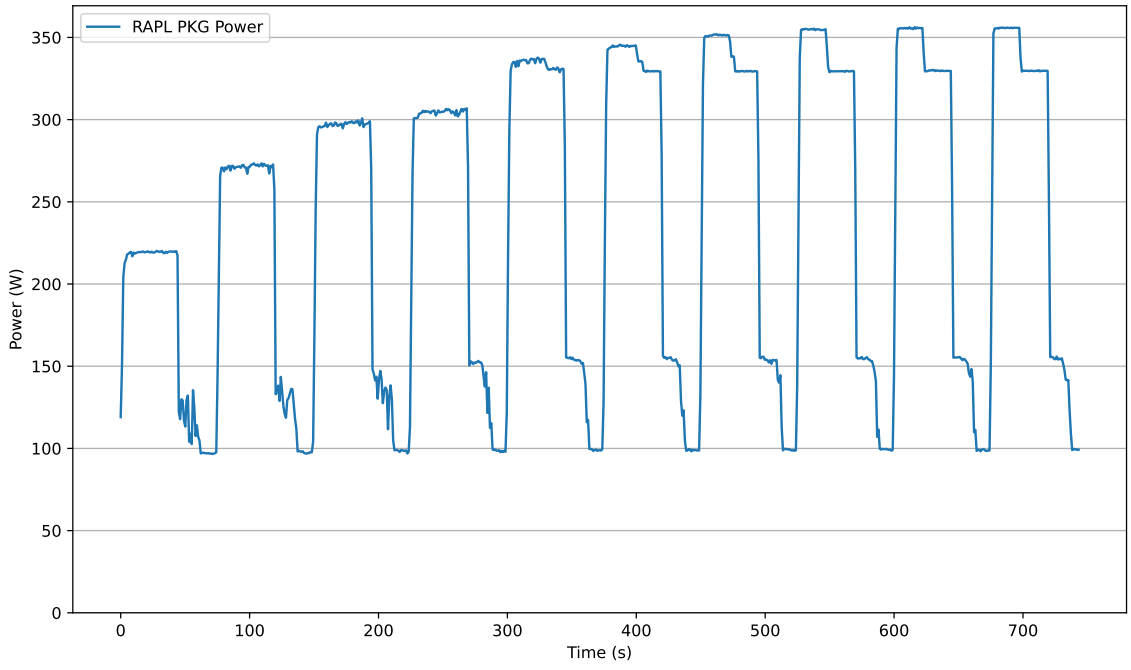


Figure 1.5: **Intel RAPL counters values** for the PKG domain under different levels of load over time with Hyper-Threading enabled. The load starts at 10% and increases by 10% every 45 seconds up to a load of 100%. There is an idle time of 30 seconds between each change of load.

Figure 1.5 shows the values of the RAPL PKG domain counter under different levels of loads over time. As the RAPL values are reported per socket, the values represented in the figure correspond to the sum of the values on both sockets. We can observe that the Intel CPU energy consumption is growing almost proportionally to the load level up to a load level of 50%. After this threshold, the power usage of the CPU is nearly constant (but still increasing slightly) up to a load of 100%. While it could seem weird, this is mostly due to Intel's Hyper-Threading / Simultaneous Multi-Threading (SMT), which allows a single physical CPU core to handle two tasks (threads) at once. Indeed, at a load of 50% all the physical cores are likely busy, but the CPU can still squeeze in more work by using those two threads per core. This additional workload increases the power usage slightly, but not as much as adding a whole new physical core would.

One particular thing to notice is that the power usage reported by RAPL under the maximal load is roughly equal to 350 watts, which corresponds to the sum of both sockets' TDP of our Intel CPU. One could think that having an exported CPU power under a maximal load equal to the TDP proves that the reported values by RAPL are correct. However, the TDP value reported by the CPU manufacturers is only an indication of the maximum amount of heat dissipated by the processor. We should consequently take this information with a grain of salt, but it is still a good indication of correctness.

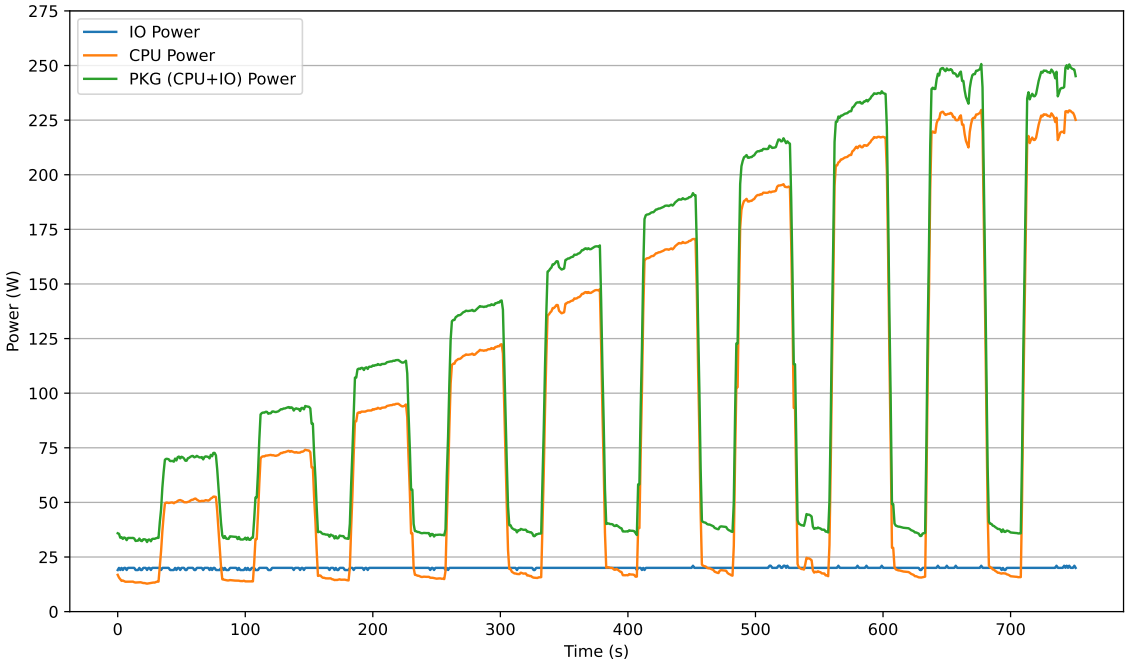


Figure 1.6: **Ampere SMpro counters values** for CPU, IO and PKG (CPU+IO) domains under different levels of load over time (no Hyper-Threading/SMT). The load starts at 10% and increases by 10% every 45 seconds up to a load of 100%. There is an idle time of 30 seconds between each change of load.

Figure 1.6 represents the same experiment, but this time on the Ampere Altra Max CPU with the metrics exposed by the SMpro interface. A first observation is that the IO domain exhibits a relatively constant power consumption across different levels of load. This is expected as our stress workload focuses solely on the CPU, minimising the impact on disks, network cards, and RAM usage (only a few additional megabytes are used). Secondly, like on Intel, the value of the PKG domain under maximal load is almost equal to the processor's TDP of 250 watts. On top of that, we can observe that the CPU power usage is almost proportional to the CPU load level. Indeed, the curve is mostly linear, as opposed to Intel's one, which could be explained by the absence of Hyper-Threading or SMT on Ampere chips. Given the reliability of the RAPL interface and the coherent results from SMpro, the next objective is to be able to read the average core frequency on the Ampere CPU.

Average core frequency

As the Smartwatts formula trains one different Elastic Net model per layer of frequency, we have to find a way to extract the instantaneous average frequency of all cores. Unfortunately, MPERF and APERF, the two Model Specific Registers used by the formula on Intel processors, are not available on Ampere Altra CPUs. However, the Linux Kernel includes the Collaborative Processor Performance Control (CPPC) driver [1]. The driver exposes several metrics of interest via the `sysfs` pseudo-filesystem (one per core). What is interesting for us is the `feedback_ctrs` event. According to the Linux Kernel documentation, the `feedback_ctrs` event is defined as follows:

- `feedback_ctrs`: "includes both Reference (REF) and Delivered (DEL) performance counter. Reference counter ticks up proportional to the processor's reference performance. Delivered counter ticks up proportional to the processor's delivered performance."

These counters look like they are measuring the same metrics as the MPERF and APERF counters on Intel. We thus define the average current frequency as follows:

$$F_{avg} = F_{base} * \frac{\Delta DEL}{\Delta REF} \tag{1.7}$$

with F_{base} the base clock frequency of the processor. To validate these counters as a reliable source for our frequency measurements, we again use the `stress-ng` tool, in matrix mode, with different levels of load. We start at a load level of one core (one core computing matrix multiplications in a loop), doubling every 30 seconds, up to 128 cores (full utilisation of the machine).

Figure 1.7 shows the average frequency of all cores for each level of load computed with the metrics exposed by the CPPC Linux driver. We know from the Ampere Altra Max CPU documentation that the base and maximum clock frequencies are respectively 1.0 GHz and 3.0 GHz. What we observe in our experiment results are very consistent. When there is a small load, less than 16 cores active, the average frequency of all cores stays relatively close to the base clock speed, 1 GHz. However,

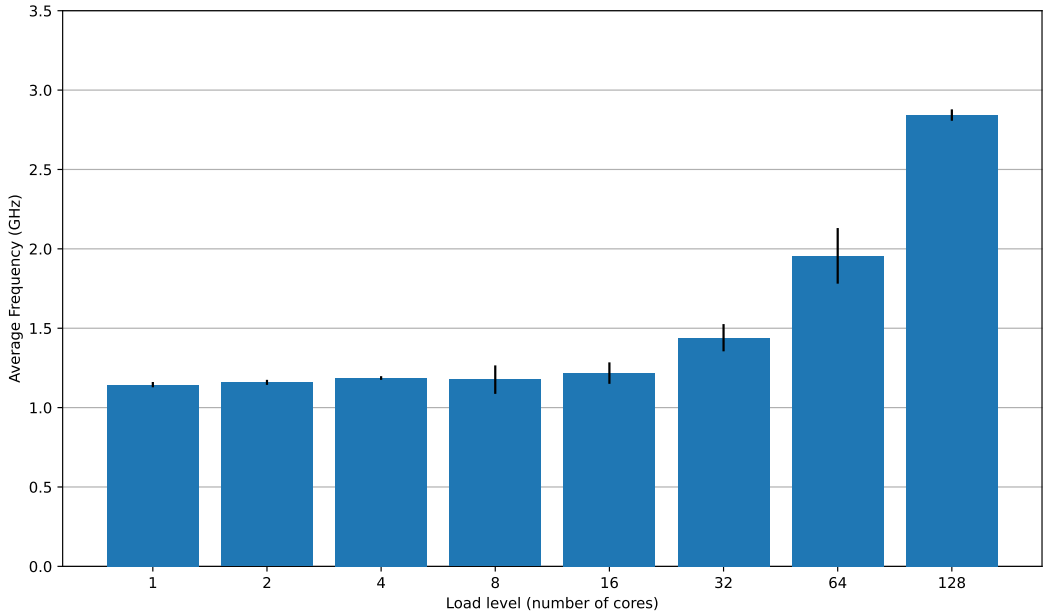


Figure 1.7: **Average frequency of all cores.** The frequency of each core is computed using Equation 1.7, based on the ACPI CPPC driver values. The average frequency is reported under different levels of loads, i.e. the number of cores at 100% load on the Ampere Altra Max CPU.

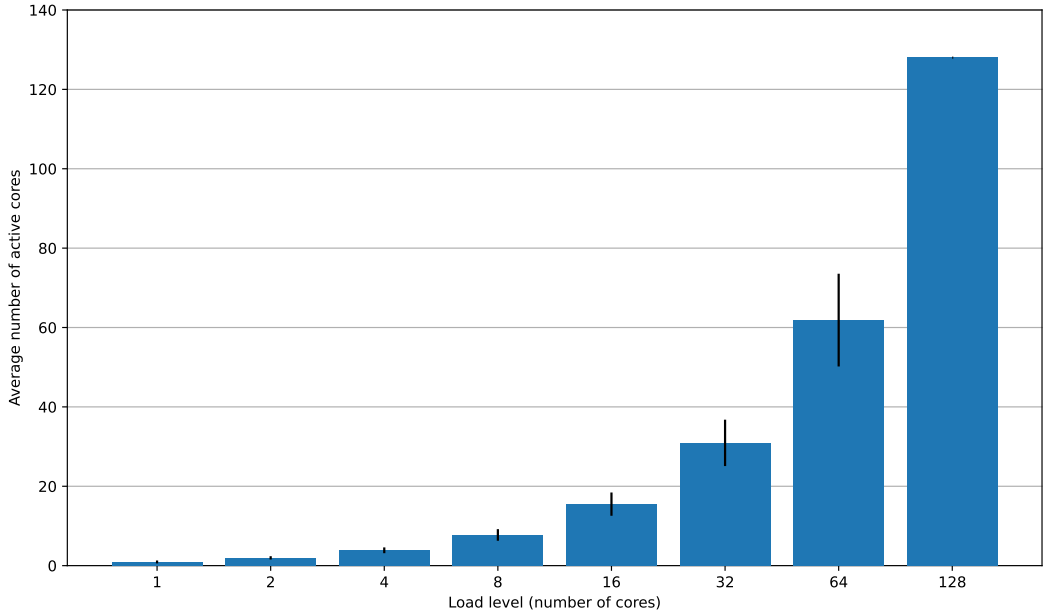


Figure 1.8: **Average number of active cores** (current frequency above 2.5 GHz). The frequency of each core is computed using Equation 1.7, based on the ACPI CPPC driver values. The average number of active cores is reported under different levels of loads, i.e. the number of cores at 100% load on the Ampere Altra Max CPU.

when we fully load the machine using its 128 cores, we observe that we are getting closer to the CPU's maximum clock speed of 3 GHz. Figure 1.8 depicts the number of cores that we estimate "active" at each level of load. We consider a core as "active" if its average frequency over the last sampling period is higher than 2.5 GHz. Ideally, we should observe for each level of load x (x cores), a relatively equal number of active cores.

As expected, the number of active cores is quite similar to the load level. We can notice that the standard deviation is growing larger up to 128 cores. This can be explained by our sampling interval period, which is 1 second. Indeed, jobs could be scheduled to another core during this period, slightly disturbing our results. Since the PowerAPI Smartwatts formula works with layers of frequency, and that the layer is computed based on the average frequency of all cores, this should not disturb our power models.

Hardware performance events

Together with the raw CPU energy consumption and the average core frequency, the Smartwatts formula relies on a set of performance events to distribute the energy consumption reliably between all monitored processes. The list of performance events that could be retrieved on each machine can be found in the vendors' documentation.

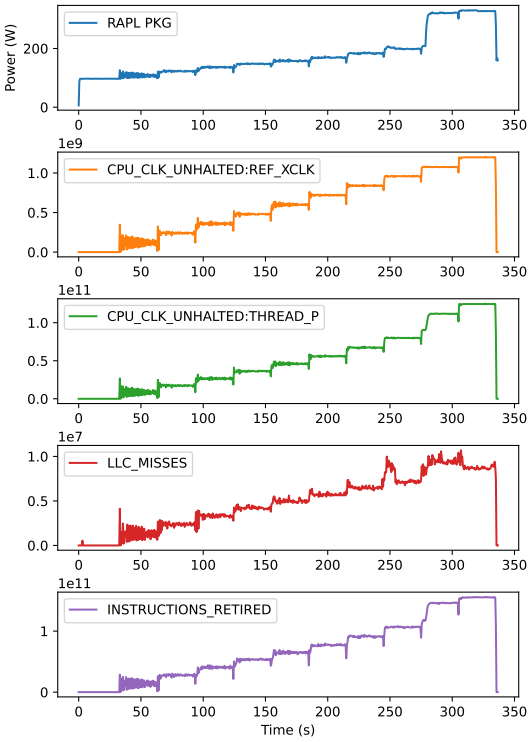


Figure 1.9: **Performance events measurements.** Measurements of the 4 different hardware performance events used by PowerAPI on Intel with a CPU stress-ng load increasing over time (0 to 100%).

A noticeable difference between Intel and Ampere CPUs that we have at hand, is that there are a lot more events available on Intel. Moreover, some events counting the same metrics are found under a different name on each machine while other events are available on one machine or the other, but not both. For example, there is an Intel event counting active CPU cycles whereas Ampere does not provide such an event. However, Ampere provides two events, one counting the stalled CPU cycles due to resource constraints and another counting the stalled CPU cycles due to the lack of operations to execute. These two events together report somewhat a kind of the opposite of Intel's single metric. However, we should be cautious about such assumptions. Verifying them requires knowledge about the exact counting behaviour of these special registers, which is not always available in the vendors' documentation.

In order to have a ground truth to base our measurements on, we decided to observe the behaviour of the four performance events collected by PowerAPI on Intel. We again use a CPU stress going from 0 to 100%, while exporting at the same time the RAPL PKG power consumption and the performance counters. Figure 1.9 shows our results. At first sight, we can directly observe how related the different events are with each other, but also with the power exported by RAPL. If we compute the correlation between the events and the RAPL PKG metric, we observe a correlation ranging from 90% to 97%. There is also a strong correlation among the different events, from 96% to 99%. The full correlation matrix is illustrated by Figure 1.10. We can also notice that the event's values when the CPU is at a 10% load are quite less stable.

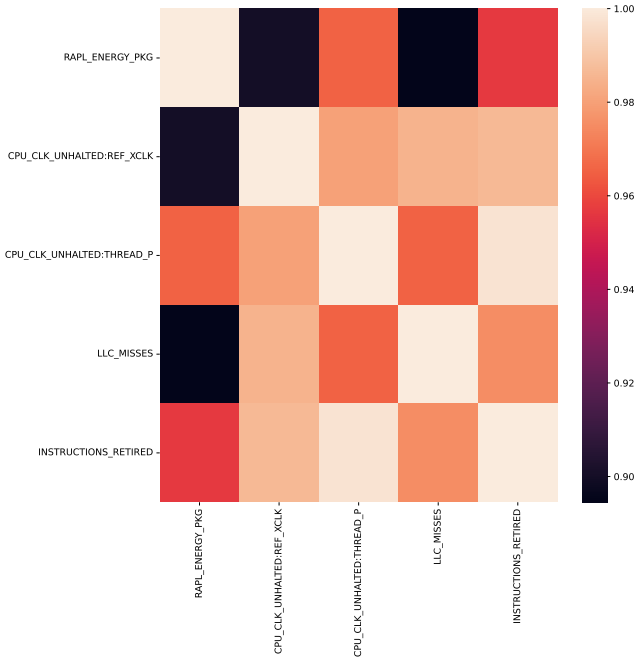
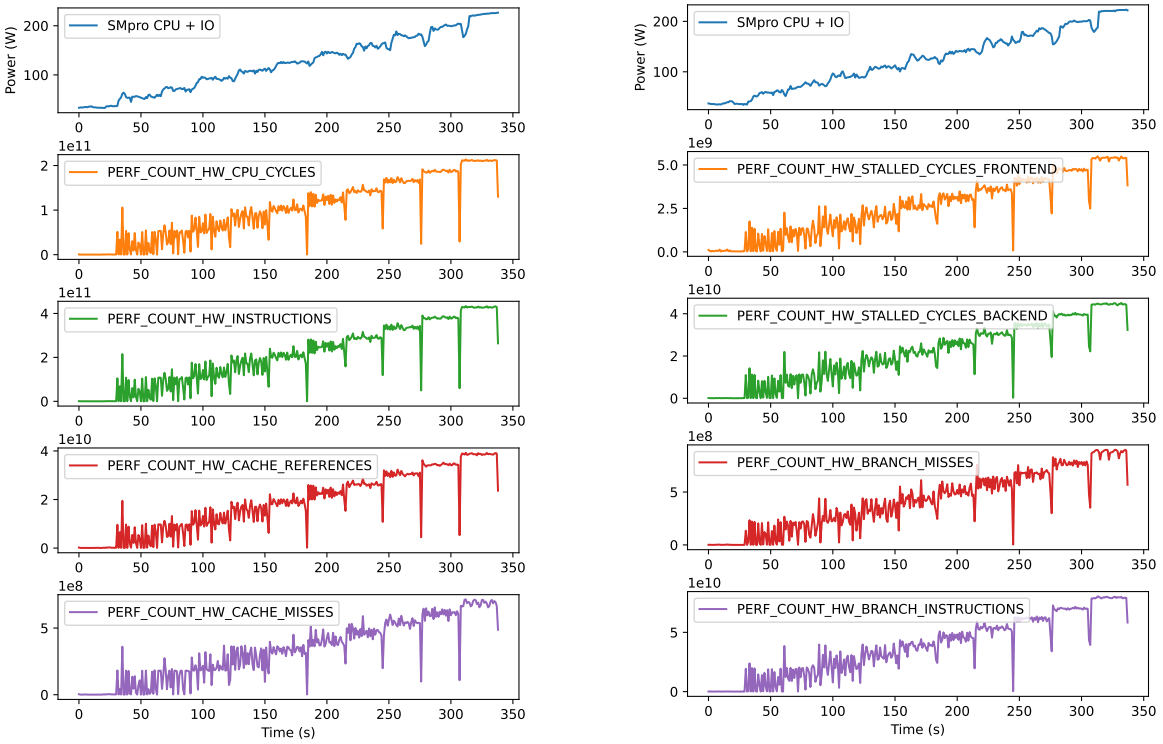


Figure 1.10: **Correlation matrix** of the different performance events collected on Intel by the sensor together with RAPL measurements.

In order to find the best set of events to be used by the formula on the Ampere CPU, we decided to test 8 hardware performance events that seem to be the closest to what the formula uses on Intel, at least based on what the documentation says about these events. To avoid `perf` multiplexing, we separate the eight events into two distinct sets of four events. Figure 1.11 shows our measurements on both sets.



(a) First set of events

(b) Second set of events

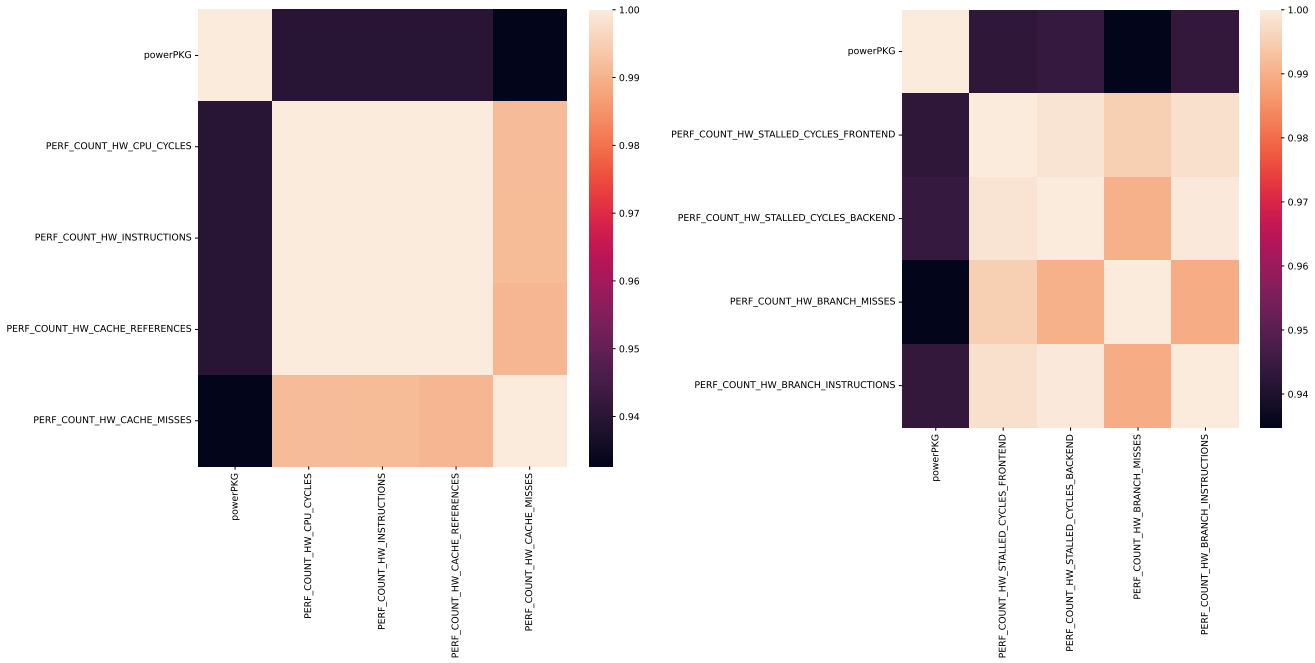
Figure 1.11: **Performance events measurements.** Measurements of 8 hardware performance events available on the Ampere Altra Max CPU with a CPU stress-ng load increasing over time (0 to 100%). The events are collected with two separate runs to avoid perf multiplexing. The drops are due to a change in the load level.

Here again, the different events seem to be highly correlated. The performance counters are correlated by more than 98% between themselves and by approximately 95% with the SMpro (PKG) power metric. The two correlation matrices are represented by Figure 1.12. However, what we observe here that was not the case on Intel is a relatively high variance in the performance events' values under 50% of load. This is definitely something that we should keep in mind for the validation of our global implementation.

As all the tested events seem to be correlated enough with the target and that they all carry the same information, we select the events that are the closest to what PowerAPI uses on Intel. Our formula configuration on Ampere contains the following three events:

- `CPU_CYCLES`: "Counts CPU cycles".

- `INSTRUCTIONS_RETIRED`: "Instruction architecturally executed".
- `STALLED_CYCLES_FRONTEND`: "No operation issued because of the frontend. The counter counts on any cycle when there are no fetched instructions available to dispatch".



(a) First set of events

(b) Second set of events

Figure 1.12: **Correlation matrices** of the two sets of different performance events collected on Ampere by the sensor together with SMpro measurements.

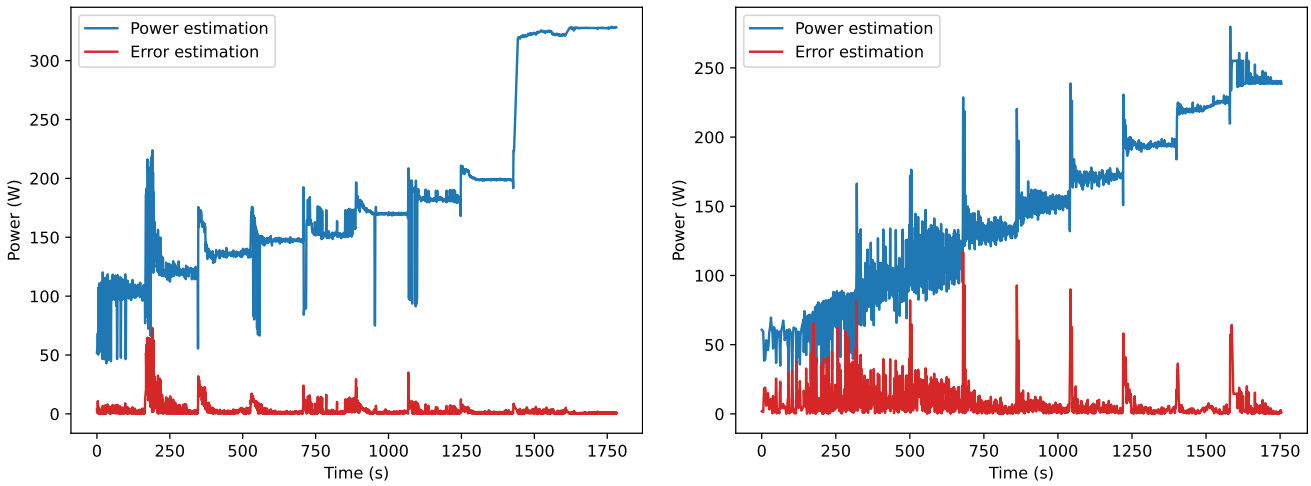
Both configurations for our sensor and formula on Ampere can be found in Appendix 1.2.

3.5 Validation of the PowerAPI support for Ampere

With our collected set of hypothetical equivalence on what the PowerAPI formula uses on Intel, we can modify the sensor to export what we need and adapt the formula to match what the sensor collects. A few things needed to be modified to make the sensor and the formula work on Ampere, in addition to changes in the configuration file:

1. The HwPC sensor on Intel reads all events, including RAPL and MSR counters through the Linux `perf` API. This was not suitable for our needs as the ACPI CPPC frequency readings and the SMpro interface are not yet integrated in `perf`. We thus modified the sensor to read the different counter values via the `sysfs` pseudo-filesystem.
2. The formula also needed some modification to work with the correct units as the metrics exposed by RAPL and SMpro are of different units.

Before trying to measure multiple processes at the same time, we start by measuring the energy estimation of one process over time. Like with the previous experiments, we launch a `stress-ng` CPU load, starting at 10% up to 100%. In order to have a first insight into the accuracy of our model, we export the estimation error over time. Figure 1.13a reports the error over time using the PowerAPI Smartwatts formula on Intel.



(a) Intel

(b) Ampere

Figure 1.13: **Measurements of the estimation error** in watts of the Smartwatts formula on both CPUs with a CPU stress-ng load increasing over time (0 to 100%).

We can make several key observations. First, the estimation error tends to decrease when the load increases. This is something that we discussed with the PowerAPI development team and that is due to how the formula deals with idle energy consumption. Under a small load, the idle power usage is distributed to the monitored process even if that process is not really consuming that energy. When the load increases, the idle part tends to diminish. Another observation is that the estimation error tends to converge to 0 watts over time for each load level. This is a feature of the formula that self-calibrates itself when the reported error is too high (we configured that threshold to 2 W). Figure 1.13b presents the same experiment's results when using our custom-built sensor and formula for Ampere.

The first noticeable difference with the Intel implementation is the estimation error below 50% load. The estimation error is higher and does not converge efficiently to 0 watts. However, our results above that 50% load threshold seem better. The estimation error is lower and with a nice convergence to a reliable stable model. Our hypothesis for this behaviour comes from the results that were presented in Figure 1.11. We can observe that the performance counters when under 50% of load are not stable enough to make the model able to correctly distribute the energy consumption to each process. Nevertheless, if we look at the power estimation made by our formula, even if the estimation varies a lot, taking its mean over time seems to correlate with the power exported by the SMpro interface.

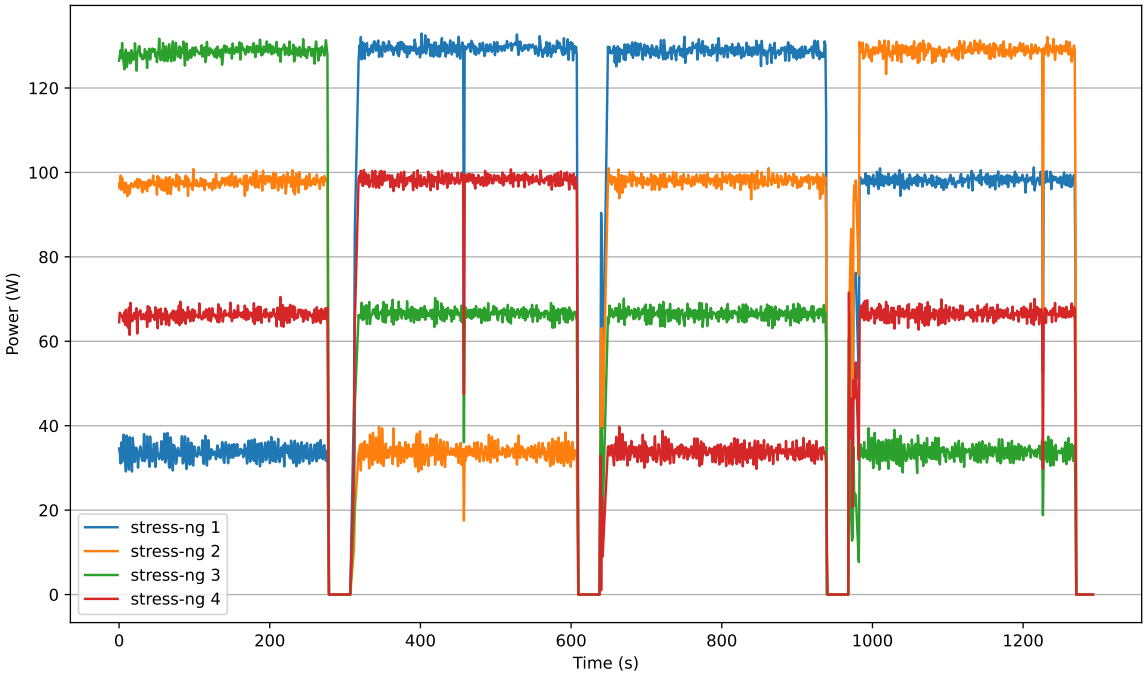


Figure 1.14: **Power consumption estimations.** Measurements of the power consumption estimations of the Smartwatts formula on Intel with 4 CPU stress-ng loads of 10, 20, 30 and 40% randomly assigned every 4 minutes.

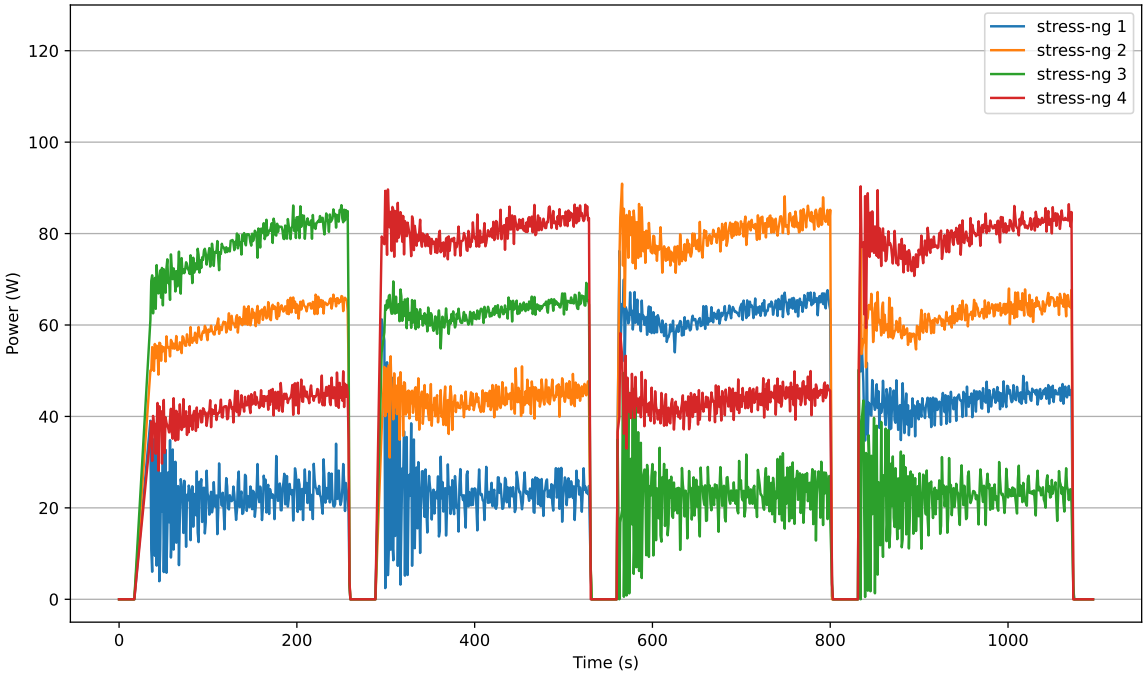


Figure 1.15: **Power consumption estimations.** Measurements of the power consumption estimations of the custom Smartwatts formula on Ampere with 4 CPU stress-ng loads of 10, 20, 30 and 40% randomly assigned every 4 minutes.

To validate further our implementation, we can look at its behaviour when multiple processes running simultaneously must be monitored. To perform such experiments, we create four distinct processes. Each process runs for 4 minutes, 4 times, leaving 30 seconds of idle period between each run. After each idle period, the 4 different processes are assigned a load of 10, 20, 30 or 40% randomly. Figure 1.14 and Figure 1.15 shows our results respectively on Intel and Ampere.

We can observe a clear difference in the model stability. The Intel implementation has much less variability, but when the load is higher, the Ampere model tends to stabilise. We think that this is still due to the high variability in the values exported by `perf` under light load. In our previous experiments, we measured that the power usage of the CPU under 100% load was roughly equal to the TDP values, 350 W for Intel and 220 W for Ampere. If we sum up the mean power consumption of the different processes, we obtain approximately 335 W on Intel and 210 W on Ampere, which is close enough to the expected values.

To improve the accuracy and stability of our implementation, we could try to fine-tune the different power models to cope with the variability in the `perf` events. However, the results presented are precise enough for us to monitor the average power consumption of various applications/services.

Chapter 2

Energy-aware scheduling in heterogeneous computing systems

Task scheduling decisions can have high impacts on performance, but also on energy efficiency. To reduce energy consumption, cloud operators often rely on consolidation to minimise the number of powered servers. On top of that, we believe that it is possible to lower the energy consumption of a set of tasks or functions by leveraging hardware heterogeneity. For this purpose, we designed and built an energy-aware scheduler upon a constraint programming (CP) modelling language and used a CP solver to generate an optimal static allocation of processes to available machines. Our scheduler creates an optimal assignment by considering process requirements and apriori energy measurements on all heterogeneous devices while respecting their capacity constraints. To compare the benefits of apriori energy measurements over consolidation, we describe an extension of our model minimising the number of required cores.

This chapter is organised as follows: Section 1 introduces the concept of energy-aware task scheduling by mentioning the state-of-the-art techniques and algorithms in heterogeneous environments. Section 2 describes the implementation of our schedulers built upon the Minizinc [56] solver-independent constraint programming language. We define the variables, constraints and possible use cases for our models.

1 State of the art scheduling algorithms

Over the last decade, researchers extensively studied the problem of achieving energy-efficient task or function scheduling in heterogeneous cloud or edge data centers. Even though all studies focus on developing energy-aware placement algorithms, they differ in several ways, from different constraints such as task ordering and dependencies, to how they model the energy consumption of the different tasks and how they design diverse placement strategies and heuristics. Our literature review focuses on heterogeneous computing systems, which add a layer of difficulty to the scheduling problem. Indeed, the same task executing on heterogeneous hardware will have a different impact in terms of energy efficiency and resource utilisation. Table 2.1 provides a comprehensive overview of energy-aware scheduling solutions in

heterogeneous environments.

Source	Algorithm	Energy model	Scheduling type	Task dependencies
[55]	Duplication-based algorithm	Single frequency busy/idle power model	Offline	Dependent
[54]	Probabilistic	Voltage and processor-specific parameter	Offline	Independent
[57]	Score minimisation	Assume linearity between resource utilisation and energy consumption	Online	Independent
[60]	Score minimisation	Measurements (PDU)	Online	Independent
[46]	Improved Cuckoo search	Voltage and processor-specific parameter	Offline	Dependent
[42]	List-based algorithm with task prioritisation	Voltage and processor-specific parameter	Offline	Dependent
[59]	Score minimisation	Measurements (RAPL)	Online	Independent
[52]	Task prioritisation, processor-merging heuristic and DVFS	Voltage and processor-specific parameter	Offline	Dependent

Table 2.1: **Comparison of multiple energy-aware task schedulers in heterogeneous computing systems.** The comparison includes the algorithm used, how the schedulers model the energy consumption, whether the solutions work online or offline and if the authors consider independent tasks.

1.1 Problem definition

Even with the broad diversity of energy-aware function or task schedulers, the problem of assigning tasks to heterogeneous computing nodes according to a specific target objective can be described commonly in high-level terms. Given a tasks set $T = \{t_1, \ldots, t_n\}$ with requirements $R = \{r_1, \ldots, r_n\}$ and a set of heterogeneous processors $P = \{p_1, \ldots, p_m\}$, one must find a feasible assignment A according to a set of constraints C that maps each task t_i to a processor p_j while minimising/maximising an objective function $O(A)$.

A first distinction between the different energy-aware task scheduling algorithms is their definition of a task. The optimal task placement according to an objective function depends on how the tasks interact. Some models assume that all tasks are independent (the execution order does not matter), and do not communicate between them [54, 57, 60, 59], while others presume that the different tasks might have dependencies and communications, leading to a more complex model which needs additional precedence constraints [55, 46, 42, 38]. The dependency-free models regroup use cases such as the Function as a Service (FaaS) model or basic task queues while the others can represent microservices applications or pipelined jobs. Additionally, schedulers can perform online or offline scheduling. Offline scheduling runs the algorithm

once on an apriori known static task set, whereas online scheduling accepts tasks arriving dynamically and sometimes reconsiders previous placement decisions in time.

Almost every energy-aware scheduler has an objective function minimising the total consumed energy, but some models combine that energy efficiency with additional objectives. As an example, Rocha I. *et al.* [60] introduce an extra parameter H set by the client which represents an acceptable trade-off between energy efficiency and performance. However, having an energy-aware scheduler does not necessarily mean that it implements an energy minimisation objective. For example, Chen J. *et al.* [42] model their problem as a schedule length minimisation while respecting some energy consumption constraints. Additionally, energy-aware schedulers are also vastly found in the edge computing field, because of the low-powered and power-constrained characteristics of edge devices [38].

1.2 Energy models

Another important difference between energy-aware schedulers is how they estimate tasks' energy performance profiles or compute the full system power usage. Panda S. *et al.* [57] and Mei J. *et al.* [55] propose two simple power models. Panda S. *et al.* [57] propose an energy model based on resource utilisation and processing time assuming a linear relation between utilisation and energy consumption. They define the energy consumption E_{jk} of a resource j at time k as:

$$E_{jk} = (p_{max} - p_{min}) \cdot UV_{jk} + p_{min} \tag{2.1}$$

with UV_{jk} the resource utilisation and p_{min} and p_{max} respectively the power consumption at the lowest and highest level of load. On the other hand, Mei J. *et al.* [55] propose a simplified energy model by disabling Dynamic Voltage and Frequency Scaling (DVFS). They base their model on the CPU specifications documentation, which reports the idle power (at the lowest frequency) denoted P_{idle} and the busy power (at the highest frequency) denoted P_{busy}. Consequently, they compute the total energy consumption E as follows:

$$E_{total} = P_{busy} \cdot t_{busy} + P_{idle} \cdot t_{idle} \tag{2.2}$$

with t_{idle} and t_{busy} the respective time spent in both frequency layers.

Li K. *et al.* [54] and Deng Z. *et al.* [46] also base their power model on CPU specifications, but taking into account the different intermediate frequency layers. They assume that the processors are based on the CMOS technology and that the power consumption is dominated by the dynamic power dissipation P_d defined as follows:

$$P_d = CV^2F \tag{2.3}$$

with C the effective switched capacitance, V the supply voltage and F the CPU frequency. Since the supply voltage V and the frequency F are proportional, the equation can be simplified as:

$$P_d = \lambda V^3 \tag{2.4}$$

where λ is a processor-dependent parameter. Finally, they compute the dynamic (E_d), idle (E_i), and total (E) energy consumption by measuring the makespan of each task. The dynamic energy consumption E_d is:

$$E_d = \sum_{k=1}^{M} \sum_{T_i \in U_k} \lambda_k V_{kr}^3 B_{ikr} \tag{2.5}$$

where M is the set of processors, U_k the set of tasks running on processor k, V_{kr} the rth voltage supply level of processor k and B_{ikr} the processing time of task T_i on processor k. The idle power consumption E_i is defined as:

$$E_i = \sum_{k=1}^{M} \left(\left(Ms(G) - \sum_{T_i \in U_k} B_{ikr} \right) \lambda_k V_{kh(k)}^3 \right) \tag{2.6}$$

where $Ms(G)$ is the total makespan and $V_{kh(k)}$ the minimum supply voltage of processor k. Finally, the total energy consumption E is:

$$E = E_d + E_i \tag{2.7}$$

Lastly, Rattihalli *et al.* [59] and Rocha I. *et al.* [60] base their power estimations on physical measurements. They respectively use the RAPL interface and PDU outlets to monitor and model beforehand the power usage profile of their specific targeted hardware at different levels of loads.

1.3 Algorithms and heuristics

Together with their problem definition and how they model energy consumption, the different energy-aware schedulers also differ in how they search for a good feasible scheduling decision. The implementations also highly vary in terms of constraints and scale considered. However, some similarities can be noted such as the use of task prioritisation for deadline-sensitive task schedulers. Most algorithms work in a three-stage manner.

The objective of the first phase is to estimate the completion time or resource utilisation of the tasks waiting to be scheduled. During this step, Panda S. *et al.* [57] try to estimate the characteristics of the task such as execution time or resource usage by looking at the task requirements and by looking at the computing nodes' current load levels. Other algorithms [42, 52] for dependent and time-sensitive task scheduling perform task prioritisation. They formulate the task dependencies as a graph and compute the task priorities based on the *upward ranks* and *downward ranks* originally defined by Topcuoglu H. *et al.* [62]. The idea behind these ranks is to assign a higher priority to a task that is the closest to its entry node and the furthest from its exit node on its dependencies' critical path.

During the second phase of the algorithms, the different tasks and possible assignments are assigned a score. Again, the scoring function used always depends on the use case or the different constraints of the modelled problem. Some scoring functions

are simple, such as considering the sum of the normalised maximum completion time estimations [42]. Other algorithms perform additional heuristic scoring. Hu B. *et al.* [52] propose a processor-merging heuristic. The heuristic works in multiple iterations. At each iteration, based on the resource requirements estimations, the heuristic verifies if the requirements can still be met while turning off one processor. The procedure continues until no processor can be turned off without exhausting the resources.

The third step simply consists of assigning resources to the different tasks, according to the previously computed scores. The result of this phase is a task to processor mapping. In some online schedulers, such as the one presented by Rocha I. *et al.* [60], the task placement is reconsidered every now and then. Behaving this way improves performance as the best scheduling decision might not be available at task assignment time.

In this three-stage pipeline, the score assignment incorporates the energy-efficiency factor. Multiple solutions directly include the energy consumption estimations in the score to be minimised [55, 46, 60], but others strive to optimise the total makespan which results in lower power usage as a side effect [52]. Some scoring functions also include both energy and makespan [57, 54]. Several implementations even describe energy efficiency in their constraints, i.e. they allow a maximum amount of energy that can be consumed by a given task [42, 38].

2 Energy-aware scheduler design

In this section, we explore the constraint programming (CP)-based static scheduling component of our thesis. We present the variables, constraints and optimisation objectives written in the MiniZinc [56] constraint modelling language. Our problem formulation is then translated to and solved by the Gecode [61] CP solver. We decided to opt for a CP approach for multiple reasons. Taking advantage of existing efficient solvers let us focus on the modelling of the optimisation problem. Moreover, CP models are convenient due to their intuitive representation using variables, constraints and objective functions. This clear structure allows effortless integration of additional considerations. For example, if a given task must run on a specific hardware for compatibility reasons, it can be easily added to the set of constraints.

We designed two models both creating an optimal assignment of serverless functions to available machines with different characteristics. The first one, called Energy-aware, minimises the total energy consumed while the other, called Cores-aware, minimises the number of utilised cores. A practical use case for the second model could be a cloud provider wanting to consolidate as many functions as possible on a limited number of CPU cores, whereas the first model aims at minimising electricity consumption. A hypothesis that we evaluate with the Cores-aware scheduler is whether reducing the total number of CPU cores used across the cluster of heterogeneous machines can improve its overall energy efficiency.

2.1 Problem formulation

We consider the case of static assignment, where a bag of tasks must be mapped to physical resources. Our scheduler focuses on independent tasks, closest to the serverless or FaaS paradigm. Where we differentiate the most from the state-of-the-art is our approach to model the energy consumption. The research community relies on coarse-grain power measurements or estimations while our approach apriori estimates the power consumption of our different tasks at the process level on heterogeneous hardware.

More formally, our problem is defined as follows. Our scheduler takes as input a set $F = \{f_1, \ldots, f_n\}$ of n independent serverless functions and a set $P = \{p_1, \ldots, p_m\}$ of m heterogeneous processors. Each function f has a requirement R_f which states the expected number of requests per second that the function must be able to serve. We define the set R as our input vector representing the functions' requirements. Together with that input vector, the scheduler needs apriori profiling data for the set of functions F. The profiling data must be obtained by monitoring every function on each heterogeneous node of the cluster. For each function j running on node i, we must first collect an estimation of the number of requests per second each core can handle (E_{ij}). Secondly, we must monitor the energy consumption of each function running separately. We define W_{ij} as the average amount of Wh needed to perform one execution of function j on node i.

With its inputs and the monitoring dataset, the model will find a feasible solution that maps every function $f \in F$ to a specific node $p \in P$, while minimising an objective function O.

2.2 Constraint programming model

Our Energy-aware and Cores-aware models share most of the variables and constraints but vary on the objective part. Both models need 2 constant inputs: we define m as the number of machines and n as the number of functions. They also need an additional constant input for each machine, we define C_i as the total number of CPU cores available on machine i.

Our models contain $m + 1$ arrays of variables. The first array, defined as A, contains variables representing the number of cores assigned to each function. The m other variables arrays, numbered from D_1 to D_m, contain boolean decision variables, indicating whether or not the function is assigned to a given machine. For example, a fixed boolean variable such as $D_{1i} = 1$ indicates that function i is assigned to run on the node/machine number 1. These m arrays contain the decision variables that will be fixed by the CP solver to create our final assignment.

Our model is composed of multiple constraints to ensure feasible solutions. The first constraint enforces that a function must be allocated to exactly one of the

machines. The constraint is defined as follows:

$$\sum_{i=1}^{m} \left(D_{ij} \right) = 1 \quad \forall j \in [1, n] \tag{2.8}$$

We also ensure that the sum of cores allocated to each function on a specific machine must not exceed the number of CPU cores of the machine:

$$C_i \geq \sum_{j=1}^{n} \left(D_{ij} \cdot A_i \right) \quad \forall i \in [1, m] \tag{2.9}$$

Lastly, we must constraint that the variables in the array of cores assigned (A) correspond to the number of cores needed to execute each function depending on the number of executions requested and on the machine to which it would be allocated. This constraint is modelled as follows:

$$A_j = \left\lceil \sum_{i=1}^{m} \left(D_{ij} \cdot \frac{R_j}{E_{ij}} \right) \right\rceil \quad \forall j \in [1, n] \tag{2.10}$$

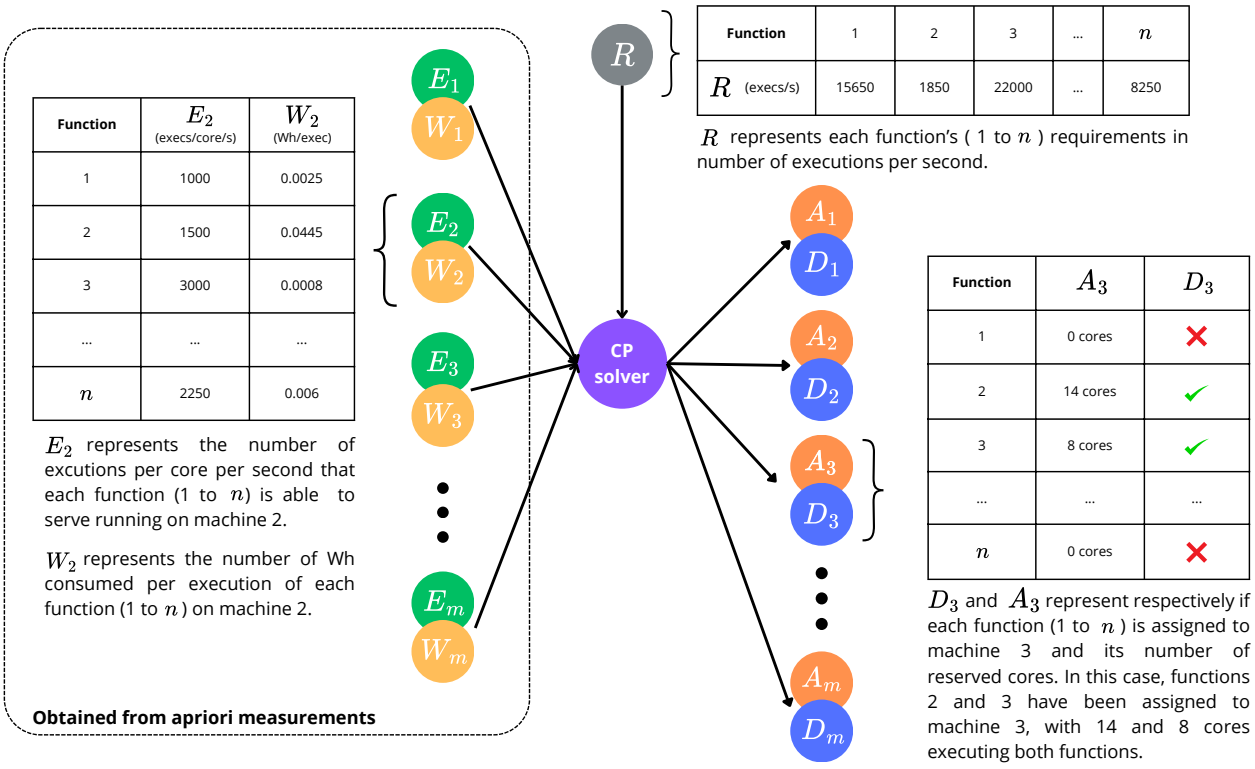


Figure 2.1: **CP-based scheduler workflow.** The CP solver takes three different inputs: the function requirements expressed in executions per second, the function performances expressed in executions per core per second, and the function energy metrics expressed in the number of Wh consumed per execution. The solver finds a feasible assignment of functions to machines.

With our input constants, variables, and constraints established, the next step is to specify the objective functions for minimisation by the CP solver. Since we have two models, one optimising the energy consumption and another one aiming at reducing the number of cores in use, we need two separate objective functions. For the Cores-aware model, we define the following objective function:

$$O : \sum_{i=1}^{m} \Big(\sum_{j=1}^{n} \big(D_{ij} \cdot A_j \big) \Big) \tag{2.11}$$

For the Energy-aware model, we define the following objective function:

$$O : \sum_{i=1}^{m} \Big(\sum_{j=1}^{n} \big(D_{ij} \cdot W_{ij} \cdot R_j \big) \Big) \tag{2.12}$$

Finally, we have two models that take inputs, ensure constraints and minimise the objective functions.

Chapter 3

Evaluation

To evaluate the efficiency of our Energy-aware scheduler compared to common cloud scheduling approaches and our Cores-aware scheduler, we start by introducing the serverless benchmarking infrastructure that composes our evaluation framework. We explain our choice of shifting from FaaS (Function as a Service) open-source orchestrators, such as OpenFaaS [18] and their corresponding benchmarking suites, to an orchestrator-independent methodology. Particularly, we detail one of our contributions which is the implementation of 22 representative and diverse serverless functions. Then, we present and define the collected performance and energy metrics that characterise our functions and we give rationales for choosing them. We also mention the methodology used to measure those metrics on the Intel and Ampere machines and discuss the relevance of the obtained results.

We follow with a detailed description of our experimental setup. We name and describe the multiple categories of functions we created and we explain the behaviours of the various basic schedulers used as a point of comparison with our Energy-aware and Cores-aware schedulers. We also explicate some choices we made to overcome constraints linked to the allocation of functions to fixed CPU cores. We detail the evaluation metrics used and annotate their formal mathematical definitions. At the end of the chapter, we disclose the results of our experiments.

This chapter is organised as follows. Section 1 details our initial experiments with OpenFaaS, our final set of 22 serverless functions, our methodology to collect the metrics needed by our Energy-aware and Cores-aware schedulers and some insights about the collected data. Section 2 describes our experimental setup, what schedulers we compare, and what metrics we use to evaluate its performances. Finally, Section 3 shows our final results, demonstrating that our Energy-aware scheduler saves on average 2 to 5% of energy, and up to 19% in some configurations.

1 FaaS orchestration frameworks and benchmarking

This first section describes our implementation choices and methodology to craft our metrics dataset, characterising the performance and energy profiles of a set of serverless functions. We start by describing existing Function as a Service (FaaS) frameworks and benchmarks, and how we arrived at our final set of functions. We continue with some implementation choices about the structure of the functions, followed by our methodology to create our dataset and finish with some insight into the collected data.

Our initial focus was on microservices-based applications. That type of application often relies on orchestrator frameworks, such as Docker Swarm [7] or Kubernetes [14], which decide on an assignment of service containers to nodes. However, efficiently profiling the energy consumption of microservices is a complex task. The interdependent nature of these applications often leads to a situation where only a few services receive enough traffic to accurately assess the energy performances of all microservices. More details and results about our experiments with microservices applications can be found in Appendix 2.

1.1 The FaaS cloud computing paradigm

FaaS is a type of cloud computing service. It is often exploited through cloud providers following a pay-per-use scheme but it can also be self-hosted. It allows to run specific pieces of code, called functions or serverless functions, without having to worry about the servers running them. The typical use case of FaaS is on-demand or event-driven functionalities, enabling the infrastructure to be turned on only when needed thus not incurring fees when not in use. Its advantages are the inherent high scalability of functions, the cost efficiency of the pay-per-use scheme removing the need for over-provisioning, and the improved developer's velocity as they can focus on the application logic instead of the infrastructure.

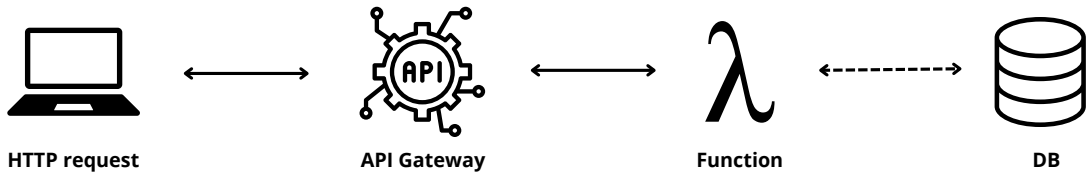


Figure 3.1: **Serverless model high-level view.** The end-user sends an HTTP request to the API gateway that runs the requested function and returns the computation's result. Functions can optionally be attached to a database, for example for storing and reading files.

FaaS orchestrators

All tech giants such as Google, Microsoft and Amazon have their own serverless infrastructure, but self-hosting solutions also exist. For this task, two well-known open-source projects have been developed to run a serverless engine in a local context of one or multiple physical or virtual machines, namely OpenFaaS [18] and OpenWhisk [19]. OpenFaaS was our initial choice for its native support for ARM CPUs as well as its ease of installation and use, and its large number of supported programming languages to write functions. However, it appeared to be not suited to our objectives, and we decided to move to an orchestrator-agnostic approach, running all functions independently packed as Docker containers without network connections.

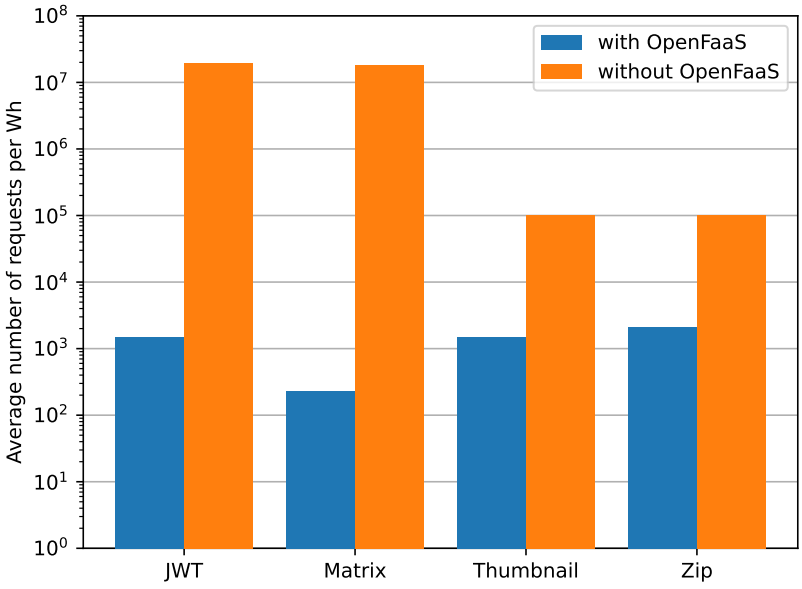


Figure 3.2: **Illustration of the cold-start problem with the OpenFaaS framework.** Comparison of the average number of requests performed per Wh of four different functions written in Python. When running inside the OpenFaaS framework, the functions are triggered with HTTP requests. Outside the framework, the functions perform as many executions as possible independently. The CPU load during both experiments was near 100%.

Multiple reasons explain our choice of shifting from serverless orchestrators to independent containerised functions. The most important one is that orchestrators have different policies in terms of how they start, run, pause and stop containers hosting functions. This leads to the cold-start problem that is well-known in the literature, and dealing with that concern is out of scope for this work. Figure 3.2 shows the differences in terms of the number of executions per watt-hour of four functions running within and without the OpenFaaS framework. The differences observed (up to a factor of 10^5 more executions per Wh) demonstrate that the OpenFaaS orchestrator induces overhead. More specifically, a lot more execution

time is spent on forking a new Python process and reloading the libraries upon each new user request than actually running the function code itself. By running the functions in isolation, we ensure to correctly assess their respective energy performances without the noise induced by an orchestration framework.

Function	Languages	Description
El Passo	C++ Web Assembly	It runs the tests of the El Passo library [63]. It is a privacy-preserving Single Sign-On system implementing PS Signatures [58].
HTML	Python Node.js Java	Read a file containing an HTML template and fill it with jinja2 [11] on Python, Mustache [36] on Node.js and FreeMaker [9] on Java
Image recognition	Python	Read an image of a flying airplane and recognises the airplane using the AlexNet model of Torchvision [30].
JWT	Python Node.js Java	Create a JWT token [12] using PyJWT [23] on Python, jsonwebtoken [34] on Node.js and Java JWT [33] on Java.
Matrix NumPy	Python	Generate two 30x30 matrices of random integers and perform a dot product of these using NumPy [17].
Matrix pure	Python Node.js Java	Generate two 30x30 matrices of random integers and perform a dot product of these without external library, only native code
PageRank	Python	Compute a page rank on a graph of 500 vertices and 250 edges based on the Barabasi-Albert model using igraph [44].
Thumbnail	Python Node.js Java	Decode an image in base64 to resize it and encode it back to base64, using Pillow [20] on Python, sharp [26] on Node.js and the native Java ImageIO.
Video to GIF	Python Node.js	Transform a video file into a resized GIF file by calling an ffmpeg [8] command.
Zip	Python Node.js Java	Compress two small files and a large file into a common zip archive using native Python, JSZip [35] on Node.js and native Java utils.

Table 3.1: **Description of our 22 serverless functions.** Each function represents a different use case and is implemented in one or more programming languages. Their textual description is given. A concrete implementation example of one of our serverless functions that generates and signs JWT tokens is presented in Appendix 3.

FaaS benchmarks

Most benchmarks for serverless functions target public cloud infrastructures such as AWS Lambda, Google Cloud Functions, Microsoft Azure Functions or IBM Cloud Functions. Only a few benchmarks exist for open-source frameworks such as OpenWhisk and OpenFaaS. Also, most benchmarks do not aim at measuring functions' CPU or energy usage but rather focus on network metrics such as the latencies for warm and cold starts, or the impacts of resources scaling. To cope with that lack of diversity and use cases, we developed our own functions and corresponding workloads extending the SeBS [43] serverless applications benchmark suite. The suite already provides interesting function examples such as a thumbnailer function in Python, an HTML templating function in Python and Node.js as well as a Python page-rank computing function. Table 3.1 details the 22 serverless functions we implemented for benchmarking, along with their programming languages. These 22 functions were not chosen arbitrarily. Two of our important criteria in the selection and implementation of our benchmark functions were diversity and representativeness.

Our set of functions is representative of real-world serverless use cases, from web services such as JWT tokens generations, HTML template filling or page-rank computations to media processing such as thumbnail resizing or video to GIF compression. But more importantly, our functions expose diversity. Diversity is found in the multiple used programming languages (Python, Java, Node.js, C++ and Web Assembly), but also in the behaviour of the different functions. Some of them are highly and only CPU intensive (e.g., Matrix, JWT, El passo) while others rely on a combination of highly demanding computations and file reads/writes (e.g., Video GIF, Thumbnail, HTML). Having such dissimilarities in our benchmark allows us to better compare the variability in the function's energy efficiency running on our heterogeneous cluster.

1.2 Energy-efficiency measurement and characterisation

To characterise the performance and energy efficiency of our functions running on our two heterogeneous CPUs, we decided to pack all of them as Docker images running in containers, for heterogeneity management simplicity and because PowerAPI is automatically capable of detecting active containers and starting their monitoring. All functions share a common structure, beginning with an entry point written in Python, which is responsible for launching the different function processes. The entry point takes as arguments the duration for which the function needs to run (t) and its allocated number of cores (x). It then spawns x processes, each performing as many executions as possible during t seconds. At the end of execution, the processes report to the entry point how many executions they managed to run. To avoid noise between the different processes, we decided to implement all the functions single-threaded. For functions using libraries taking advantage of multi-core systems, such as Pytorch, we deactivated multi-threading. Some functions require specific user input, such as the zip, image recognition or thumbnail functions. These inputs are static files included in the container.

As each function is assigned a static number of available cores, we need to find a way to restrict the functions to not exceed their quota. Fortunately, Docker enables the possibility to put in place such constraints. Two specific parameters of the Docker `run` command are of interest, respectively `--cpus` and `--cpuset-cpus`. According to Docker's documentation [6], the two parameters are defined as follows:

- `--cpus`: a fractional number that represents the number of cores that the docker container can use. For example, if there is one core, a value of 0.5 will tell Docker to use a maximum of 50% of the unique core. The default value is 0.000, which means no limitation.
- `--cpuset-cpus`: represents the set of cores on which a container is allowed to run. For example, specifying a value of `0-2` will restrict Docker to only use the three first cores (0, 1 and 2) for that container. There is no restriction by default.

In order to make the best placement decisions, our scheduler needs to be fed with past profiling data. For each of our 22 serverless functions described in Table 3.1, we need to collect multiple metrics:

- **Performance** metrics: the number of requests a given function can serve running on one CPU core for one second, for each machine.
- **Energy consumption** metrics: how much energy (in Wh) a given function consumes by serving one request, for each machine.

We define as e_{tot} the total number of executions performed in t seconds running on x CPU cores. We can then compute the number of executions that a single function can execute per CPU core per second (e_{core}) as follows:

$$e_{core} = \frac{e_{tot}}{t \cdot x} \tag{3.1}$$

By taking the mean power usage in watts (p) of the function and the total duration time (t) in seconds, we can compute the average number of Wh (w_{exec}) consumed per execution with the following formula:

$$w_{exec} = \frac{p \cdot (\frac{t}{3600})}{e_{tot}} \tag{3.2}$$

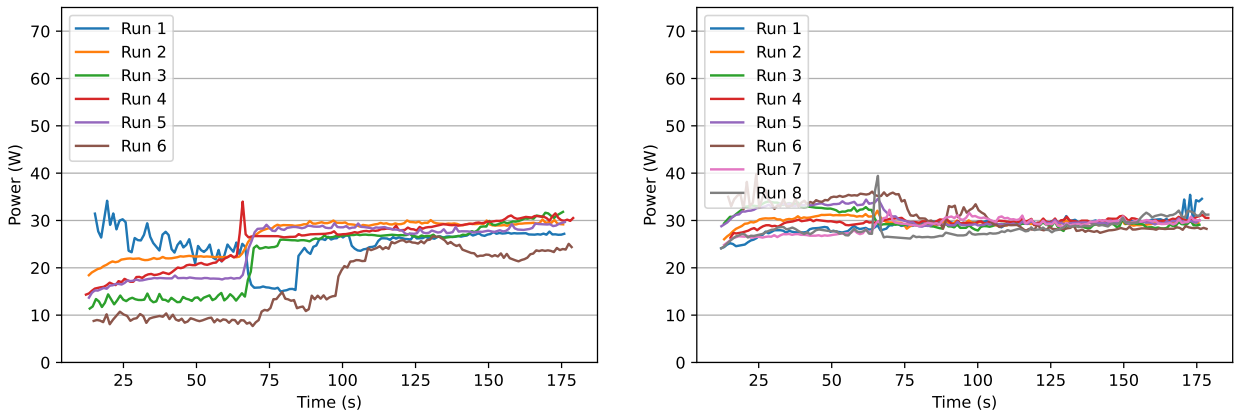
Our first idea to collect these metrics was to run one function at a time on both machines, with all cores actives, together with PowerAPI to collect their respective energy consumption. We quickly gave up on this methodology for the reason that it does not capture the possible resource contention that several functions of different natures can cause by running simultaneously. Multiple functions will not have the same throughput per second if another function, that competes for resources, runs at the same time. To cope with that variability, we tried to have greater and more representative samples of metrics by running multiple experiments. Each experiment is running 8 different functions simultaneously out of our bag of 22. Each function has an eighth of the available cores (threads for Intel) and is assigned to a static set

of cores with the Docker `cpuset-cpus` parameter. As the configuration sums to an assignment of 100% of the machine resources, we decided to limit the CPU usage to 85% with the Docker `cpus` parameter in order to leave some resource for PowerAPI to accurately measure and estimate the power consumption of the containers.

1.3 Preliminary insights

Several interesting characteristics can be extracted from our metrics dataset. First, we can experimentally demonstrate why we decided to give up on our first methodology of benchmarking one function at a time. As we decided to run 20 batches of 8 different functions, all functions have been scheduled multiple times, every time with different neighbouring functions. Table 3.2 shows the mean number of executions each function was able to perform, together with the standard deviation in percentage. It is worth noting that the only variable between executions is the other functions running simultaneously. The running time and number of cores assigned remain unchanged.

We can observe that the number of executions performed per function highly varies, up to 37 %, especially on the Ampere CPU. By running 20 different batches of multiple functions, we try to estimate as accurately as possible the average number of executions per core and the power usage under different potential scheduling decisions. Together with the variability in terms of number of executions, one could wonder if the power usage varies as much as the throughput. Specifically, we try to figure out if the difference in the variability can also be observed in the power profiles of functions. Figure 3.3 shows the PowerAPI power estimations over time of the Python HTML and Node.js JWT functions on the Ampere machine. They respectively have standard deviations of 37.3 % and 2.3 % regarding their average number of executions.



(a) Python HTML function (37.3% STD)

(b) Node.js JWT function (2.3% STD)

Figure 3.3: **PowerAPI process-level power consumption estimations.** Power usage over time estimated by PowerAPI of two different functions running on the Ampere Altra Max CPU. Each run corresponds to a different batch of multiple functions, selected at random, running simultaneously with the targeted function.

Function	Intel		Ampere	
	Executions	**STD (%)**	**Executions**	**STD (%)**
Node.js thumbnail	128063.0	8.9	141983.1	20.5
Node.js video to GIF	1109.5	4.0	2085.0	15.2
Node.js JWT	1143216.8	5.4	3490203.8	2.3
Node.js zip	86385.6	1.8	137058.3	17.4
Node.js HTML	9757928.6	15.1	9089083.4	14.7
Node.js matrix	4903028.8	3.4	10950290.2	4.0
WebAssembly El Passo	1616.2	9.7	1929.4	11.8
C++ El Passo	23095.9	13.1	19078.1	18.2
Java zip	133670.4	9.4	384004.0	2.8
Java JWT	106939231.1	7.9	192244617.7	9.3
Java thumbnail	18957.3	8.7	56657.3	5.1
Java matrix	15817160.4	13.7	40142549.4	2.1
Java HTML	2101205.1	7.0	4082272.8	8.8
Python3 thumbnail	186732.9	9.8	360016.7	21.6
Python3 matrix pure	36601.6	2.0	133862.8	0.3
Python3 matrix NumPy	32261426.1	11.4	43536342.9	34.8
Python3 JWT	34848480.9	6.3	107897601.5	1.0
Python3 zip	159627.6	10.3	249864.0	22.8
Python3 image recognition	12577.8	9.3	6268.0	17.2
Python3 PageRank	24235.0	7.4	38527.8	28.5
Python3 HTML	686861.0	4.7	1019133.7	37.3
Python3 video to GIF	1047.6	2.8	2054.4	11.7

Table 3.2: **Comparison of all functions' total number of executions observed.** Average and standard deviation in percentage of the number of executions of our 22 serverless functions taken on 20 runs with 8 randomly chosen simultaneously running functions.

As we could expect, the Node.js JWT function which has a low deviation in terms of number of executions, exposes a power usage over time that does not vary much. In contrast, the Python HTML function shows more irregularities but tends to stabilise over time. We observe here that the variability in the number of executions approximately transposes to the variability in power usage. However, this is not always the case. Figure 3.4 shows a counter-example with the Python matrix pure function running on Intel. That function has little standard deviation regarding its number of executions (2%), while showing large differences in terms of

power consumption estimates. This is a first important result that demonstrates the difficulty of correctly estimating the energy consumption at the process level due to the high dimensionality of the various factors involved.

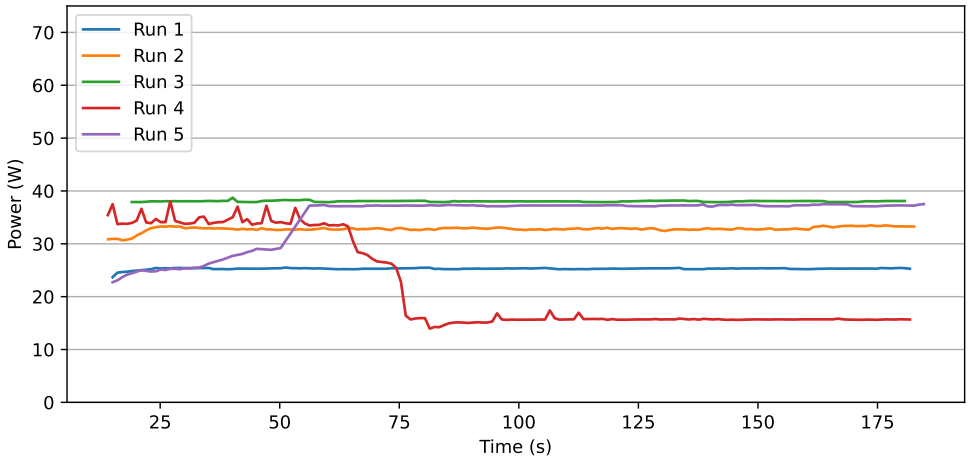


Figure 3.4: **PowerAPI process-level power consumption estimations.** Power usage over time estimated by PowerAPI of the Python matrix pure function running on the Intel Xeon CPU. Each run corresponds to a different batch of multiple functions, selected at random, running simultaneously with the targeted function.

Our hypothesis concerning that behaviour questions the PowerAPI estimations. We do not question PowerAPI from an accuracy point of view but more about the intrinsic way of how it estimates the energy consumption of the different processes and containers. We need to find a reason why a given function that performs approximately the same number of executions from one run to the other, has a divergence of up to 25 W in the estimated power usage.

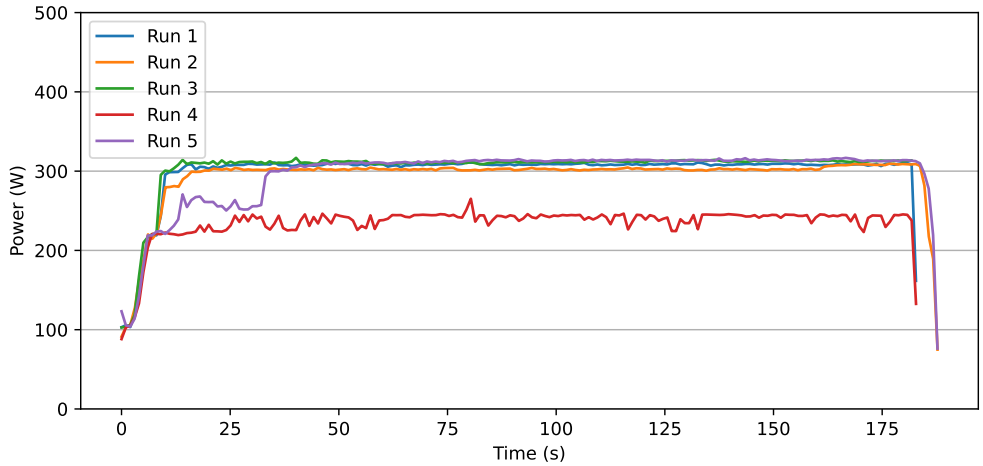


Figure 3.5: **RAPL CPU-level power consumption.** Power usage over time at the CPU level obtained with the RAPL interface of each batch containing the Python matrix pure function running on the Intel Xeon CPU. Each run corresponds to a different batch of multiple functions, selected at random, running simultaneously with the targeted function.

PowerAPI relies on a ground truth power measurement at the CPU level to estimate the energy consumed by each monitored process, according to individual performance events collected per process. A probable reason for PowerAPI to make such volatile estimations is that the other functions running could be increasing the CPU-level power, with a non-proportional impact on their relative performance counters, resulting in a different energy distribution across functions. Figure 3.5, representing the RAPL power values during the same 5 experiments, shows that our reasoning might be correct, or at least part of the explanation. We can observe that RAPL shows less CPU power usage during the fourth run, which impacts the power estimations of the Python matrix pure function. However, Figure 3.4 also shows that there is a difference between runs 1, 2 and 3 which is not significantly present in Figure 3.5.

Figure 3.6 presents our final dataset in terms of per-function energy efficiency used by our Energy-aware scheduler. By considering the mean, 21 out of our 22 functions achieve a higher number of executions per Wh on the Ampere CPU. However, as shown earlier, there can be high variations. For example, the Node.js HTML function is on average better on the Ampere CPU but could very well be more energy-efficient on the Intel CPU in some configurations. Even if the vast majority of our functions perform better energetically on Ampere, the functions do not expose the same level of affinity towards the Ampere CPU. With the assumption that a single machine is not sufficient to host all the demand, these variations of energy efficiency can still be exploited to choose the right function(s) to offload on the Intel CPU.

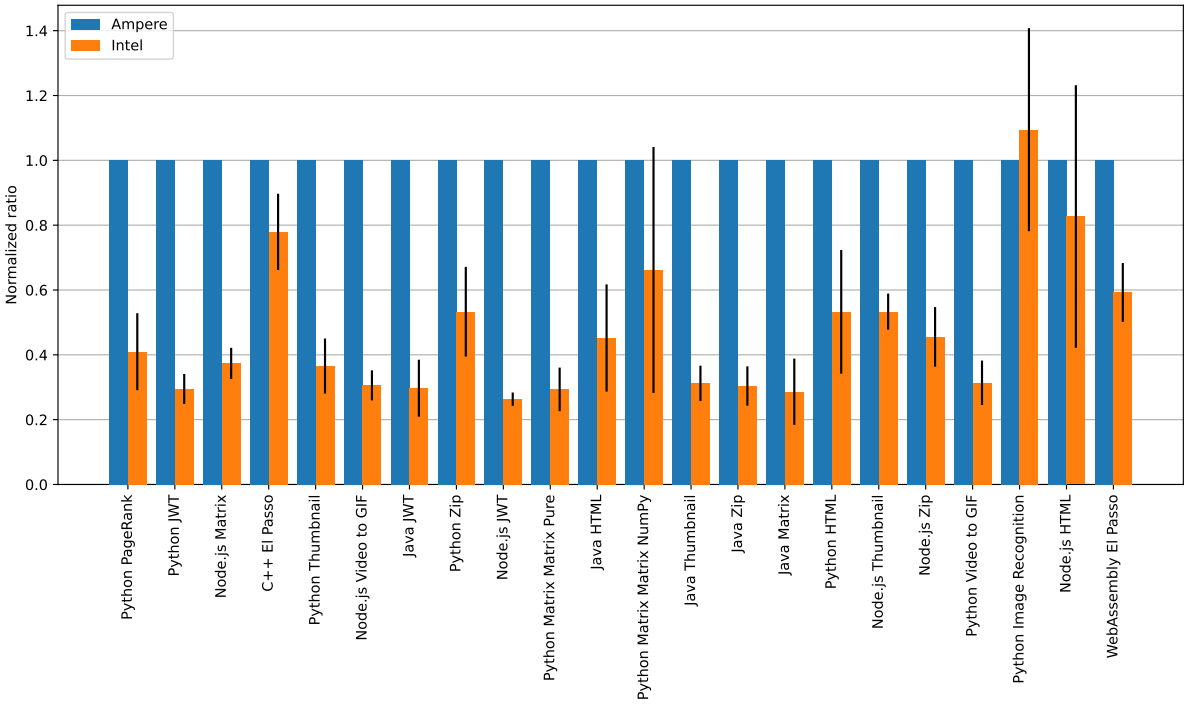


Figure 3.6: **Comparison of the energy efficiency of our 22 functions on our Intel Xeon and Ampere Altra Max CPUs.** On average, 21 out of our 22 functions perform more executions per Wh on the Ampere CPU.

2 Experimental setup

In this section, we present the comprehensive experimental setup we made to evaluate the effectiveness of our Energy-aware and Cores-aware schedulers. We detail the benchmark workloads employed, the various schedulers to which we compare our proposed schedulers, and the metrics considered for performance evaluation.

2.1 Categories of functions

To capture the diversity of serverless functions, we decided to design several use cases that we call categories. All categories are composed of all the 22 functions but the requirements differ for each category. Thus, the portion of CPU cores taken by each function varies depending on the category. We designed our inputs so that for each category, except the general one, the targeted functions take approximately 70% of the total load. Based on our dataset, we could estimate the number of cores required to run a given amount of executions per second for a given function. Table 3.3 defines the categories.

Category	**Description**	**Targeted functions**
General	General purpose category that covers all functions with equal load demand.	All
Cryptography	Related to cryptographic computations, such as token authentication and anonymous credentials.	JWT, El Passo
Media	Covers multi-media tasks transforming and analysing images and videos.	Thumbnail, Image Recognition, Video to GIF
Scientific	Represents science-related workloads, doing heavy computation on matrices and graphs.	Matrix (all versions), Page-Rank
Web	Relates to common Web sites operations such as HTML templates filling and file compression.	HTML, Zip

Table 3.3: **Categories descriptions.** Each category runs our 22 serverless functions, but with different requirements. The targeted functions together take 70% of the total machines' load, except for the general category where the load is evenly distributed to all functions. We consider all supported programming languages for each function. For example for the cryptography category, we run the JWT function in Python, Node.js and Java.

2.2 Reference schedulers for evaluation

Our Energy-aware and Cores-aware schedulers are built upon two constraint programming models. Beyond our proposed schedulers, we developed several additional schedulers to provide a basis for comparison. Specifically, we created 3 of these that mimic common cloud scheduling strategies. The following list names and describes how these schedulers attribute the set of functions to either the Intel or Ampere machine.

- **Ampere-First**: This scheduler simulates an approach of basic consolidation that starts with the Ampere machine. It fills up the Ampere machine first until there are no more CPU cores available, in which case it places the remaining functions on the next machine, the Intel machine in this case.
- **Intel-First**: This scheduler also simulates an approach of a basic consolidation but this one starts with the Intel machine. It fills up the Intel machine first until there are no more CPU cores available, in which case it places the remaining functions on the next machine, the Ampere machine in this case.
- **Round-Robin**: This scheduler represents another basic way of attributing tasks to a cluster of machines. It alternates the placement of functions across the two machines.

Out of curiosity, we made yet another scheduler that behaves in the opposite manner as our Energy-aware scheduler. Indeed, the model is the same as Energy-aware except that the objective function is a maximisation instead of a minimisation. Thus, maximising the energy consumption with respect to our input dataset. It represents a kind of worst-case scenario, we named it **Energy-unaware**. This additional scheduler does not serve as a point of comparison between our implementation and commonly found schedulers in data centers, but rather as a way of evaluating our dataset. Ideally, the Energy-unaware scheduler should be the worst in terms of energy efficiency.

2.3 Function requirements considerations

When a given function is attributed to a machine, the number of CPU cores assigned depends on the machine and the required amount of executions per second. Consequently, schedulers will consider a machine as being full if there is not enough remaining CPU cores to fulfil the requirements. For example, if we consider that a function needs 3 CPU cores to reach its number of executions per second requirement on the Intel machine, based on our dataset values, but only 2 cores are not yet allocated, the scheduler will consider that the machine is full and will assign the function to the Ampere machine. In Chapter 2, we defined the number of CPU cores assigned to every function j by:

$$A_j = \left\lceil \sum_{i=1}^{m} \left(D_{ij} \cdot \frac{R_j}{E_{ij}} \right) \right\rceil \quad \forall j \in [1, n] \tag{3.3}$$

where there are m machines, n functions, D_{ij} is always 0 except for the machine on which the function is allocated, R_j is the requirement of function j in terms of

executions per second and E_{ij} is the average amount of executions per second of the function j made by a single core on machine i. For instance, if a function requires to perform 240 executions per second and that we previously measured that a CPU core does 100 execs per second on a given machine, we need 2.4 cores and we allocate this number rounded to the upper integer value (3). From such calculation, it seems reasonable to think that the function will reach a higher throughput than required as it has been allocated more cores than needed. To accurately compare the energy consumption of the schedulers, we need the computational loads to be as close as possible across configurations. For that purpose, we decided to throttle the CPUs. If we take the previous example where there are 3 cores assigned to a function that only needs 2.4 cores, we will use the Docker `--cpus` option to limit the usage of the 3 cores at 80%. The percentage is computed as the number of cores required divided by the amount of cores allocated.

We specifically crafted the input sets for each category to ensure that there are always functions dispatched on both machines. Otherwise, the most efficient allocation is to place all functions on the Ampere machine and turn off the Intel one. Moreover, we aimed at maximising the load on both machines while not exceeding their capacities. The functions' requirements vary depending on the scheduler or category. Indeed, the number of CPU cores needed could be different whether the function is allocated to one machine or another. Consequently, an assignment of functions to machines made by the Cores-aware scheduler may fit on the two machines while a less efficient assignment made by a Round-Robin scheduler may exceed the number of cores of the machines, even though it is based on the same set of functions and requirements. In such cases, we had to progressively decrease the requirements until no more scheduler assignments exceeded the capability constraints of the machines.

2.4 Evaluation metrics

Once we had our categories, schedulers and functions to benchmark, we had to choose what metrics and values to observe. Firstly, we retrieve the average power in watts to compare which scheduler is more energy-efficient on average. We look at both the PDU values and the metrics exported by RAPL on the Intel machine and SMpro on the Ampere one. It is calculated following this formula, where $\overline{P}$ is the average power, T is the set of timestamps in seconds at which each power usage has been recorded and P_i is the power measured in watts at timestamp i:

$$\overline{P} = \frac{\sum_{i=1}^{|T|} P_i}{|T|} \tag{3.4}$$

Secondly, we compute an approximation of the total energy consumed in watt-hours throughout the experiments. The formula to compute it is as follows:

$$E = \frac{\sum_{i=2}^{|T|} P_{i-1} \cdot (T_i - T_{i-1})}{3600} \tag{3.5}$$

Such calculation of the total energy consumed assumes that the power is stable for the duration between one timestamp to the next. The result is not the exact energy

consumed during the experiment but it is a good approximation if timestamps are sufficiently close to each other. Thirdly, we observe the amount of cores active on each machine from the schedulers' assignments. Lastly, we evaluate the percentage of commonness between the assignment made by the Energy-aware scheduler and the other schedulers, defined as H_i for scheduler i. By reusing notations from Section 2 of Chapter 2, the set S_i of common assignments between a scheduler i and the Energy-aware scheduler is defined as:

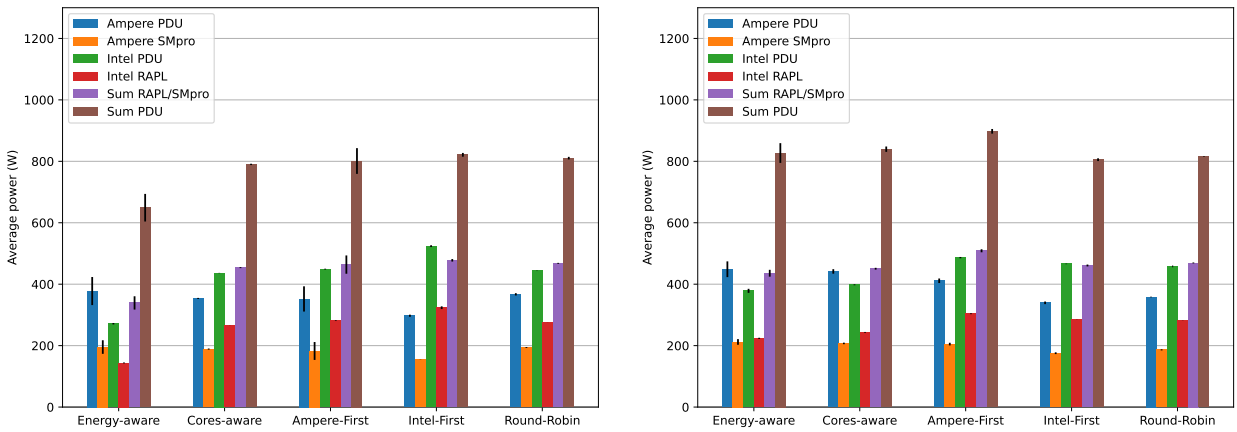
$$S_i := \{A^i_f \in A^i \mid D^i_{jf} = D^e_{jf}\} \tag{3.6}$$

with A^i the set of CPU cores assignments for scheduler i and D^i and D^e the sets of placement decisions of schedulers i and Energy-aware respectively. Our last metric is formulated as:

$$H_i = \frac{\sum_{j \in S_i} j}{\sum_{f \in F} A^e_f} \tag{3.7}$$

3 Results

This section presents the final evaluation of our Energy-aware function scheduler compared to other classic data center schedulers. We show that our scheduler focused on energy efficiency allows us to reduce the total energy consumed by up to 19% compared to common cloud scheduling strategies.



(a) Cryptography category

(b) Scientific category

Figure 3.7: **Average power usage of five different scheduling strategies running two specific categories.** The purple and brown bars respectively measure the total power of both machines at the CPU (RAPL/SMpro) and machine (PDU) levels.

Our first experiment consists of running every category with all our schedulers. We run the functions for one minute on their respective machine selected by each scheduler. Figure 3.7 shows our results for five schedulers running two different categories. While the results of our Energy-aware scheduler are promising within the

cryptography category, the performance of our scheduler is less satisfactory in the scientific category.

A first hint trying to explain those results comes from the high variability in the number of executions performed, as demonstrated in Section 1.2. Ideally, if our metrics collected during the dataset creation were stable, we should observe approximately the same total throughput over our five schedulers. However, this is not the case. Figure 3.8 shows that there is a difference of up to 23% of the total number of executions performed according to the scheduling placement decision and to the target requirement. Having such dissimilarities makes the power usage comparisons between our schedulers distorted.

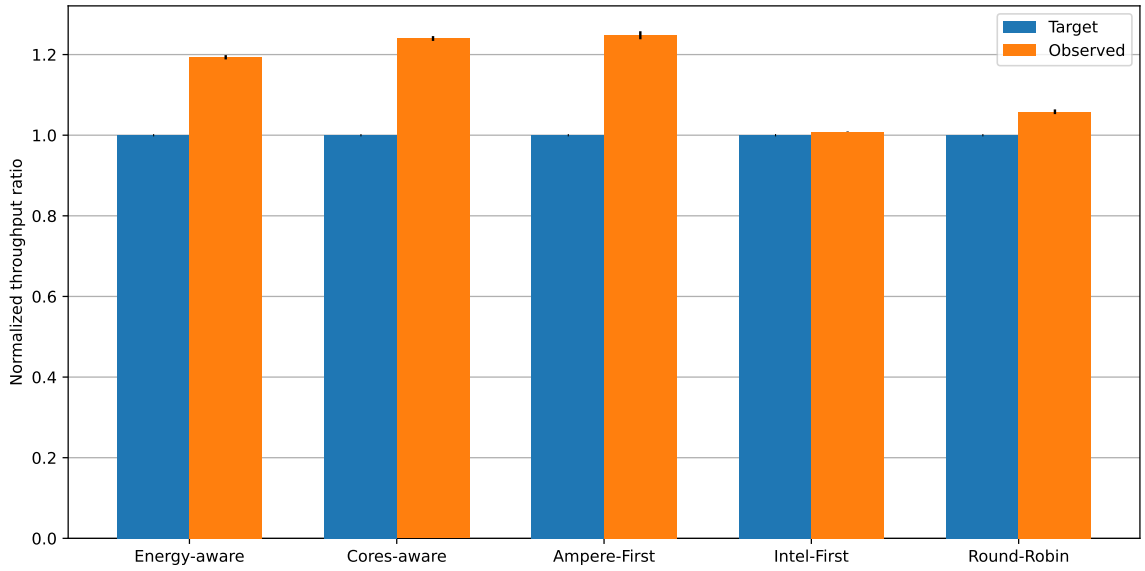


Figure 3.8: **Throughput comparison.** Normalised total throughput comparison for the scientific category. It compares the functions' requirements to the observed throughput.

To have a stable throughput across configurations leading to comparable results, we tried two different methodologies, both limiting the number of executions of the functions to exactly match their throughput requirement. The difference in the two methods resides in the stopping criteria for the functions.

The first method tries to run, for each function, the exact number of desired executions. If the allocated minute to run all their executions is over, then the function stops. If the function finishes its executions early, for example after 50 seconds, it stops but we still collect energy measurement for exactly one minute for each scheduler/category configuration. This methodology has its advantages and drawbacks. As we do not wait for every function to complete its requirements,

some differences in the total throughput could still be observed. However, running the functions that way has the advantage of drastically reducing the gap in total throughput between each configuration.

In contrast with the first methodology, our second approach respects all function execution requirements and only stops once every function has finished. In that case, the total throughputs are exactly the same, but it comes at another cost. Sometimes, only one function has been assigned a too low number of cores to reach its desired number of executions in less than one minute. Unfortunately, these functions are likely to mislead our total energy consumption results. When only one or two functions remain, more cores will become idle. This raises a few concerns as the two machines are drawing different idle power consumption.

We decided to use the first method for the evaluation, as the throughput is sufficiently stable and it avoids measuring idle consumption for too long if a function takes more time than planned to finish its executions. A possible future improvement to evaluate even better the energy consumption of functions running on multiple machines would be to maintain a constant throughput throughout the experiment duration for all functions. One way to do it is to implement an auto-balancing execution throttler that should strive to keep the execution rate constant seconds after seconds by continually adapting the waiting time between executions. Such tools already exist, such as wrk2 [31]. However, those tools achieve a constant throughput and correct latency for sending network requests whereas, in our case, we need it for local function execution disconnected from the network.

Figure 3.9 shows multiple metrics comparing the different schedulers running the general category. First, we can observe that the vast majority of our running functions manage to meet their requirements. The total throughput has almost always a ratio of 1, meaning that all expected executions have been performed. This allows us to more accurately assess the energy performance of our various schedulers.

Speaking about energy efficiency, we can observe that the Energy-aware scheduler consumes slightly less energy than the second best, the Cores-aware scheduler. However, all schedulers are relatively close to each other in terms of energy consumption, with a difference between the best and worst of approximately 5%. In this particular case, the good performances of the Cores-aware scheduler can be explained by its assignment that is very close to the one decided by the Energy-aware scheduler. All other schedulers have between 50 and 70% of common assignment with regards to the Energy-aware.

Another noticeable observation is the relation between the number of cores allocated and the total energy consumed. If we look closely at the energy measured on Intel, we can see that between the Energy-aware and the Energy-unaware schedulers, there is a slight increase. However, if we also look at the number of active cores with both schedulers, there is a wider gap. That observation tells us that consolidation might be beneficial for energy efficiency. Another way to see the importance of

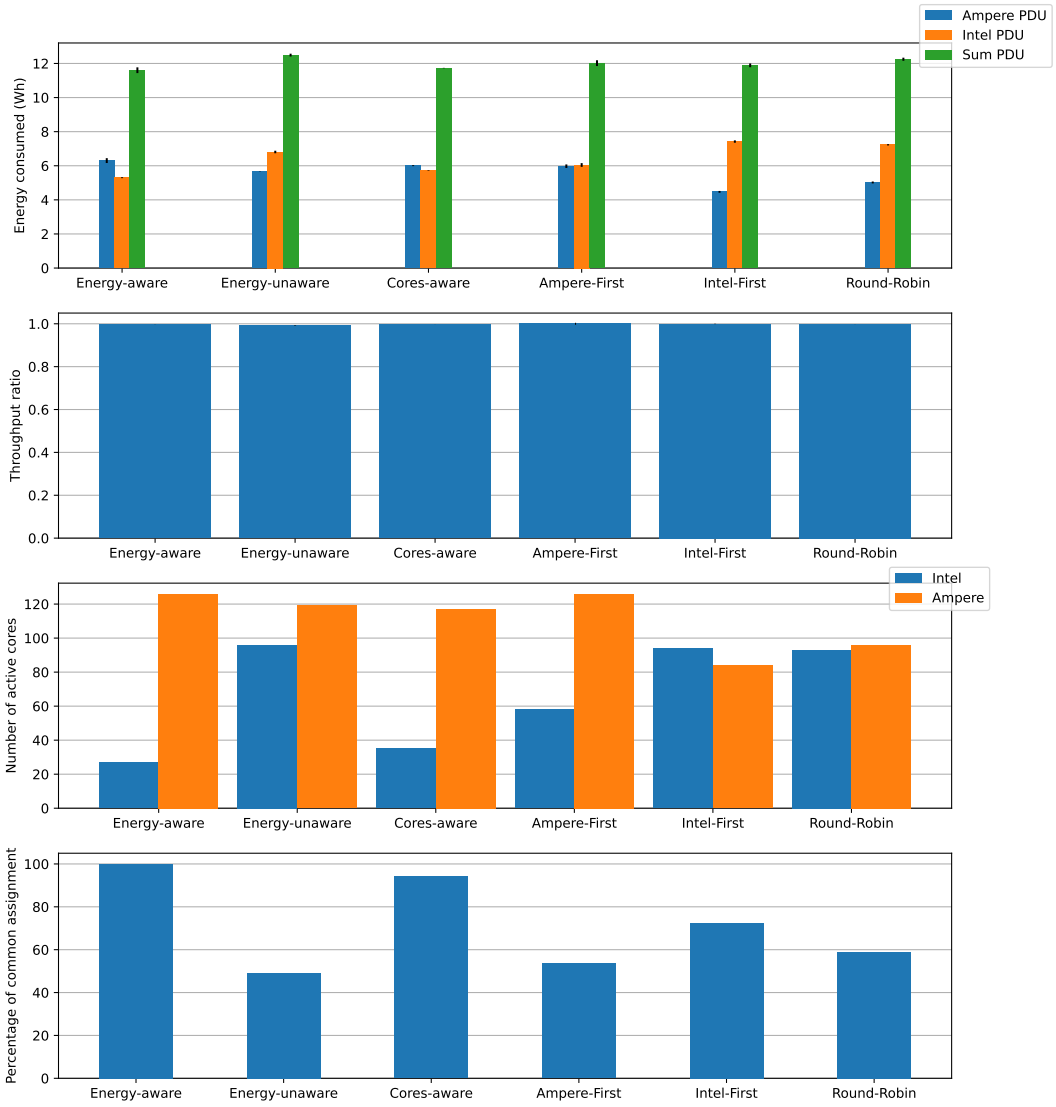


Figure 3.9: **Comparison of different scheduling strategies running the general category.** The first subplot represents the total energy consumed by all functions respecting their requirements. The second one shows the total throughput ratio compared to the original requirements. The third one depicts the number of cores assigned on each machine while the last subplot compares our commonness metric between every assignment.

consolidation is by comparing the Ampere-first and Intel-first schedulers. They expose almost the same total energy consumption, and that was surprising for us as almost every function was tagged as more energy efficient on Ampere during our dataset creation. That behaviour could be explained by two factors. First, as said earlier, it seems that consolidating function on the Intel machine has a positive impact on energy efficiency. But secondly, we must also be careful that the Intel-first configuration has 20% more resemblance with the Energy-aware functions placement decision than the Ampere-first configuration.

Figure 3.10 shows the energy consumption savings of our Energy-aware scheduler compared to the worst and the average of all other schedulers for every category. For this plot, the Energy-unaware scheduler has been withdrawn as it does not represent a realistic use case. We can see that compared to the worst scheduler for each configuration, our Energy-aware spares between 4 and 19.3% of energy. With regards to the average of all other schedulers, our implementation saves between 1.9 and 13.9% of energy.

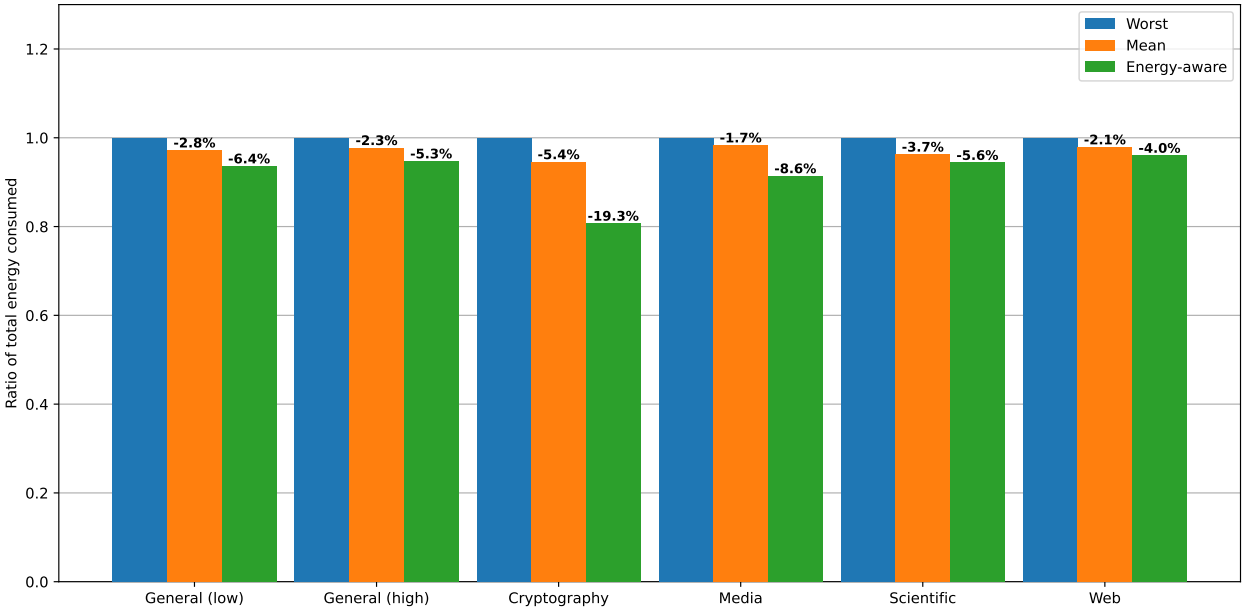


Figure 3.10: **Energy consumption ratio comparison** between the worst performing scheduler, the average of non-energy aware schedulers and our Energy-aware scheduler. The Energy-unaware scheduler is not included.

By measuring each scheduler's energy performance compared to our Energy-aware scheduler, we can observe that it is inversely proportional to its average percentage of commonness with the Energy-aware scheduler. If we compute the Pearson's correlation, both are inversely correlated at 94%. We can think that by evaluating our scheduler on larger heterogeneous clusters with a larger set of functions, the percentage of assignment resemblance between schedulers will decrease, which might lead to even wider gaps in terms of energy consumed.

Chapter 4

Conclusion and future work

This master's thesis helped us deepen our knowledge of cutting-edge and topical subjects. Energy consumption is a central concern nowadays and understanding how to precisely and accurately estimate that usage is of importance, but also a challenging task with many difficulties. Process-level power consumption estimation and energy-efficiency prediction for software running on heterogeneous resources are complex tasks influenced by many, often interdependent, factors. Additionally, our work demonstrates that fine-grain power usage estimates allow more efficient energy resource management.

Our main contributions are the following. We developed a software support for monitoring energy consumption at the process level on the novel Ampere Altra CPUs based on the PowerAPI toolchain. Using our 22 custom serverless functions, we designed, developed, and evaluated two constraint programming-oriented schedulers deciding on a static placement of functions. A Cores-aware scheduler that minimises the total number of CPU cores in use and an Energy-aware scheduler that minimises the total energy consumed, based on real-world apriori energy measurements. We showed that our Energy-aware scheduler outperforms standard cloud schedulers and the Cores-aware scheduler in terms of Wh consumed by up to 19%.

During our work, we discovered many variability points that could impact the performance of our Energy-aware scheduler. We believe that several of them can be addressed as future work to improve energy efficiency even further.

A first point for future work involves improving the quality of the dataset used by the scheduler. Our current approach of building our dataset might not be optimal, based on our variability observations. Rather than flattening the variability by taking multiple batches of different simultaneous functions, a more effective approach would involve characterising reliably the resource contention and dependencies between individual functions. As we showed in Section 1.3 of Chapter 3, a given function might achieve better energy efficiency on a different machine based on neighbouring functions. However, this methodology raises some difficulties, especially in terms of scalability. If we have hundreds of functions, cross-testing every function becomes exponentially difficult if not unfeasible. In addition, each function might have a

different energetically preferred CPU architecture or generation according to its requirements, which adds another dimension to the problem.

Another potential area for improvement lies in server consolidation. Currently, our scheduler is agnostic of machine load, with the exception of respecting the capacity constraint to not overload the machines. However, in some cases, it would be worth considering consolidation with energy-efficiency scores. For example, in the extreme case where there are already 90 out of 96 cores running on machine A and 20 out of 128 on machine B, placing a new function on machine B (with machine B measured as the most energy-efficient machine for that function) might be worse in terms of energy consumption than consolidating the function on machine A. Indeed, a nearly fully-loaded machine's energy consumption might be only slightly increased by the addition of an additional task, as previously shown by Ching-Hsien Hsu *et al.* [51]. This relates to our first proposed improvement point, where we could also take into account multiple layers of loads to measure the most energy-efficient placement.

A third specific point of interest could be to consider the dynamic case with online scheduling. Online scheduling poses a significant challenge compared to static scheduling and requires very careful implementation, mostly due to a higher number of parameters. Several questions would have to be answered such as: How do we report the current load to the scheduler? What if the optimal assignment is not available at scheduling time? Do we consider service migration? All these questions are not trivial to answer and could be part of future work.

Another important but challenging point of improvement is to incorporate in our scheduling pipeline a way to characterise the CPU affinity of previously unseen functions using only statically extracted information and a restricted set of apriori monitored functions. Two hypotheses must be met for that idea to work. The first is that functions or tasks can be classified into multiple categories. These categories can be programming languages, functionalities or use cases. A difficulty in regrouping services in categories is the limited amount of information that can be extracted before running the software. For example, reading the source code to check for library import is a possibility, but not as easy for compiled code such as C programs. Another possibility could be to classify the different services upon a dynamic monitoring phase. For that purpose, we find the work of Chang H. *et al.* [41] very interesting. They leverage machine learning for microservices fingerprinting and classification based on system call monitoring. Their methodology could also very well be applied to the serverless paradigm. Yet, our idea cannot work if the second hypothesis is inconclusive. The second hypothesis that needs to be met is that the different obtained classes of services have the same energy-efficiency profiles, or at least an energy affinity towards the same hardware. In the case where the two hypotheses would be met, we would be able to create a generic and pre-trained Energy-aware scheduler, capable of distributing unseen services towards the most energy-efficient machine. We could even consider including this kind of Energy-aware scheduler in a container orchestrator such as Kubernetes.

A final point that could be addressed as future work would be to add the support for Ampere SMpro registers readings into the Linux `perf` tool. It is still not integrated, and poses difficulties for a clean integration in PowerAPI. Indeed, the PowerAPI framework for Intel CPUs only relies on `perf` readings for every performance event, including RAPL. Our implementation for Ampere CPUs makes their code base more tortuous, and having perf support would ease the integration of our implementation.

Bibliography

[1] ACPI Collaborative Processor Performance Control (CPPC) documentation. https://docs.kernel.org/admin-guide/acpi/cppc_sysfs.html. Accessed: 18/04/2024.

[2] Ampere Altra device family documentation. https://amperecomputing.com/customer-connect/products/altra-family-device-documentation. Accessed: 19/04/2024.

[3] Ampere Altra Max CPU product web page. https://amperecomputing.com/products/processors. Accessed: 28/05/2024.

[4] Ampere's Altra SMpro hwmon kernel driver documentation. https://docs.kernel.org/hwmon/smpro-hwmon.html. Accessed: 19/04/2024.

[5] Arm Chips Gaining in Data Centers, But Still in Single Digits. https://www.datacenterknowledge.com/arm/arm-chips-gaining-data-centers-still-single-digits. Accessed: 23/05/2024.

[6] Docker online documentation. https://docs.docker.com/. Accessed: 16/05/2024.

[7] Docker Swarm online documentation. https://docs.docker.com/engine/swarm/. Accessed: 23/05/2024.

[8] FFmpeg a complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org. Accessed: 09/05/2024.

[9] FreeMaker Java template engine. https://freemarker.apache.org/index.html. Accessed: 09/05/2024.

[10] Intel performance monitoring events documentation. https://perfmon-events.intel.com/. Accessed: 15/04/2024.

[11] jinja2 Python template engine library online documentation. https://jinja.palletsprojects.com/en/3.1.x/. Accessed: 02/06/2024.

[12] JSON Web Token (JWT) - RFC7519. https://datatracker.ietf.org/doc/html/rfc7519. Accessed: 09/05/2024.

[13] Kind local Kubernetes clusters. https://kind.sigs.k8s.io. Accessed: 02/05/2024.

[14] Kubernetes online documentation. https://kubernetes.io/. Accessed: 23/05/2024.

[15] Linux stress-ng manual page. https://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html. Accessed: 20/04/2024.

[16] Memcached high-performance, distributed memory object caching system. https://memcached.org. Accessed: 30/04/2024.

[17] NumPy Python library website. https://numpy.org/. Accessed: 02/06/2024.

[18] OpenFaas Serverless orchestration framework website. https://www.openfaas.com/. Accessed: 02/06/2024.

[19] OpenWhisk open source distributed Serverless platform. https://openwhisk.apache.org. Accessed: 30/04/2024.

[20] Pillow: Python imaging library online documentation. https://pillow.readthedocs.io/en/stable/. Accessed: 02/06/2024.

[21] PowerAPI HwPC sensor configuration. https://powerapi.org/reference/sensors/hwpc-sensor/. Accessed: 15/04/2024.

[22] PowerAPI Smartwatts formula configuration. https://powerapi.org/reference/formulas/smartwatts/. Accessed: 16/04/2024.

[23] PyJWT Python library online documentation. https://pyjwt.readthedocs.io/en/stable/. Accessed: 02/06/2024.

[24] Redis source-available, in-memory storage system. https://redis.io. Accessed: 30/04/2024.

[25] Scaphandre online documentation. https://hubblo-org.github.io/scaphandre-documentation/. Accessed: 02/06/2024.

[26] sharp Node.js image processing library online documentation. https://sharp.pixelplumbing.com/. Accessed: 02/06/2024.

[27] The Apache Thrift software framework. https://thrift.apache.org. Accessed: 30/04/2024.

[28] The MongoDB Database website. https://www.mongodb.com/. Accessed: 02/06/2024.

[29] The Prometheus monitoring system and time series database website. https://prometheus.io/. Accessed: 02/06/2024.

[30] Torchvision online documentation. https://pytorch.org/vision/stable/index.html. Accessed: 09/05/2024.

[31] wrk2 HTTP benchmarking tool GitHub repository. https://github.com/giltene/wrk2, 2019.

[32] DeathStarBench social network GitHub repository. https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork, 2024. Version: 0.4.0.

[33] Java JWT library GitHub repository. https://github.com/jwtk/jjwt, 2024. Version: 0.12.5.

[34] JsonWebToken Node.js library GitHub repository. https://github.com/auth0/node-jsonwebtoken, 2024. Version: 9.0.2.

[35] JSZip JavaScript library GitHub repository. https://github.com/Stuk/jszip, 2024. Version: 0.33.2.

[36] Mustache Node.js template engine library GitHub repository. https://github.com/mustache/mustache.github.com, 2024. Version: 4.2.0.

[37] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. Kepler: A framework to calculate the energy consumption of containerized applications. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 69–71. IEEE, 2023.

[38] Mohammad Sadegh Aslanpour, Adel N Toosi, Muhammad Aamir Cheema, and Raj Gaire. Energy-aware resource scheduling for serverless edge computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 190–199. IEEE, 2022.

[39] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[40] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News*, 92:43–44, 2013.

[41] Hyunseok Chang, Murali Kodialam, TV Lakshman, and Sarit Mukherjee. Microservice fingerprinting and classification using machine learning. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2019.

[42] Jinchao Chen, Yu He, Ying Zhang, Pengcheng Han, and Chenglie Du. Energy-aware scheduling for dependent tasks in heterogeneous multiprocessor systems. *Journal of Systems Architecture*, 129:102598, 2022.

[43] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[44] G Csardi and T Nepusz. The igraph software. *Complex syst*, 1695:1–9, 2006.

[45] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 189–194, 2010.

[46] Zexi Deng, Zihan Yan, Huimin Huang, and Hong Shen. Energy-aware task scheduling on heterogeneous computing systems with time constraint. *IEEE Access*, 8:23936–23950, 2020.

[47] Spencer Desrochers, Chad Paradis, and Vincent M Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, pages 455–470, 2016.

[48] Guillaume Fieni. HWPC-sensor GitHub repository. `https://github.com/powerapi-ng/hwpc-sensor`, 2024. Version: 1.3.

[49] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. Smartwatts: Self-calibrating software-defined power meter for containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 479–488. IEEE, 2020.

[50] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[51] Ching-Hsien Hsu, Kenn D Slagter, Shih-Chang Chen, and Yeh-Ching Chung. Optimizing energy consumption with task consolidation in clouds. *Information Sciences*, 258:452–462, 2014.

[52] Biao Hu, Zhengcai Cao, and Mengchu Zhou. Energy-minimized scheduling of real-time parallel workflows on heterogeneous distributed computing systems. *IEEE Transactions on Services Computing*, 15(5):2766–2779, 2021.

[53] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[54] Kenli Li, Xiaoyong Tang, and Keqin Li. Energy-efficient stochastic task scheduling on heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):2867–2876, 2013.

[55] Jing Mei, Kenli Li, and Keqin Li. Energy-aware task scheduling in heterogeneous computing environments. *Cluster Computing*, 17:537–550, 2014.

[56] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[57] Sanjaya K Panda and Prasanta K Jana. An energy-efficient task scheduling algorithm for heterogeneous cloud computing systems. *Cluster Computing*, 22(2):509–527, 2019.

[58] David Pointcheval and Olivier Sanders. Short randomizable signatures. In *Topics in Cryptology-CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29-March 4, 2016, Proceedings*, pages 111–126. Springer, 2016.

[59] Gourav Rattihalli, Ninad Hogade, Aditya Dhakal, Eitan Frachtenberg, Rolando Pablo Hong Enriquez, Pedro Bruel, Alok Mishra, and Dejan Milojicic. Fine-grained heterogeneous execution framework with energy aware scheduling. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 35–44. IEEE, 2023.

[60] Isabelly Rocha, Christian Göttel, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. 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. IEEE, 2019.

[61] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1, 2010.

[62] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[63] Zhiyi Zhang, Michał Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. El passo: efficient and lightweight privacy-preserving single sign on. *Proceedings on Privacy Enhancing Technologies*, 2021.

Appendices

1 PowerAPI configurations

This section describes our configuration files for the PowerAPI sensor and formula on our Intel and Ampere CPUs.

1.1 PowerAPI configurations on Intel

Listing 1 shows the full configuration of the PowerAPI HwPC sensor for Intel. The sensor collects hardware counter values every 1000 ms, which are stored in a local MongoDB instance. At the system level, the events collected are the RAPL energy for the whole CPU and the MPERF and APERF counters measuring the cores' frequency. At the process level, four different events are monitored. This configuration is the default configuration recommended by PowerAPI.

```
{
  "name": "sensor",
  "verbose": true,
  "frequency": 1000,
  "output": {
    "type": "mongodb",
    "uri": "mongodb://127.0.0.1",
    "database": "db_sensor",
    "collection": "report"
  },
  "system": {
    "rapl": {
      "events": ["RAPL_ENERGY_PKG"],
      "monitoring_type": "MONITOR_ONE_CPU_PER_SOCKET"
    },
    "msr": {
      "events": ["APERF", "MPERF"],
      "monitoring_type": "MONITOR_ALL_CPU_PER_SOCKET"
    }
  },
  "container": {
    "core": {
      "events": [
        "CPU_CLK_UNHALTED:REF_XCLK",
        "CPU_CLK_UNHALTED:THREAD_P",
        "LLC_MISSES",
        "INSTRUCTIONS_RETIRED"
      ]
    }
  }
}
```

Listing 1: PowerAPI HwPC sensor configuration on Intel

Listing 2 shows the configuration of the Smartwatts formula on Intel. The formula is configured in stream mode, estimating the energy consumption online. It takes its input from the local MongoDB instance filled by the sensor. The process-level energy consumption estimations are exported to a local InfluxDB instance. Finally,

some information about the CPU characteristics are given. We disabled the DRAM power consumption estimations.

```
{
  "verbose": true,
  "stream": true,
  "input": {
    "puller": {
      "model": "HWPCReport",
      "type": "mongodb",
      "uri": "mongodb://127.0.0.1",
      "db": "db_sensor",
      "collection": "report"
    }
  },
  "output": {
    "pusher_power": {
      "tags": "socket",
      "model": "PowerReport",
      "type": "influxdb",
      "uri": "127.0.0.1",
      "port": 8086,
      "db": "db_result"
    }
  },
  "cpu-base-freq": 2100,
  "cpu-tdp": 165,
  "cpu-error-threshold": 2.0,
  "sensor-reports-frequency": 1000,
  "disable-dram-formula": true
}
```

Listing 2: PowerAPI Smartwatts formula configuration on Intel

1.2 PowerAPI configurations on Ampere

Listing 3 shows the full configuration of our implementation of the PowerAPI HwPC sensor for Ampere. The sensor collects hardware counter values every 1000 ms, which are stored in a local MongoDB instance. At the system level, the events collected are the CPU and IO energy consumption from the SMpro driver and the CPPC driver's counters measuring the cores' frequency. At the process level, three different events are monitored.

```
{
  "name": "sensor",
  "verbose": true,
  "frequency": 1000,
  "output": {
    "type": "mongodb",
    "uri": "mongodb://127.0.0.1",
    "database": "db_sensor",
    "collection": "report"
  },
  "system": {
    "rapl": {
      "events": ["XGENE_ENERGY_CPU", "XGENE_ENERGY_IO"],
      "monitoring_type": "MONITOR_ONE_CPU_PER_SOCKET"
    },
    "msr": {
      "events": ["CPPC"]
    }
  },
  "container": {
    "core": {
      "events": [
        "PERF_COUNT_HW_CPU_CYCLES",
        "PERF_COUNT_HW_INSTRUCTIONS",
        "PERF_COUNT_HW_STALLED_CYCLES_FRONTEND"
      ]
    }
  }
}
```

Listing 3: PowerAPI HwPC sensor configuration on Ampere

Listing 4 shows the configuration of our implementation of the Smartwatts formula on Ampere. The formula is configured in stream mode, estimating the energy consumption online. It takes its input from the local MongoDB instance filled by the sensor. The process-level energy consumption estimations are exported to a local InfluxDB instance. Finally, some information about the CPU characteristics are given. We disabled the DRAM power consumption estimations.

```
{
  "verbose": true,
  "stream": true,
  "input": {
    "puller": {
      "model": "HWPCReport",
      "type": "mongodb",
      "uri": "mongodb://127.0.0.1",
      "db": "db_sensor",
      "collection": "report"
    }
  },
  "output": {
    "pusher_power": {
      "tags": "socket",
      "model": "PowerReport",
      "type": "influxdb",
      "uri": "127.0.0.1",
      "port": 8086,
      "db": "db_result"
    }
  },
  "cpu-base-freq": 1000,
  "cpu-tdp": 220,
  "cpu-error-threshold": 2.0,
  "sensor-reports-frequency": 1000,
  "disable-dram-formula": true
}
```

Listing 4: PowerAPI Smartwatts formula configuration on Ampere

2 An initial attempt at measuring the power consumption of individual microservices

This appendix presents our first attempt, before focusing our attention on the Function as a Service (FaaS) or serverless cloud paradigm, of characterising the energy consumption of microservices to build an efficient scheduler minimising the total energy consumed by a distributed application. Before having any fine-grain tools for the Ampere Altra Max CPU, we created a coarse-grain experimental setup to verify the feasibility of such scheduler. Our coarse-grain methodology involves monitoring separately the consumption of each single service composing an application, on each server, so that we could have an insight into which machine consumes less energy for each service. For this purpose, we leveraged the open-source DeathStarBench [50] microservices applications benchmark suite.

2.1 Microservice benchmarks

Multiple frameworks available online enable developers to run benchmarks on microservices applications. The main reference in terms of microservice applications benchmarking, and the one we selected, is the open-source benchmark suite named DeathStarBench [50], developed by the SAIL group at Cornell University. It includes applications composed of dozens of microservices that simulate large, modular and extensible end-to-end services, such as a social network, a media service, a hotel reservation site, an e-commerce site, a banking system, and IoT applications. Yet, only the 3 first applications are released. Currently, the DeathStarBench's applications do not support ARM architecture, they only work on x86 CPUs. Consequently, we cross-compiled their source code to have dual compatibility whenever it was possible.

Each application is defined by a docker-compose file and comes with scripts to populate the databases and the workloads to stress the system. To be more precise, it uses wrk2 [31] for the workloads, an HTTP benchmarking tool written in C able to send an important load towards an API endpoint to test its performance. Its particularity is that it sends the load at a constant request rate to simulate multiple users hitting the URL concurrently.

2.2 Coarse-grain measurements

The objective of our methodology is to verify experimentally, without using fine-grain tools, the feasibility of an efficient energy-aware scheduler for microservice applications based on apriori energy measurements. We rely on multiple Power Distribution Units (PDUs) connected to our heterogeneous machines for the energy measurement part. Since these measures are coarse-grain, at the machine level, we cannot easily characterise the power usage of individual microservices. Consequently, we designed an experiment that isolates a single microservice to monitor its energy consumption, while all other microservices of the application are hosted on another machine.

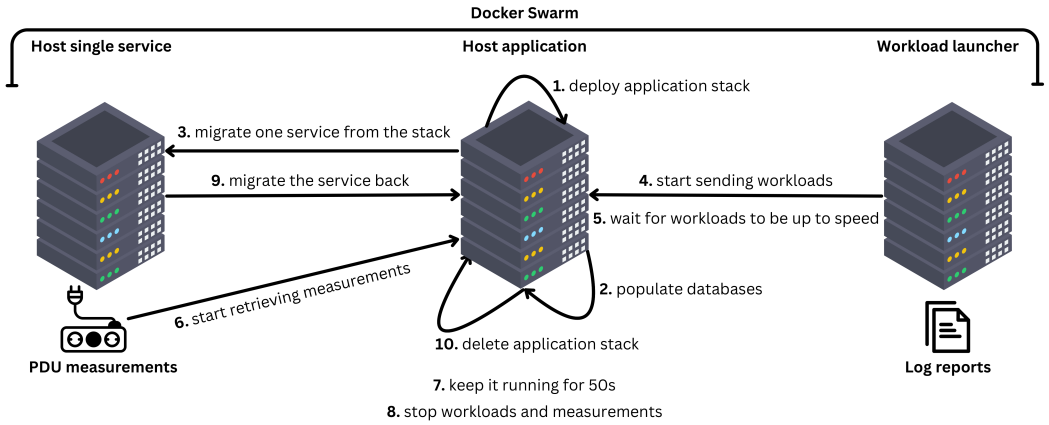


Figure 1: using a Docker Swarm of 3 machines.

Figure 2: **Illustration of a PDU microservice measurement iteration.** It represents a Docker Swarm of 3 machines and shows the steps together with their order. The machine acting as the host for the single service has either the Intel Xeon CPU or the Ampere Altra Max, depending on which machine we want to measure the microservice energy consumption.

Our experiment involves three machines, one for hosting the single service from the application that will be measured, another one for hosting all the other services from the stack and a last one for sending the workloads toward the application. The three machines are part of a common Docker Swarm cluster. The PDU measurements are collected from the machine hosting the single service being measured and the machine sending workloads generates log reports on the requests being sent to the application. To begin, one of the machines creates the cluster and the two others join it. Once the cluster is created, we repeatedly apply the methodology shown in Figure 2, each time with a different microservice from the stack. The DeathStarBench application is deployed on the machine that hosts the application. Then, the application's databases are populated using the scripts given in the benchmark suite. Once up and running, one microservice is migrated to the measurement machine and the third machine launches the workloads, also included in DeathStarBench. During this step, we have to wait for the workloads to get up to speed as it takes some time for wrk2 to reach a constant request rate. Once stable, we retrieve power measurements from the PDU for 50 seconds before stopping the measurements and workloads. Then, the single microservice is migrated back to the machine hosting the application and the whole application stack is deleted. After repeating this procedure for all the services present in the application and for both types of machines, we obtain a dataset with the energy consumption under load of each microservice. For our experiments, we decided to use the social network application from DeathStarBench. The microservices composing that application are illustrated by Figure 3.

Figures 4 and 5 show the energy consumption of each microservice respectively running on the Ampere and Intel CPU. We decided to remove the idle consumption from the raw consumption measured by the PDU to have a more visual comparison of the Intel and Ampere measurements. Otherwise, it would have been harder to interpret the results because our machine having the Intel CPU has a higher idle

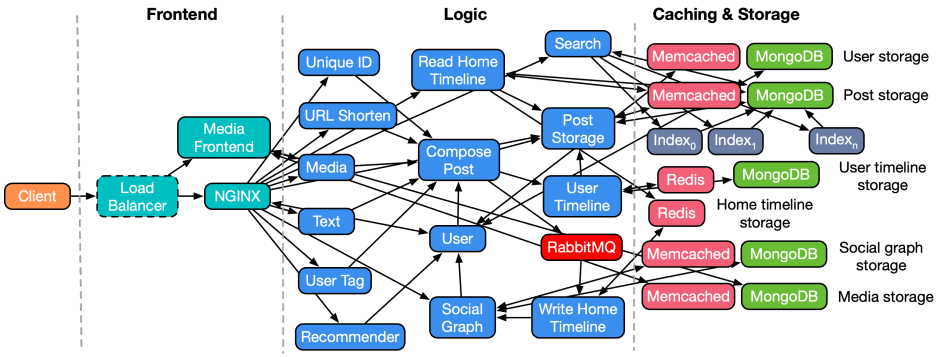


Figure 3: **DeathStarBench social network application.** All microservices are represented, together with how they interact with each other. Illustration taken from the DeathStarBench GitHub repository [32].

power consumption than the one having the Ampere CPU. However, the impact of such a manipulation is difficult to evaluate as the energy consumption that is external to a process is not necessarily the consumption measured before the process started. Indeed, the idle power consumption of a CPU is probably lower when a process is doing heavy work than when the CPU has no task to execute because the duration during which the CPU is idling is decreased. Some services that have experienced near-idle consumption sometimes appear in negatives on those figures. It is due to the removal of the average idle consumption from measurements on which the idle power usage varies slightly.

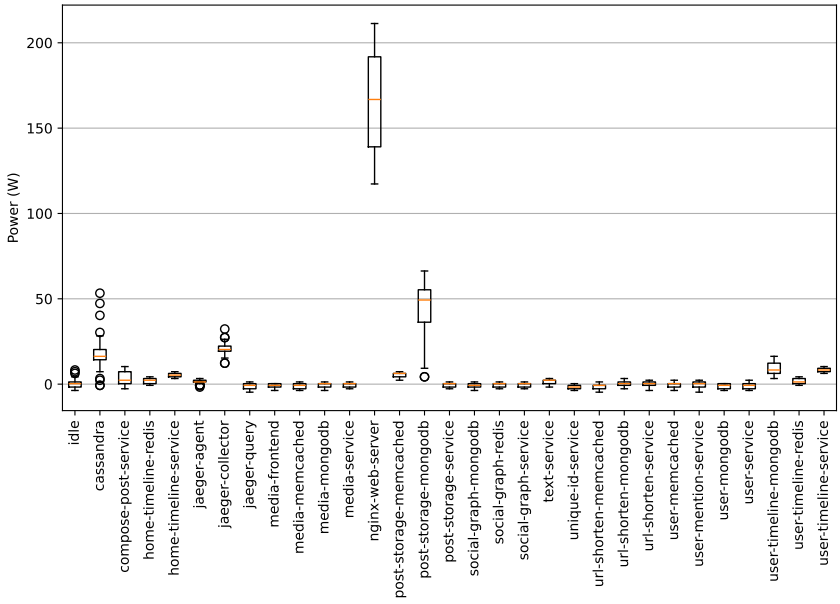


Figure 4: **Microservices' energy consumption from the social network application on Ampere.** PDU power measurements of the social network application from DeathStarBench with the mean idle consumption removed on the machine with an Ampere Altra Max CPU.

A noticeable behaviour in Figures 4 and 5 is that only a few microservices of the application are consuming energy even though the whole application is under stress. This is due to the interdependence of the microservices. Indeed, some services have to do heavier work than others for each request, making it difficult to measure the energy consumption of the under-solicited ones as we need them to be under load to characterise their power usage.

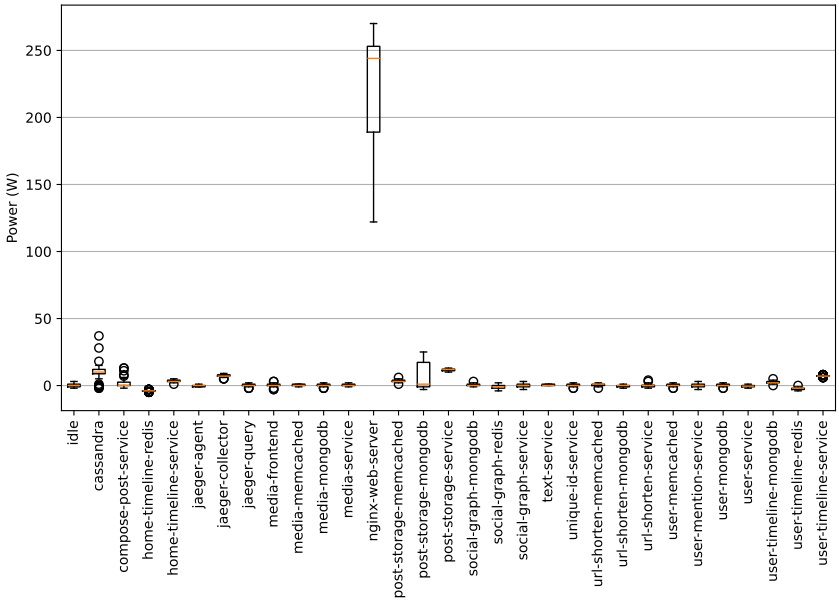


Figure 5: **Microservices' energy consumption from the social network application on Intel.** PDU power measurements of the social network application from DeathStarBench with the mean idle consumption removed on the machine with an Intel Xeon Gold 5318Y CPU.

2.3 Microservice limitations

Although measuring the difference in energy consumption of microservices in an application stack seemed relevant to be able to schedule the microservices toward the most efficient machine raises multiple issues. Some were practical problems such as the lack of applications that could be run on both CPU architectures and the similarity in the microservices, others were more technical such as the interdependence of microservices and its impact on the power consumption measurements.

Similarity in microservices

The first issue we encountered was the similarity in the microservices forming the application, at least for the ones available in the DeathStarBench suite. For instance, the social network app is composed of 26 microservices plus another one writing logs

but those services are all made from a set of only 7 distinct Docker images. In fact, most nodes are conceived of the 3 same images, namely a custom server based on Apache Thrift [27], a MongoDB database [28] and an in-memory caching service such as memcached [16] or redis [24]. Consequently, the variety in the microservices is rather low even though there are 26 microservices composing the application.

Complexity of finding applications supporting both architectures

Frameworks with relevant applications composed of diverse microservices together that include the databases and the workloads are quite rare. DeathStarBench is one of them and even though its applications were supporting only the x86 architecture by default, we could add the support for ARM CPUs to some of them. Yet, we managed to port only 2 applications, which is not a lot to study the energy consumption behaviours of the CPUs. Most applications we found were either too small and not relevant, or created only for x86 CPUs for which adding support for ARM processors would be a tedious and time-consuming process.

Interdependence of microservices

The last problem of microservices applications is the interdependence of microservices. It is very difficult to isolate a microservice to measure its power consumption under load because a microservice receiving lots of requests will also exchange many requests with all the other services that it depends on, or that rely on its encapsulated data. Consequently, the whole application suffers from the load. Moreover, there are often a few microservices that have to take heavier work upon request reception preventing the characterisation of the energy consumption of the majority of microservices.

3 JWT token generator offline serverless function

Listing 5 shows the Dockerfile of our custom-built Python JWT function. It is based on the python3 alpine docker image. It copies the different files needed by the function to run properly, installs the requirements and sets the entry point to be executed at container startup.

```
ARG PYTHON_VERSION=3
FROM python:3-alpine

WORKDIR /home/app/

COPY function.py         /home/app/
COPY entrypoint.py       /home/app/
COPY requirements.txt    /home/app/

RUN pip install -r requirements.txt

ENTRYPOINT [ "python3" , "entrypoint.py" ]
```

Listing 5: Custom Python JWT serverless function Dockerfile

Listing 6 shows the entry point function written in Python. That entry point is the same for every function. The function takes two arguments, the running time (t) and the number of CPUs allocated (x). It starts x processes, each running the actual function during t seconds. Finally, it waits for all processes to finish and collects their respective number of instructions executed.

```
from argparse import ArgumentParser
import subprocess
import shlex

if __name__ == "__main__":
    parser = ArgumentParser()
    parser.add_argument('time')
    parser.add_argument('cpus')
    args = parser.parse_args()

    nbr_cpus = int(args.cpus)
    processes = []
    for i in range(nbr_cpus):
        command = f"python3 function.py {args.time}"
        process = subprocess.Popen(shlex.split(command),
        ↪  stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        processes.append(process)

    total_execs = 0
    for p in processes:
        (stdout, stderr) = p.communicate()
        if stdout:
```

```
            total_execs += int(stdout.decode().strip())
        if stderr:
            raise RuntimeError("Error during python3 JWT execution : " +
            ↪  stderr.decode())
    print(total)
```

Listing 6: Custom Python JWT serverless function entry point

Listing 7 shows the actual Python JWT function code. It receives as an argument from the entry point the time it needs to run. It then loops during t seconds and counts the number of executions performed.

```
from argparse import ArgumentParser
import time
import jwt

ALGORITHM = 'HS256'
EXPIRATION_TIME = 3600
SECRET_KEY = "our_secret_key_please_do_not_steal"

def generate_jwt_token(user_id: int):
    expiration_time = int(time.time()) + EXPIRATION_TIME
    payload = {
        'user_id': user_id,
        'username': 'example_user',
        'role': 'admin',
        'exp': expiration_time
    }
    token = jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)
    return token

def handle():
  userID = 42
  generate_jwt_token(userID)

if __name__ == "__main__":
    parser = ArgumentParser()
    parser.add_argument('time')
    args = parser.parse_args()
    start_time = time.time()
    nbr_func = 0
    while (time.time() - start_time) < int(args.time):
        handle()
        nbr_func += 1
    print(nbr_func)
```

Listing 7: Custom Python JWT serverless function code

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**