Hochschule
München
University of
Applied Sciences

FAKULTÄT FÜR INFORMATIK

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

Master Thesis in Computer Science

Measuring Energy Consumption in Virtualized Kubernetes Clusters

Energieverbrauchsmessung in virtualisierten Kubernetes-Clustern

Author:	Maximilian Hösel
Matrikelnr:	11576019
Advisor:	Prof. Dr. Benedikt Dietrich
Supervisor:	Boris Heisel
Submission Date:	05.04.2025

Acknowledgements

I would like to thank Boris Heisel and Fabian Müller for all their support as supervisors on behalf of Kraut.Hosting GmbH. Their insights provided during the creation of this thesis proved very valuable and their support was always appreciated. By generously providing access to high-end enterprise server hardware they enabled the testing, refinement, and verification of the implementation to a much higher standard than would otherwise have been possible.

I would also like to thank Antonio Di Turi for providing the original inspiration for the topic of this thesis, as well as providing valuable pointers for the foundation of the attribution model presented in chapter 5.

Finally, I would like to thank Prof. Dr. Benedikt Dietrich for originally sparking my interest in Green IT and server-side energy consumption and subsequently mentoring me during the creation of this thesis.

Abstract

With the rise of cloud computing and "AI"/LLM services, computing-as-a-service has become a key utility for an increasingly digitized society, as well as a significant contributor to global energy consumption and thus climate change. Optimizing software for energy-efficiency is one way to reduce computings carbon footprint, but requires the existence of quality energy consumption metrics - which are often unavailable in modern cloud computing environments. Todays cloud workloads typically run in highly abstracted environments, such as inside a Kubernetes cluster on top of virtual machines, far removed from the actual hardware. Not only does his obscure any available hardware-level energy metrics, the high level of isolation offered by these abstractions also allows many clients to share hardware resources, making accurate attribution to a single application extremely difficult. While progress has been made towards overcoming this barrier, many solutions rely on estimates with poorly-known accuracy, or address only part of the problem space. This thesis provides a comprehensive approach for modeling energy consumption in a shared, abstracted environment based on available hardware data. The core of this work is a foundational model that can attribute energy consumption to processes across virtualization and containerization boundaries, utilizing proven attribution methods adapted for cloud environments. This is followed by a demonstration of the model through an vertically-complete implementation on controlled hardware, with attribution accuracy up to 95% in selected tests and with further potential for growth. When compared to existing attribution solutions relying on estimation, the model provides more accurate and much more consistent attribution overall, further highlighting the benefits of a hardware-backed solution. In conclusion, good attribution results can be achieved today even with simple tooling, although the prototype implementation still shows weaknesses, such as performance at low loads. Future work is still needed to make hardware-backed energy attribution feasible for cloud environments.

Kurzfassung

Die ständige Verfügbarkeit von Diensten wie Cloud-Computing und "AI"/LLMs ist ein zentraler Treiber der fortschreitenden Digitalisierung der modernen Gesellschaft. Gleichzeitig tragen derartige Dienste und die dazugehörige Infrastruktur zunehmend zum globalen Energieverbrauch (und somit zu Treibhausgasemissionen und dem Klimawandel) bei. Eine Möglichkeit zur Reduktion der IT-verursachten CO2-Emissionen ist die Optimierung von Software auf Energieeffizienz. Dafür sind jedoch hochwertige Metriken zum Energieverbrauch erforderlich, welche in modernen Cloud-Umgebungen oft nicht verfügbar sind. Heutige Cloud-Workloads laufen typischerweise in stark abstrahierten Umgebungen, etwa in Kubernetes-Clustern auf virtuellen Maschinen, ohne Zugriff auf die eigentliche Hardware und eventuell vorhandene Inferfaces zur Stromverbrauchsmessung. Gleichzeitig erschwert die geteilte Nutzung von Hardware durch mehrere isolierte Nutzer die Zuweisung des Energieverbrauchs an einzelne Prozesse. Bereits existierende Ansätze nutzen oft Schätzwerte mit schwer abschätzbarer Genauigkeit für den Stromverbrauch oder bedienen nur Teil des Problems. Diese Arbeit präsentiert einen vollständigen, Hardware-basierten Ansatz für die Energieverbrauchszuweisung in gemeinsam genutzten, abstrahierten Umgebungen. Kern der Arbeit ist das entwickelte Zuweisungsmodell, welches mit bewährten Methoden die Energieverbrauchszuweisung über VM- und Container-Grenzen hinweg ermöglicht. Die demonstrative Implementierung erreicht auf eigener Hardware bereits Genauigkeiten von bis zu 95%, mit Potential für zukünftige Verbesserungen. Im Vergleich mit auf Schätzungen basierenden Ansätzen zeigt die Implementierung oft bessere Genauigkeit und deutlich höhere Konsistenz, ermöglicht durch die hardwarebasierte Natur der Verbrauchsdaten. Zwar leidet die Implementierung unter Schwächen bei geringer Auslastung, dennoch beweist sie, dass akkurate Verbrauchs-Zuweisung auch heute bereits mit einfachen Mitteln erreichbar ist. Ein produktiver Einsatz in Cloud-Umgebungen erfordert dennoch weitere Fortschritte, welche abschliessend skizziert werden.

Contents

1. Introduction

The nature of computing is evolving. More than ever before, computational power has become a commodity for developers and users alike. The availability of near-endless, on-demand computing power has made previously impractical use cases trivial, with the most recent example being the surge of "AI"/LLM-based services capable of natural language-based processing. Since many of these models are prohibitively demanding for an individual user and their device, they are instead run in the cloud, that is, on centralized infrastructure inside of datacenters. The rise of such services is the latest reason for the drastic increase in datacenter count and resource consumption: Over the last decade, the amount of power allocated for datacenters has grown significantly, with a per-year increase of as much as 30% for some locations in recent years [1][2]. Figure 1.1 shows recent capacity growth in several key locations. Note that total capacity is now in the GW range for some regions, equivalent to multiple conventional power plants.

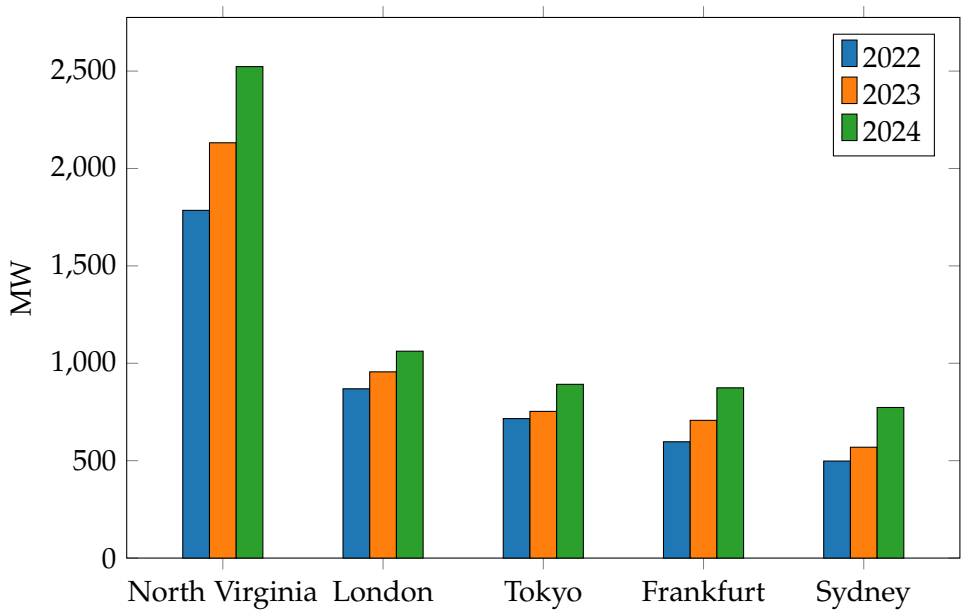


Figure 1.1.: Capacity available for data center power consumption in several key markets from 2022-2024 [1][2]

In 2022, data centers were already estimated to consume approximately 300 TWh, or 1% of the global energy consumption [3]. As of 2024, there are now over 11 000 data centers worldwide, and the recent wave of Machine Learning-based products is bound to increase

this number even further. Large players in this field have already announced the creation/re-activation of not just new datacenters, but entire power plants for the specific purpose of LLM training and operation [4]. With computing services becoming more accessible to users worldwide and increasingly becoming an integral part of modern society, the aforementioned 1% share is now forecast to increase significantly in the coming years.

This development runs counter to a more sustainable computing future. With climate change ongoing and predictions worsening, the computing industry must too make an effort to reduce its environmental footprint in all aspects of its lifecycle. The umbrella term "Green IT" encompasses many of these efforts, including the design of more energy-efficient products, the reduction in raw materials for component production, an increase in reuse and recycling of hardware, and the authoring of more energy-efficient software. While recent "AI" and associated developments are likely to have a negative impact on the computing industries overall carbon footprint, increased effort has been made to reduce the footprint elsewhere. For example, new data centers may implement energy-efficient construction, heat management, or efficiency-focused hardware, resulting in higher power usage effectiveness (PUE) [5][6]. At the same time, modern consumer devices feature increasingly sophisticated energy-efficiency mechanism, ranging from low-power modes to specialized hardware such as efficiency-oriented CPU cores for background workloads. Efficiency improvements can also be made on the software side: Many tactics for improving energy efficiency exist here, such as more efficient algorithmic processing, lower-overhead implementations or smarter energy budget allocation (such as Race-to-Idle). To know the true effects of any such tactic, one must however be able to measure software energy consumption with sufficient accuracy. Without this data developers are forced to work blind, effectively guessing about energy consumption or usage patterns. If those guesses prove incorrect, the implemented tactics may be ineffective or even backfire, increasing energy consumption further due to unforeseen consequences. Thus, the availability of energy consumption data is a requirement for optimizing applications and services energy efficiency.

1.1. Problem Description

Despite their importance for improving software efficiency, acquiring energy consumption data remains a major challenge for any environmentally concious developer or operator.

Many modern platforms have facilities to self-report their energy consumption through special hardware interfaces. For example, modern servers may feature power monitoring at the Power Supply Unit (PSU) level, while modern Intel CPUs can report their energy consumption at the microjoule level through an interface known as Running Average Power Limit (RAPL). In theory, such interfaces could enable developers and operators to make more informed decisions about how to improve energy efficiency, allowing them to reduce their own systems carbon footprint. While this data is accessible on most OSes, there are two major hurdles that hinder usage of these interfaces for energy reporting.

First, the hardware-reported information is usually not differentiated: Interfaces such as

RAPL are not aware of different processes or applications running on the same system and merely provide a summed value [7]. This means that energy attribution for any system with more than one application needs a way to attribute fractions of the reported energy consumption to running processes proportionally. Doing so requires construction of a model that maps the information available about running processes to their relative energy consumption, a non-trivial task.

Second, accessing metrics such as RAPL requires that the attribution solution has direct access to the host machine and its hardware interfaces. This runs counter to the increasing trend of abstraction in application deployments. As applications grow in scope and complexity, modern application deployment is increasingly driven by a need for flexibility and scalability, alongside encapsulation. Instead of running directly on physical servers, many applications are now confined to a Virtual Machine (VM) with no direct access to the hardware, separated from possibly numerous other applications running on the same machine through a hypervisor. Usage of VMs has its benefits of course - it isolates applications for increased security and reliability, allows operators to work at a higher logical level and, when combined with cloud computing solutions, provides increased scalability. In fact, the rising adoption and thus influence of cloud computing (where the use of VMs is generally a basic assumption) has made bare-metal deployments increasingly rare outside of specialized scenarios. This is challenging for energy attribution, as such VMs generally do not have access to the physical hosts energy consumption interfaces. In fact, exposing this interface to VMs unmodified may even pose a security risk and would thus be undesired by cloud providers [8]. Unfortunately, even if a VM was given access to these hardware counters it would be of little use: Since the readings are likely to be undifferentiated, they would contain the consumption of all VMs running on a given host, not just the targeted VM. Without a way to filter this data for individual VMs, using it for attribution would be very difficult on a crowded machine with noisy neighbour VMs.

The use of bare cloud-provided VMs is commonly considered to be part of Infrastructure-as-a-Service (IaaS) offerings. However, modern cloud offerings go much further than managing just the virtual hardware: Offerings such as Platform-as-a-Service (PaaS) or Function-as-a-Service (FaaS) also manage aspects such as high-level networking, the operating system and even the application runtime environment. This delegation of responsibility can be seen in Figure 1.2, with more sophisticated offerings moving more of the underlying stack into the hands of the cloud provider. This further worsens the challenge of energy attribution based on hardware counters, as now applications may not even have access to the (Virtualized) operating system, let alone the energy hardware metrics.

A particularly common approach to higher-level service offerings is the use of containers inside an orchestration platform, such as Kubernetes (k8s) or OpenShift. By shipping applications as a container image along with all their direct dependencies, they can be deployed and scaled easily, even more so than with just VMs. Container orchestration engines such as Kubernetes enable this by providing a resource-oriented interface for managing services and their dependencies, effectively a PaaS-style offering. While Kubernetes clusters can be deployed directly on physical hosts, managed offerings from cloud providers are also

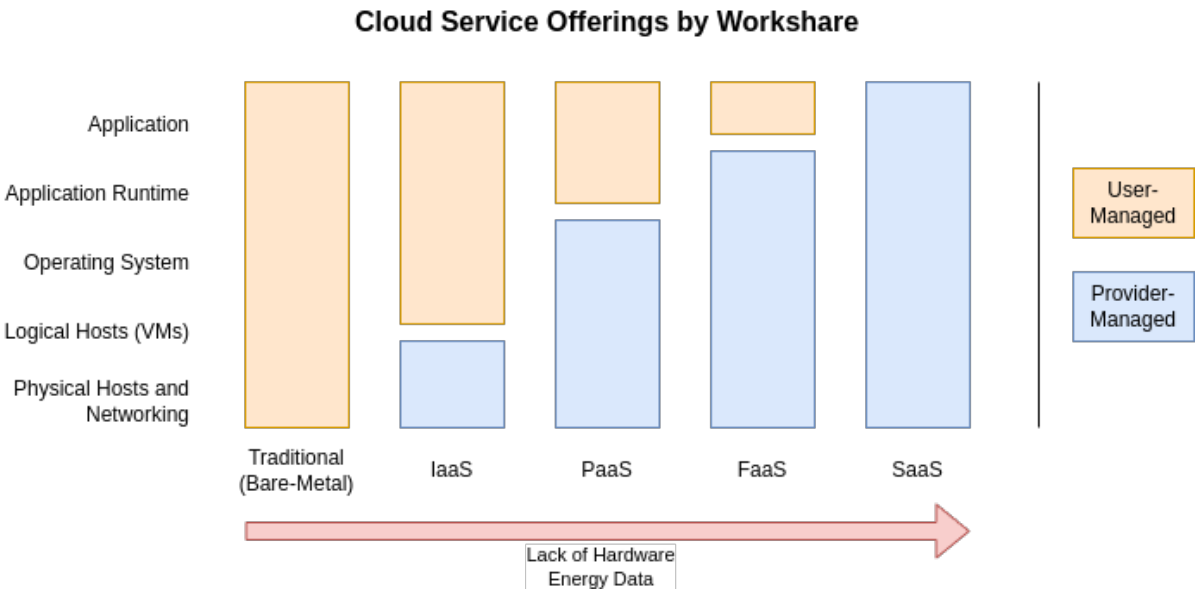


Figure 1.2.: Workshare of different cloud service offerings and corresponding difficulty of acquiring hardware energy consumption data

available (such as AWS EKS or Google GKE) and allow for easier on-demand scaling. In such offerings, the individual hosts ("nodes") comprising a cluster are usually VMs running on top of a hypervisor. As such, an application running inside a cloud providers Kubernetes cluster is effectively abstracted from the physical hardware twice - first through containerization, then through the VMs boundary.

The result of this abstraction can be seen in Figure 1.3: An individual application is highly decoupled from the physical hardware it is running on, both in technical terms (abstraction) as well as organization terms (provider-managed hardware). While this may be desirable for flexibility and scaling, it is highly challenging for energy consumption measurement and optimization. In theory, cloud providers could provide users with energy consumption metrics by implementing attribution on their systems, however no major provider appears to do so as of 2025. Reasons for this may include the aforementioned security concerns, the difficulty of constructing a proportional and accurate attribution model, or lack of economic incentive and interest.

As a result of this lack of data, several projects have sprung up to fill in the gap. Many of them rely not on hardware-reported energy metrics, but on estimates produced manually or sometimes using trained machine learning models [9]. This indicates that there is a genuine demand for energy consumption data, even if it is generated by imperfect means. However, estimate-based data is inherently less reliable and may require additional verification to be trusted. Providing stakeholders access to hardware-based attribution data is therefore still valuable and should lead to increased efficiency for modern cloud-based applications.

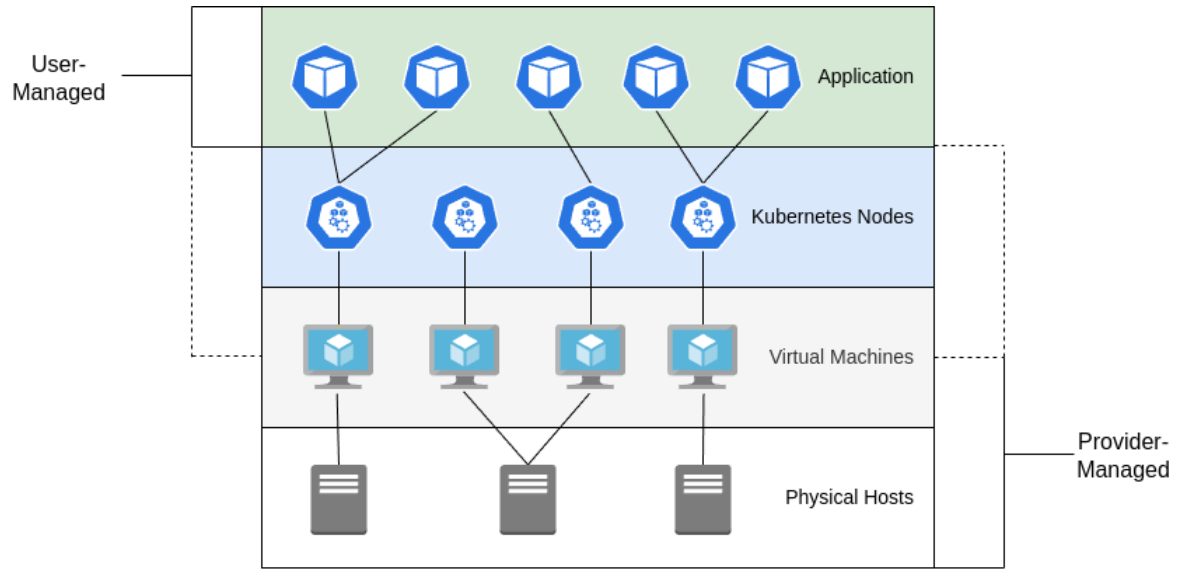


Figure 1.3.: Abstraction layers for an application running on a modern cloud and Kubernetes-based architecture. Management of Kubernetes nodes is done by the provider in case of managed offerings and by the user if the cluster is managed manually

1.2. Thesis Goals

The lack of hardware-backed energy consumption data presents a significant problem for developers looking to optimize their softwares efficiency in modern containerized cloud environments. Lifting this limitation requires a practical model for energy attribution across the aforementioned abstraction layers, an implementation of said model, and the validation of the implementations attribution data. This thesis aims to cover these points in detail, providing some of the fundamental building blocks for platform operators to build their attribution solutions on. To achieve this goal, each problem and its associated research question are addressed in sequence, starting with the construction of an attribution model.

The purpose of such a model is the attribution of consumed energy to individual applications, even across virtualization and containerization layers. The model should describe the logical steps needed to attribute energy consumption in each layer. It should provide a mathematical foundation, describing the required data sources, attribution algorithms and potential limitations. This model would serve as a base for future implementations, allowing such solutions to focus their efforts on wide compatibility or acquisition of quality data, rather than having to deal with the mathematical process of attribution itself. Constructing this model presents the first step towards better energy attribution in modern deployments and is the first goal of this thesis. Not only will this provide insights into the intricacies of energy attribution in complex systems, it will also give future implementations solid ground to build on. The construction of this model will draw upon existing attribution solutions for less complex scenarios, combining them to arrive at a Cloud-Kubernetes compatible solution. Thus, this thesis first research question concerns the construction of this model: **RQ1: How can one define a conceptual model for energy attribution in a virtualized Kubernetes cluster?**

To prove that this model is capable of providing quality data in a real-world environment, part of this thesis will be in the form of a demonstration implementation. The goal of this implementation is to provide a vertically complete reference that addresses all technical challenges of energy attribution in the given scenario. This thesis details the construction of the implementation on a technical level, followed by the deployment in an environment that mirrors a cloud-based setup to evaluate its basic capabilities. This deployed implementation can then be tested with regards to attribution accuracy and consistency, with the goal of high enough precision to enable application developers to optimize their workloads. As such, the second research question is: **RQ2: How can one construct an implementation of the model from RQ1 and how does it fare with regards to accuracy?**

Finally, solutions implementing energy attribution in the above scenario via machine-learning or other estimate-based approaches already exist. Such tools present not only a compelling, established alternative but also provide another reference to compare the final implementation to. Comparing their accuracy and reproducibility with the model-backed implementation should provide insights into their strengths, weaknesses, and usability for the purpose of application optimization. To do so, part of this thesis deals with executing benchmarks against these solutions and the proposed implementation in a controlled environment. This data will allow operators and application developers to make an informed decision about which attribution solution to use and whether a hardware-backed implementation like the one presented in this thesis will provide a real benefit in real-world usage scenarios. This forms the core of the third question: **RQ3: How does the attribution implementation from RQ2 fare against existing estimate-based solutions with regards to accuracy?**

2. Background

To answer the stated research questions, a solid understanding of the problem space and its detailed aspects is needed. The goal of making hardware-based energy attribution available even in abstracted environments can be broken down into three separate aspects: Hardware-assisted energy metrics, energy attribution, and overcoming virtualization and Kubernetes barriers. Since the energy attribution work relies on a solid understanding of hardware metrics and the involved abstraction layers, this chapter only covers the latter, while energy attribution is the focus of chapter 4.

2.1. Hardware Energy Measurement

The measurement of a systems energy consumption is at the core of the issues described so far. Not only are there many methodologies for doing so, understanding the difference between them is crucial for gaining a better understanding of the problem at hand. As a first step, consider a scenario with a system that is only accessible from the outside: Its energy consumption is a simple metric that can be measured by monitoring the energy consumed by the systems PSU. This is precisely what happens when one puts a power meter in between an appliance and the wall outlet: This simply monitors the power that is consumed by the device and compensates for elapsed time to arrive at an energy reading. One can also do so for servers, either by measuring with an external meter or by using a PSU with power reporting capabilities. This external approach to power monitoring is versatile and works well for appliances and simple loads, partially because it requires little understanding of the system under test [10].

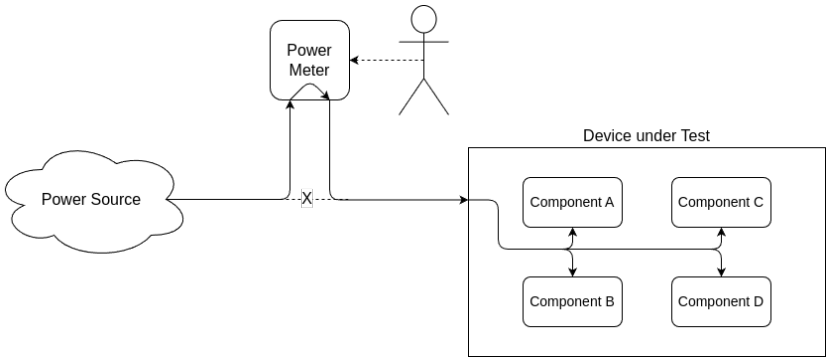


Figure 2.1.: External energy measurement setup

However, this approach becomes limiting when the system under test is more complex

and contains many subsystems, as is the case with computing hardware. For servers, this complexity increase comes in two dimensions:

First, servers (and computers in general) contain many different components that are heterogenous in their energy consumption behavior. CPUs and GPUs are often the first components mentioned when discussing energy consumption, but depending on the system and the applied load, many other components such as DRAM, storage or cooling/fans can have a large impact on energy consumption as well. This is on top of components with largely static power consumption such as mainboards, system backplanes or management interfaces. Since this method can only ever measure the sum of all components, any attempt at isolating an individual components energy consumption will be hindered by the noise from other components.

Second, a single server will typically runs multiple processes, both in parallel through multi-core CPUs and concurrently through time-based scheduling. The load (and thus energy consumption) of a server therefore depends on whatever process is scheduled to run at any given instance, with each process having a different component usage pattern. For measuring process energy consumption, this temporal noise from other processes makes accurate measurement very challenging. Put another way: By only measuring external power consumption one can only ever really observe the sum of all components and all processes. This temporally and spatially limited view can still be useful as a reference or for simpler systems with few components, but it makes per-application energy monitoring highly challenging.

Thankfully, modern server platforms often come with the ability to self-report their energy consumption via hardware sensors. The results of this self-reporting are then made available to the operating system via interfaces, such as RAPL on x86_64 platforms. Originally developed by Intel, RAPL is now available on both Intel and AMD platforms and provides both energy consumption metrics as well a way to limit system power (hence "Average Power Limit") [7]. Under Linux, RAPL is exposed to applications via the `powercap` driver as a series of files located in `/sys/class/powercap/intel-rapl/`, the structure of which is shown in Figure 2.2. The RAPL interface provides at least one "domain" (`:0` and `:0:0` in this case), with each domain having its own energy consumption counter. To read a given domains energy consumption since startup in Microjoules, one simply needs to read the numeric value from its `energy_uj` file.

We can then calculate instantaneous power consumption P_t by taking repeated energy readings E_t and dividing the difference between readings by the difference in elapsed time $\Delta t = t - t_{-1}$:

$$P_t = \frac{E_t - E_{t-1}}{\Delta t} \tag{2.1}$$

This internal energy monitoring approach has some immediate benefits over the external method: For one, the metrics are much more accessible since there is no need for additional hardware or an experimental setup, one only needs OS-level access. Second, these metrics can have very high temporal accuracy (in the case of RAPL the unit is Microjoules). This is key for separating energy consumption between processes, as operating systems can execute

```
/sys/class/powercap/intel-rapl
|-- enabled
|-- intel-rapl:0
|   |-- enabled
|   |-- energy_uj <-- energy counter
|   |-- intel-rapl:0:0
|   |   |-- enabled
|   |   |-- energy_uj <-- energy counter
|   |   |-- max_energy_range_uj
|   |   |-- name
|   |   |-- power/
|   |-- max_energy_range_uj
|   |-- name
|   |-- power/
|-- power/
```

Figure 2.2.: Truncated directory layout of the `powercap` RAPL driver on Linux

thousands of scheduler actions per second.

Another key feature of RAPL and many similar interfaces is their ability to differentiate between components inside a system. RAPL divides a system into a tree structure of "domains", wich each domain representing a different system component. A logical overview of the domain structure can be seen in figure Figure 2.3. Here, the `pp0` domain (CPU Cores) is a subdomain of the `pkg` domain (CPU Package), which is in turn a subdomain of `Psys`. For multi-socket systems, several sockets with independent domains are created [11].

Since each domain has its own energy consumption counter, the consumption of multiple components can be tracked accurately and independently. Therefore, hardware-assisted counters such as RAPL are capable of dealing with both of the aforementioned complexities, making them a good choice for energy attribution like for this work.

Unfortunately, hardware counters like RAPL also come with shortcomings. For one, RAPL is only available on x86_64 systems. Other platforms may provide similar interfaces, but there is currently no widely supported standard for popular platforms such as ARM-based systems. Given the increasing adoption of ARM systems in the server space [12] this lack of standardization is unfortunate and will hinder energy monitoring on those platforms. A more serious concern is the possibility of under-reported energy consumption. Since RAPL only monitors at the component level, any components not mapped into a RAPL domain (such as storage devices or cooling) will not be metered and their energy consumption will be "lost". The number and scope of available domains are dependent on the particular system and range from fully-featured, to only covering the CPU package + cores. Additionally, RAPL does not monitor GPU power consumption at all, though equivalent interfaces are available from major GPU manufacturers [13]. This means that while RAPL can often provide a highly proportional energy reading, its metrics should still be cross-checked with external power

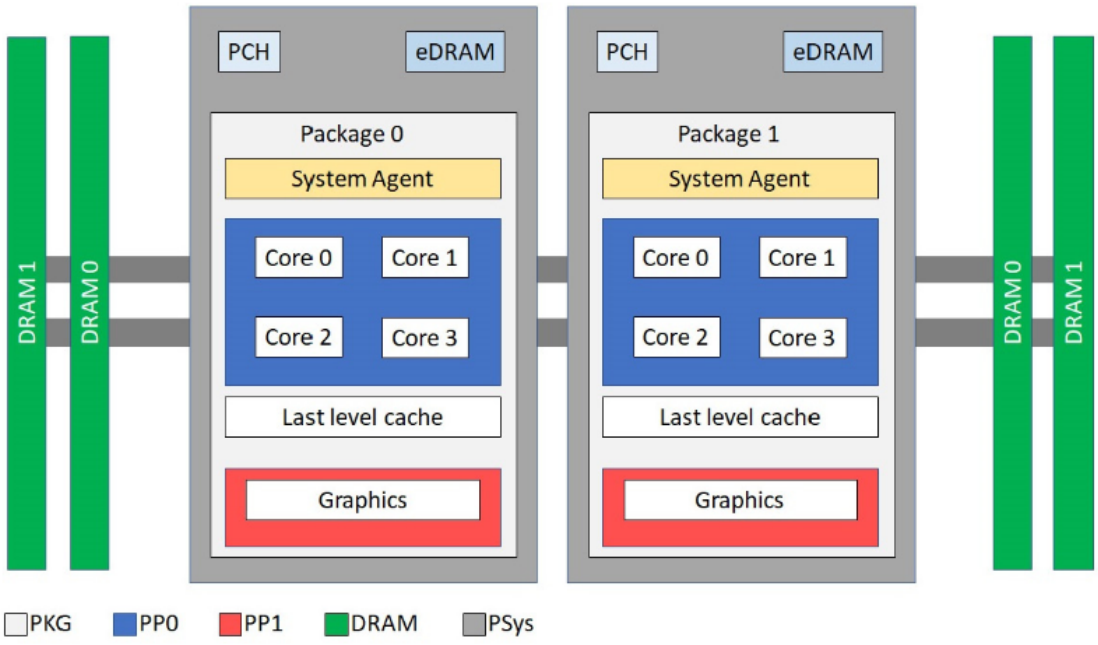


Figure 2.3.: Example of power domains as modeled by RAPL [11]

measurements to ensure the absolute values are within reason. Aside from accuracy issues, systems such as RAPL can also pose a security risk through side-channel attacks [14]. Such information leaks can be mitigated by hardware vendors, but system operators may still want to perform a risk assessment before exposing data from such interfaces

Some server systems also support a form of "integrated external" monitoring through the PSU. In such systems, the PSU provides power consumption data to either the OS or a Baseband Management Controller (BMC) through an interface like the Data Center Manageability Interface (DMCI). These readings are usually of similar granularity to an external reading, but their easier availability still makes them a good solution for cross-checking RAPL data.

2.2. Abstraction, Virtualization and Kubernetes

If every server only ran a single application, energy attribution would be trivial: In this scenario, one could simply consider the hardware energy data from the previous section to be the applications energy consumption. However, as discussed in section 1.1, this is usually not the case anymore, and it is strictly untrue for the scenario covered in this thesis. Instead, the given scenario requires an attribution solution to deal with a stack consisting of 4 abstraction layers (see Figure 1.3) that build on top of each other, with the application being located in the topmost one. Underneath that layer, virtualization and containerization can both result in more than one application running on the same server, along with the regular

multi-tasking capabilities of operating systems. A comprehensive understanding of how these layers function, their purpose, and their interactions with adjacent layers is essential for determining how an attribution solution can effectively deal with them.

2.2.1. Virtualization

Virtualization has been a fundamental building block of application hosting for over a decade. Hardware-Level Virtualization as discussed in this thesis differentiates between a host (the physical machine) running a hypervisor and guests inside VMs. Each guest is assigned a subset of hardware resources, such as CPU cores, DRAM and Storage. The hypervisor then acts as an intermediary between the guests and the underlying hardware - it performs scheduling and mediates I/O access on the guests behalf. The guests are isolated from each other, effectively acting as individual machines with no knowledge of other guests running on the same host. While this kind of virtualization is possible in pure software, this comes with significant performance overhead and thus cost. In practice, virtualization is usually assisted by the host through hardware support, and often further accelerated by virtualization-aware guest operating systems. Exact implementations and terminology surrounding Virtualization varies (see Type 1 vs. Type 2 hypervisors, privileged guests such as `dom0` and paravirtualization), but these details are not relevant to understand the challenge virtualization poses to energy attribution and are thus not elaborated on further in this thesis.

Virtualization can be beneficial in multiple ways: For one, it avoids the overhead and fixed costs associated with running multiple physical machines, such as supplemental hardware and energy consumption. The heightened level of abstraction also allows for more automation and better scalability in administration - creating a new VM can be done in seconds, miniscule compared to the lead-up time for ordering and provisioning a physical host.

A simple scenario in which virtualization can be used is to split up a single large server into several smaller virtual machines for applications with lower demands. This works well for applications with static demands and helps avoid the fixed costs associated with multiple servers. For applications with dynamic (and especially burst-heavy) demands, over-provisioning can be used, a simplified example of which is shown in Figure 2.4: Here, each guest is given access a larger subset of resources capable of meeting the applications maximum expected load. The sum of resources assigned to guests may exceed the physically available resources on the host. However, as long as different applications experience their load spikes at different times, they will not interfere with each other and performance will remain unaffected. Over-provisioning is a common tactic to increase utilization and increase application density, though the level of over-provisioning must be monitored carefully to ensure performance remains acceptable.

For all its advantages, virtualization imposes significant challenges to energy measurement form the perspective of the guest. First, hardware-assisted internal monitoring typically makes use of specific hardware interfaces (such as RAPLs Model-Specific Registers [7]). Access to these is disabled by the hypervisor as these interfaces are not virtualization-aware,

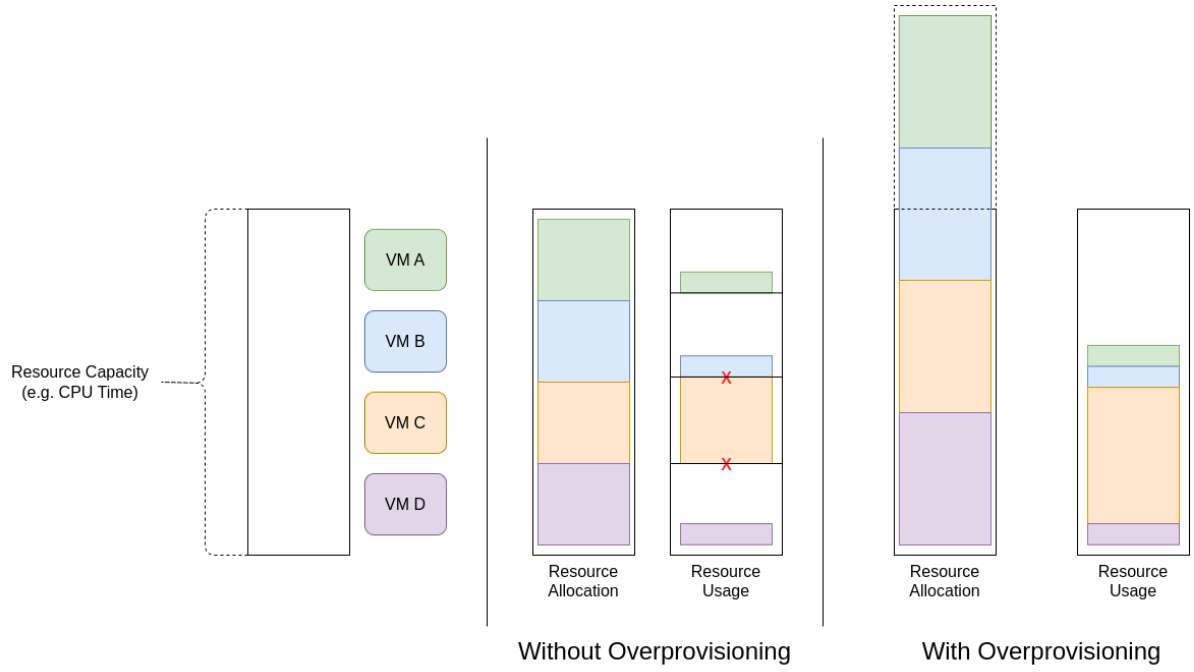


Figure 2.4.: Example for overprovisioning. On the left, each VM is assigned a fixed number of resources and no overprovisioning is performed. During a utilization burst, the orange VM then runs into resource limits and is slowed down. On the right, all VMs have been assigned more resources than are physically available (overprovisioning). However, since not all VMs use all their resources at the same time, this works. The orange VM can use a larger part of the available resources for its burst workload.

meaning that exposing them could reduce or even break isolation between guests. This makes measuring internal energy consumption measurement inside of VMs impossible without implementing further measures. External monitoring does remain possible in principle, especially if the monitoring is accessible over the network (host-attached devices such a via USB would be subject to the same hardware isolation principles). However this method provides no VM-level granularity on its own, limiting its usefulness. One may conceive an estimation-based approach where energy consumption is attributed by multiplying the measured external consumption with the fraction of the guests assigned resources compared to the host, but this still requires knowing the hosts physical resource size. Over-provisioning is also likely to make such an estimate highly inaccurate due to noise from other VMs affecting the measured energy. Both energy metering methods thus fail to penetrate the VM barrier on their own.

2.2.2. Containerization and Kubernetes

Just like virtualization provides an abstraction between hardware and operating systems, containerization provides an abstraction between application runtimes and the underlying operating system/kernel. Containerization makes use of OS-level mechanisms to isolate and contain individual processes, providing the containerized process with a limited and controlled view of resources it can access. The exact mechanisms involved depend on the operating system - under Linux this includes `cgroups` for hardware resource isolation and `chroot`/bind mounts for limiting filesystem access. Using these mechanisms allows creating environments with their own isolated file systems, process views and hardware access, only sharing a common kernel (that of the host operating system). A schematic view of containerization can be seen in Figure 2.5 - note the lower amount of overhead (and thus performance loss) required for containerization when compared to fully-fledged VMs. This process is typically performed by a container runtime such as Docker, containerd or podman, which also provide additional integrations such as networking and lifecycle management.

In order to make applications work inside a container runtime, they must be *containerized* - this involves packing the application and all its runtime dependencies into a so-called container image, which can then be distributed. On the target system, the container runtime then mounts this image as the virtual root filesystem inside the container and uses metadata included with the image to find and launch the application executable.

The principal benefit of containerization lies in the separation of the operating system and applications (and their dependencies): By bundling everything required to run an application into a container image, that image is now portable across different hosts, operating system variants and hardware, as long as a compatible container runtime is present. The image format and runtime itself has in turn seen increasing standardization by organizations such as the Open Container Initiative [15], allowing use of the same image across different environments such as cloud provider services or locally hosted machines.

Containers neatly package application deployments and components. Container orchestration

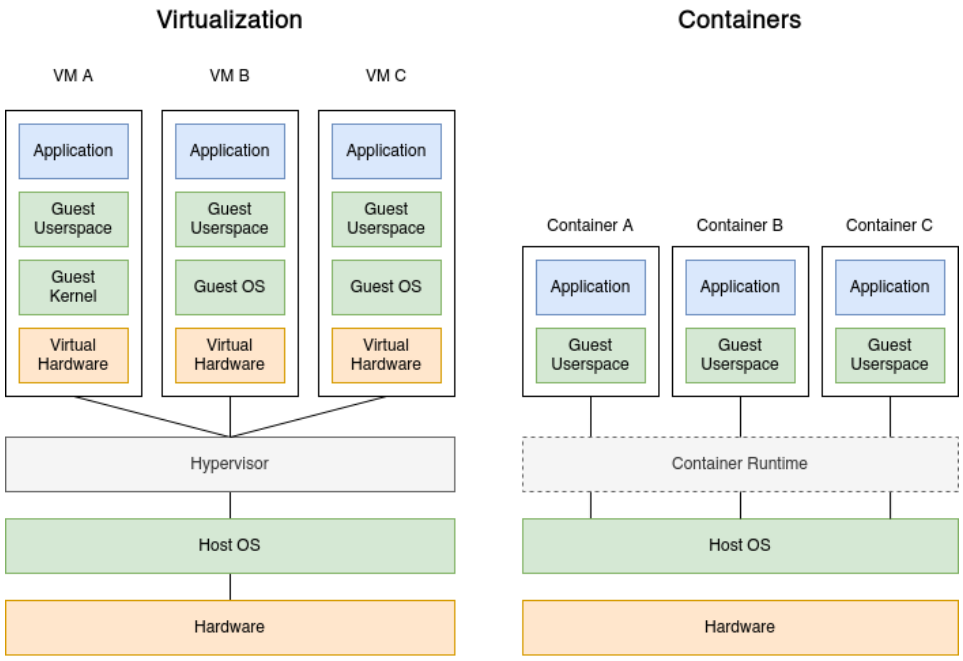


Figure 2.5.: Containerization (OS-Level virtualization) as opposed to bare-metal virtualization

solutions such as Kubernetes (k8s) build on top of this foundation to enable more complex workload management. Kubernetes' core goal is to provide a resilient, self-healing system that optimizes resource utilization while abstracting the complexity of infrastructure management from its users. To do so, Kubernetes' "control plane" exposes an API in which all aspects of an application deployment (such as containers, secrets and networking) are presented as resources. Developers can then create or update these resources in a declarative manner, while the control plane ensures the specified resources are in their desired state. On the infrastructure side, a Kubernetes cluster consists out of the control plane, as well as one or more worker nodes. Container workloads are scheduled by the control plane and executed by the worker nodes.

While Kubernetes provides a number of interfaces and API definitions for purposes such as networking or storage, it does not provide implementations for many of these itself. Instead, different Kubernetes distributions combine the core Kubernetes with components such as cluster networking, ingress and a storage solution. Many cloud providers also integrate their managed services into Kubernetes by plugging into the existing APIs/Interfaces. As such, Kubernetes has become a common denominator for deploying applications in a cloud-native context, providing developers with a well-known and resource-oriented API to define application runtime requirements.

Unfortunately, containerized applications (Kubernetes or not) also struggle with energy consumption visibility: To ensure proper isolation, direct access to hardware interfaces is typically restricted, as is access to kernel-level interfaces such as the `powercap` driver. For

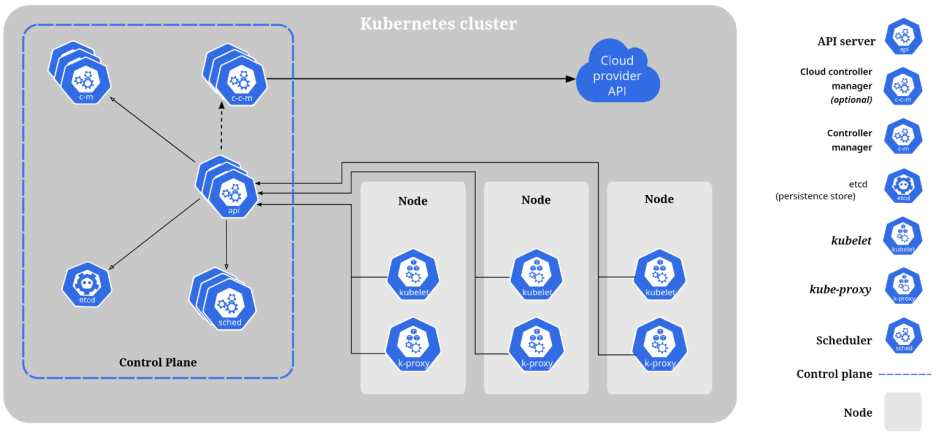


Figure 2.6.: Overview of the Kubernetes Architecture and common components

example, the `/sys` directory containing the powercap interface is generally not passed through to containers as this would present a security risk. Therefore, "normal" containers are blind to their energy consumption, similar to virtual machines. However unlike with hardware-level virtualization, this isolation can be selectively broken up: A given container can be provided access to specific host directories, allowing access to `powercap`-like interfaces. Alternatively, a container can also be given specific capabilities (enforced by the Linux kernel) that permit the container to perform near-hardware actions to access the required interfaces [16]. This provides an "escape hatch" for energy attribution within the context of containerization. Of course, containerization, just like virtualization, also has the goal of running multiple applications at the same time on a single host, resulting in the aforementioned noise issue.

3. Related Work

Before discussing approaches for answering RQ1 (attribution model construction), this chapter examines existing work in this problem space. Existing research into application energy attribution already covers a wide spectrum, but there is little discussion about layered scenarios. Still, it is possible to base a model on this research by taking models, approaches and tactics from existing works where they suit into the layered scenario. On the other hand, several implementations have already begun to explore the cloud-native energy attribution problem space using differing approaches. This chapter therefore introduces and discusses both existing research and implementations, evaluating their findings and methodologies with regards to the given scenario.

3.1. Attribution Models

To begin, Centofanti et al. provide an overview of the current state of the art, giving both a comprehensive enumeration of existing attribution solutions and a more detailed discussion for a small subset. The authors also evaluate the accuracy of several implementations for their test scenario [9]. Their review of the field can serve as a basis for identifying attribution models worth considering. Of note is that the works discussed use a wide variety of mechanisms for determining an applications energy usage and are fare from homogenous: Some such as Choochotkaew et al. and Pathania et al. present a machine-learning approach to work around the difficulties in gathering hardware-level energy data [17][18]. On the other hand, many other works attempt to attribute energy consumption based on process-level data reported by the operating system: Husain Bohra and Chaudhary use a combination of CPU utilization/time, memory usage and disk I/O to arrive at an per-application energy attribution estimate that encompasses many common server components [19]. Along with [20], the authors perform this attribution in the specific context of VM power estimation, one of the key parts of the cloud-native scenario. Other works focus on individual hardware components (often CPUs in combination with RAPL) and attempt to provide highly accurate per-process energy consumption metrics [20]. Of particular interest is the work by Hè, Friedman, and Rekatsinas, which introduces the concept of energy credit to combat noisy-neighbour effects in cloud computing environments [21]. Other work uses a mixture of models: the `cwatts++` implementation by Phung, Lee, and Zomaya provides both an estimation model based solely on OS metrics and a RAPL-backed model [22].

Together, these works provide a rich foundation for building an attribution model that accounts for the complexities of a virtualized Kubernetes cluster. Applying their methodologies

to the discussed problem space will provide an answer RQ1, resulting in an accurate energy attribution model.

3.2. Implementations

Along with constructing an attribution model, this thesis also aims to present a reference implementation of said model, including benchmarks against existing implementations. To ensure that the implementation has the best chances at producing an accurate attribution value, its design should be informed by decisions made in existing implementations. While many of the previously referenced works come with a proof-of-concept implementation, there have also been some implementation-focused projects that aim to provide a practical, comprehensive energy attribution solution for the given scenario. This section discusses two in greater detail: Kepler and Scaphandre.

3.2.1. Kepler

Kepler, or Kubernetes-based Efficient Power Level Exporter is a project that aims to provide an easy-to-use energy and CO2 estimate for applications running in a Kubernetes Cluster. It provides fine-grained energy attribution data for Kubernetes applications and exports them through standard metrics interfaces such as Prometheus. The project is supported by the Cloud Native Computing Foundation and is being actively developed through its GitHub repository[1].

Kepler is of interest for answering both RQ1 and RQ3: First is its tight integration with Kubernetes. Kepler is not only designed to attribute energy consumption to applications inside a k8s cluster, but is also meant to be run from inside a cluster itself. By being a Kubernetes-aware application Kepler can provide insights into how to handle the complexities caused by running inside of a k8s cluster.

Second, Kepler is able to operate both on bare-hosts hardware and inside of virtual machines. To enable this, it splits its energy measurement model into three steps: Node total power (equivalent to external power), node component power (per-component power consumption), and pod power (attribution to k8s application pods). While Kepler always uses a ratio-based model for the final pod attribution, it can use either hardware interfaces (such as RAPL) or power estimation for its other two steps. The latter is intended for scenarios where no hardware metrics are available, such as in a VM. As can be seen in Figure 3.1, Kepler then uses a pre-trained power model to estimate energy consumption of the node. While Kepler provides tooling to train custom power models [24] it also comes with several models ready to go, making it an attractive comparison target for the final implementation. Also note the third "passthrough" mode shown in Figure 3.1: This proposed mode would use a passthrough mechanism between the host and VM to enable in-VM power attribution without the need for an estimation model. However, this mode is currently unimplemented [23].

[1] `https://github.com/sustainable-computing-io/kepler`, accessed 2025-01-17

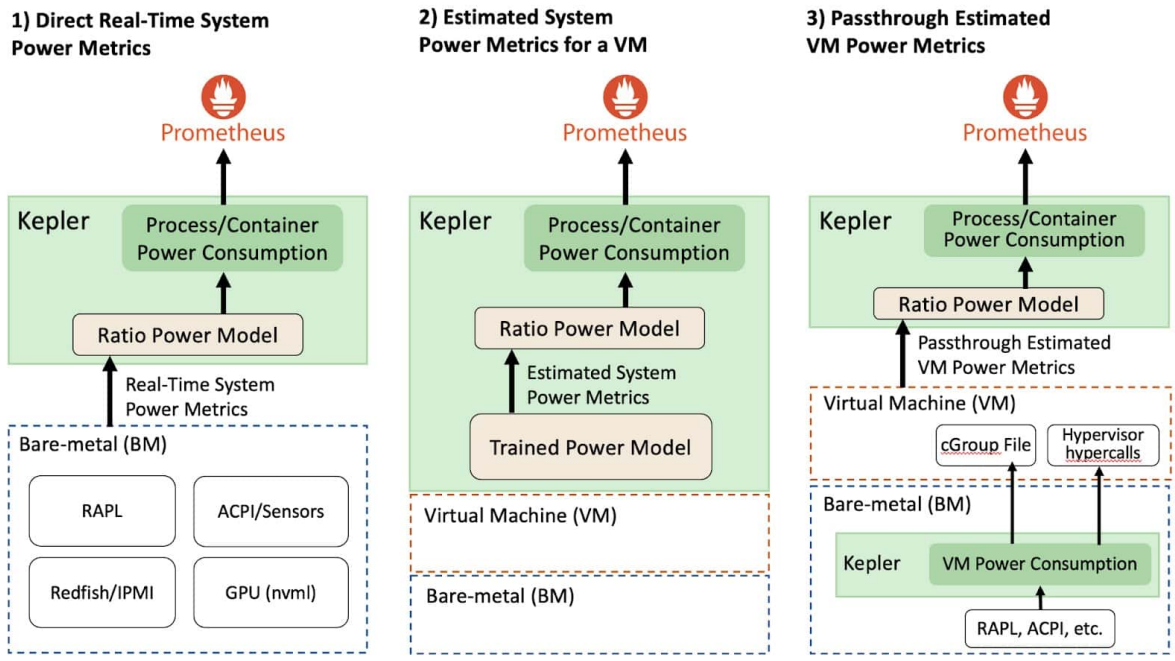


Figure 3.1.: Keplers different operating modes, including the proposed but unimplemented passthrough mode [23]

For its Ratio Power Model, Kepler makes use of hardware counters if available, while also using the eBPF framework in the Linux Kernel to collect per-process cpu scheduling events, CPU time and cache misses [25]. Combined with scheduling information from the Kubernetes API Kepler can then provide a per-Application/Namespace/Pod power reading.

3.2.2. Scaphandre

Scaphandre is another recent project that aims to be a widely compatible energy attribution solution. Unlike Kepler, Scaphandre is not Kubernetes-native, but instead aims to support a wide variety of platforms, including bare-metal, virtualized, and containerized applications. Scaphandre focuses on attribution based on hardware metrics (specifically RAPL), with its per-process energy attribution using a CPU-time-ratio approach [26]. While Scaphandre is not a Kubernetes-native application, it does have support for monitoring and attributing energy consumption to containers, an important baseline for Kubernetes support. It also provides a large number of data exporters for purposes of visualization, including JSON, Riemann and Prometheus.

Scaphandre is of particular interest RQ2 because of its claimed ability to implement VM passthrough, using a special QEMU exporter to create "emulated" RAPL directories to pass to VMs [27]. This is of particular interest as the implementation will need to somehow overcome the host-guest barrier as well. Scaphandres passthrough approach is one promising candidate for such a mechanism and will be discussed in more detail further on.

4. Attribution Model and Architecture

This chapter details the construction of the energy attribution model used throughout the rest of this thesis. The proposed model is rooted in the insights gathered about the problem space so far and inspired by existing implementations. It is able to track and attribute energy consumption across all the aforementioned layers in the virtualized k8s scenario. This model is both the answer to the first research question and serves as the architectural groundwork for RQ2. In total, the construction process involves three major steps: Host-side attribution, VM passthrough, and guest-side/k8s attribution (see also Figure 1.3). The challenges and complexities involved in each step in are discussed the following sections, following a bottom-up approach.

4.1. Challenges and Requirements

Beginning the model first requires a brief review of the challenges discussed in the preceding chapters. At its core, the model must be able to attribute the energy consumption of a physical machine to individual applications running inside Kubernetes cluster, which is itself running inside virtual machines. The primary challenges for the model will be to overcome the layers of abstraction present in this scenario while preserving attribution data. To help better visualize the attribution process, consider Figure 4.1, which shows a tree view of how energy consumption must be split up among the involved layers. At the bottom of the tree, the systems total energy consumption is first split between hardware components such as the CPU, DRAM and GPU (hardware attribution). For each component, consumed energy must then first be split between the processes running on the host operating system, including the Kernel, userspace processes and VM processes (host-level attribution). Note that each component to be monitored has an attribution tree of its own. Once the model has attributed energy consumption to individual VMs, it must then perform this same attribution again, this time inside the VM (guest-level attribution). Here, it must once again differentiate between kernel and userspace processes, filtering for the processes running in the application container. Finally, it can attribute the remaining energy consumption to the application process inside the container, completing the attribution process.

To ensure that the attribution model can deliver useful results in such a complex environment, a set of additional requirements must be met. A detailed discussion of these requirements follows below:

Precision The models attributions must be accurate enough to allow for useful measurements and conclusions to be drawn about energy consumption, even when operating across

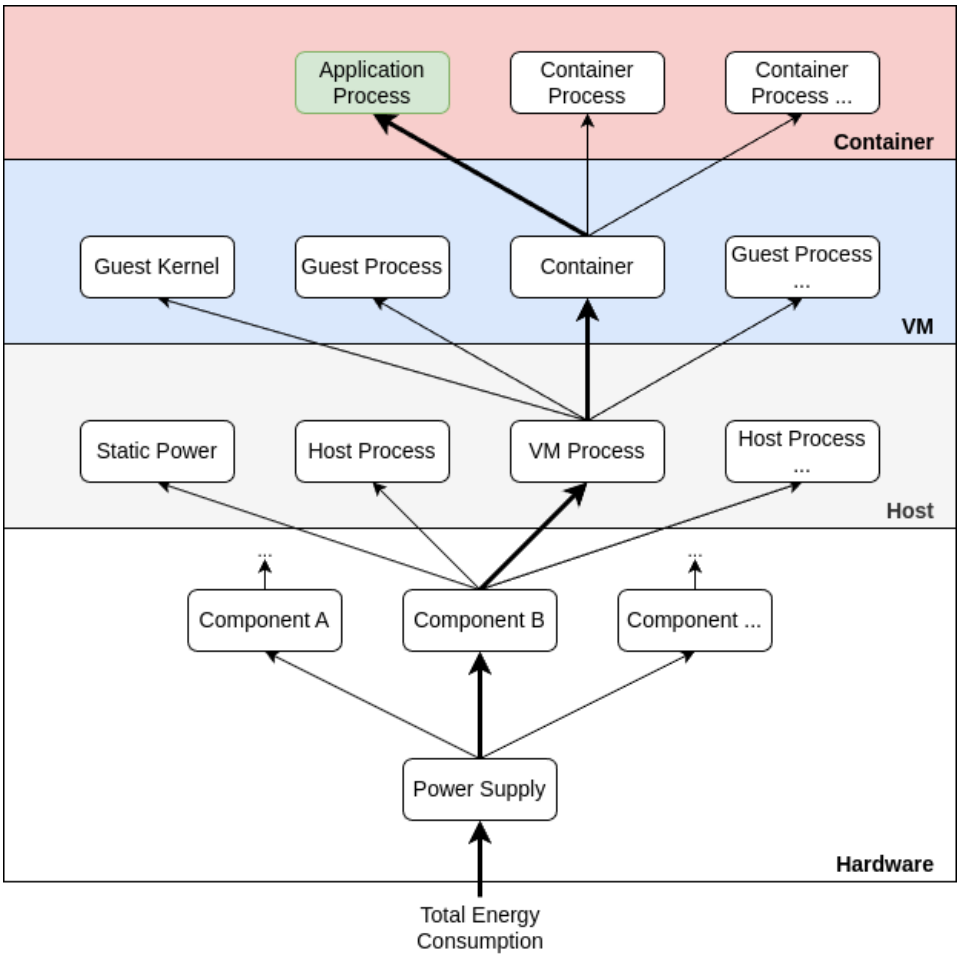


Figure 4.1.: Energy attribution tree with application attribution path highlighted (thick arrows). Each component has its own tree for the host level and above

all involved layers. Additionally, the accuracy of the values returned should be known.

Reliability The model must be able to provide filter out possible interference in the intermediary attribution steps to ensure accurate values are always returned.

Compatibility The model (but not a given implementation) must be generic enough as to be independent of any particular layer implementation.

4.1.1. Precision

The ultimate goal of this work is to enable developers to optimize their applications energy consumption. In doing so, high precision in the attribution results is desirable, as more precise values reduce the noise floor, allowing smaller optimizations to be measured. For the given scenario, attribution precision takes on two forms that we call absolute and relative. Absolute precision is the difference between real physical energy consumption and the attributed value for a given application. High absolute precision is useful for providing operators with a quantitative assessment of their applications environmental impact. Relative precision is the attribution precision when comparing multiple applications at the same time. High relative precision means that the model values are highly proportional to actual energy consumption, allowing developers to compare different applications/implementations running on the same system.

Currently, many existing solutions use estimation-based energy consumption modeling, which runs counter to the precision requirement. For example, the aforementioned Kepler project comes with two different modes, depending on the availability of hardware energy data. If no such data is available (like inside a virtual machine), it uses a machine-learning trained estimation model to provide energy readings. The precision of such models is much harder to assess than for hardware-backed metrics, as it depends on the quality of the training data, similarity to the particular system under test, and the method used for training the model. Ultimately, estimation models are prone to being lower precision than hardware-based models for this reason. Kepler in particular tries to combat this lack of precision through machine-specific model training: The project provides a pipeline that can be used to train machine-specific models, known as the Kepler Model Server [24]. While this has the potential to increase precision, it also comes with significant drawbacks, such as needing a bare-metal host of the system under test and the costs associated with training.

On the other hand, hardware metrics have the potential for very high precision attribution, but suffer from their own set of issues. For one, hardware interfaces such as RAPL are known to sometimes under-report component consumption [28][29] and non-instrumented components such as power supply losses will go unreported and thus unattributed. However, this error should be well-characterized and can thus be accounted for by an implementation - it should thus not affect the usefulness of the attribution data.

4.1.2. Reliability

Being intended for a cloud-heavy usage scenario, the model will inevitably have to operate in noisy environments - that is, systems that are shared by many customers and applications. While precision is key for providing accurate attribution data, for this data to be truly useful it must also be resistant to common sources of noise. If we consider only a single application, we may consider other applications and VMs as sources of noise, to be filtered and ignored in order to achieve accurate attribution. Doing so requires knowledge of these other processes however. Ultimately, to eliminate this noise the model must adopt a holistic approach: By looking at all possible energy sinks (processes, VMs) and attributing their consumption, it can identify, classify and practically "eliminate" this source of noise, instead making it part of the attribution values. Doing so allows the model to deliver reliable and repeatable attribution results, independent of other loads on the system.

4.1.3. Compatibility

In attempting to provide energy attribution for such a complex scenario, the model will inevitably have to work with different layer implementations: Instead of RAPL, energy measurements may be provided by a different interface such as through IPMI or a hardware meter. The virtualization host may run Linux with libvirt VMs or it may be using Proxmox with direct QEMU VMs - or the host may even be a Windows machine instead. On the k8s layer, the model may find a standard distribution like a k3s, or there may be a separate control plane operated by a cloud provider with minimal worker nodes. In short, there are a huge number of possible approaches to arrive at the given scenario.

For the attribution model, this means not relying on any implementation detail. It should be generic enough so that it can be adapted to many possible layer implementation combinations, while still enabling a good degree of accuracy. To keep the scope of this thesis manageable, the implementation presented in later chapters will be restricted to one common scenario, but still remain modular enough to be extensible in the future.

In addition to supporting different platforms, compatibility also refers to supporting different applications. Ideally, energy attribution should be done in the background, without requiring any steps from the application operator. Therefore, any approaches that rely on directly instrumenting applications (such as using Linux' `perf`) will be avoided.

4.2. Attribution: Hardware and Host Layer

There are a number of ways to construct a model from the previously defined requirements: This thesis follows a bottom-up approach. Thus, the first step involves establishing a good baseline for energy attribution at the hardware level. Attribution layers are then added individually, along with a discussion about the complexities they bring with them. The final result will be a full-stack attribution model capable of operating in the previously described scenario.

At the basis of the model lies the host layer, whose primary job is attribution of energy consumption to individual tasks and processes running on a given system. The hosts energy consumption provides the primary data source for this task. For this, consider a simple server system running an operating system with multiple processes. We can define the energy consumption E of this system as

$$E = \sum_{c \in C} E^c \tag{4.1}$$

where c is a hardware component of the system (such as a CPU, RAM or Storage) and C is the set of all components in the system. Each component may have different energy usage patterns and must thus be treated separately for this analysis. The energy consumed by each component can be split into two parts, Static and Dynamic:

$$E^c = E^c_S + E^c_D \tag{4.2}$$

Here, static energy consumption refers to the energy consumed by a component irrespective of any load applied, while dynamic energy refers to energy consumption caused by an applied load, such as a running process on a CPU. E^c_S thus represents the load-independent consumption of a system and can be treated as a constant. The value of E^c_S must be determined for any given system, for example by measuring total energy consumption E^c on an idle system with no processes running (so $E^c_D \approx 0$) [21]. Since static energy consumption is load-independent, it cannot be reasonably attributed to a single process and must therefore be handled separately.

The ratio between E^c_S and E^c_D can vary greatly depending on the component c, as can the difference between typical and maximum $max(E^c_D)$. For example, modern CPUs and GPUs feature vew low E^c_S thanks to power-saving mechanisms (as low as a few Watts) while also consuming hundreds of Watts in E^c_D during high load [30]. Since only E^c_D is attributable to individual processes, components where $max(E^c_D) \gg E^c_S$ are of special interest and will be discussed in detail later. This section will focus on CPUs, DRAM and storage specifically. Meanwhile, many other components either have $max(E^c_D) \leq E^c_S$ (meaning that they are dominated by static energy consumption), and/or have values of $max(E^c)$ that are insignificant compared to the aforementioned components. Examples for such components can include basic platform components such as the mainboard itself, peripherals, management interfaces (IMPI), etc. For the purposes of the attribution model, these components can be treated as purely static.

With per-component energy/power now defined, the next step is attribution of dynamic component energy consumption to individual processes. This requires splitting consumed component energy E^c_D into energy consumed by processes E^c_p

$$E^c_D = \sum_{p \in P} E^c_p \tag{4.3}$$

where P is the set of all processes p utilizing the component c. For this first attribution step there is no differentiation between process types - host-level processes and VM-related tasks are treated the same. This allows the model to be agnostic about the type of higher-level processes running on a system, increasing flexibility.

To determine accurate values for E_p^c, information about the per-process utilization of each component is needed. We can define a scalar $0 \leq C_p^c \leq 1$ as the fraction of a components c energy usage that should be attributed to the process p. Effectively, this is the processes "energy **c**redit" for the given component c [21]. This lets us write E_p^c as

$$E_p^c = E_D^c * C_p^c, \textit{ where } 0 \leq C_p^c \leq 1 \textit{ and } \sum_{p \in P} C_p^c = 1 \tag{4.4}$$

Ensuring that C_p^c is both proportional and accurate to actual energy usage presents a significant challenge. Doing so requires establishing a mapping between a processes component utilization and the resulting energy consumption. While accurate values can in theory be gathered using extensive process and component profiling, this is often too expensive and/or complex to implement for every process and component running on a system. Instead, automatically reported utilization values (such as those provided by the OS) can be used as proxy values. However the accuracy of these values is unknown and requires validation, with examples for key components being discussed below.

4.2.1. Dynamic Energy Attribution for CPUs

CPUs have traditionally been the biggest consumer of electricity for a given computing system. While the advent of GPU computing is now challenging this status quo, a majority of the previous work regarding energy attribution has focused on CPUs, resulting in a large number of approaches and possible proxy values. At the same time, CPU utilization instrumentation is extensive and important metrics are available on all major operating systems. Therefore, CPUs are a natural first step for defining and implementing an attribution model.

To find E^{CPU}, one may either use hardware measurement (for example by measuring current through the ATX12V power supply rail) or software counters provided by RAPL-like interfaces on server systems. Both modern Intel and AMD systems provide a RAPL CPU domain with an energy consumption statistic [7], which provides good accuracy and excellent proportionality to actual CPU energy consumption [29]. Using this metric, it is possible to estimate E_S^{CPU} by measuring energy consumption of the system in its natural idle state, with as few processes running as is realistic for its use case. E_D^{CPU} can then be calculated via $E^{CPU} - E_S^{CPU}$.

To find satisfactory C_p^{CPU} for all processes a model implementation must either use OS-provided statistics, or instrument processes itself to retrieve metrics using tools such as `perf`. This latter option is undesirable however, as it would require instrumenting all processes individually, hindering portability and compatibility. Therefore, the most practical approach for constructing C_p^{CPU} for all processes is to make use of values reported by the OS. The most obvious OS metric for measuring CPU utilization is the OS-reported CPU usage U^{CPU}. On

Linux, tools such as `top` report both a total CPU utilization U^{CPU} and a per-process utilization U_p^{CPU} [31]. Using these metrics, we can construct a simple per-process attribution model as

$$C_p^{CPU} = U_p^{CPU} / U^{CPU} \tag{4.5}$$

Note that the divisor is the total CPU *utilization* (U^{CPU}), not *capacity* ($max(U^{CPU})$), thus excluding idle time. This results in C_p^{CPU} being a more accurate representation of a processes contribution to energy usage. C_p^{CPU} is equivalent to the "energy credit" concept proposed in [21].

The utilization values reported by tools such as `top` are based on scheduled CPU time reported in the `/proc` filesystem, measured in jiffies[1]. Information about total per-core CPU time can be retrieved from `/proc/stat` as the sum of the `user + nice + system` values Per-process times can be read from the process-specific stats at `/proc/<pid>/stat` as the sum of the `utime + stime` values [32]. These values act as accumulators that increase over time, so to determine the fraction of CPU time in a given interval $[t_{-1}, t]$, we can use

$$C_p^{CPU} = \left[\frac{utime + stime}{user + nice + system}\right]_t - \left[\frac{utime + stime}{user + nice + system}\right]_{t-1} \tag{4.6}$$

Alternatively, CPU process scheduling information can also be gathered using the ePBF framework included in modern Linux kernel versions, which can report CPU scheduling events along with associated process IDs. This method is used by the Kepler project in some circumstances as it promises higher precision than the `/proc`-based approach [25]. Since the `/proc` variant has the advantage of being far simpler and not requiring as many privileges (while still providing high accuracy), the eBPF approach will not be discussed here.

Using CPU time in this fashion makes the implicit assumption that all CPU time is equal with regards to energy consumption. This is likely not the case in practice, as different CPU operations (computation, memory/IO access) are going to incur varying energy costs due different implementations in the silicon [33]. For example, cache misses are likely to incur a high penalty and thus energy cost since they require accessing memory and can stall other operations. Additionally, different CPU instruction access patterns (integer vs. floating-point vs. logic ops vs. AVX) will also result in varying power demands. Linux does provide kernel-level metrics for events such as cache misses through `eBPF`, making the creation of more precise attribution proxy values possible.

Another alternative is to assume that all processes on a system will have similar access patterns if viewed over a long enough time period. By sampling over a large interval Δt, the usage pattern for a given process should then even out. This approach is likely to produce less accurate results than `eBPF` counters, but should still be able to provide sufficient precision for general attribution, provided that none of the processes have highly unusual or pathologic usage patterns.

[1] a kernel-internal unit which has a system-specific but known conversion rate to seconds [32]

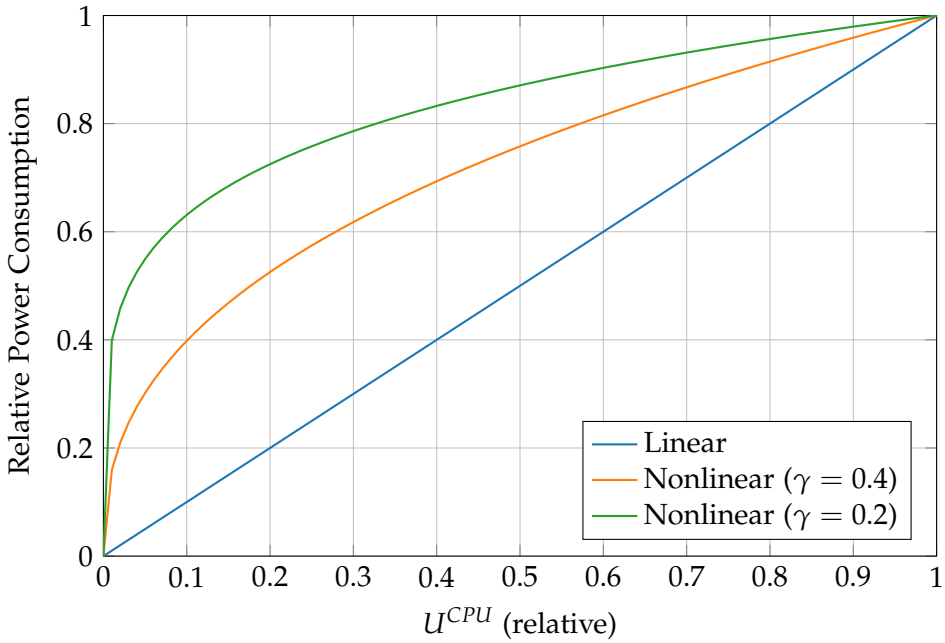


Figure 4.2.: Relation between CPU utilization and power draw. Real CPUs typically follow a nonlinear relation approximated as $f(x) = x^{\gamma}$ [21] .

Even with this simplification, CPU utilization is not necessarily equivalent to power consumption. This is because on many modern CPUs, the relation between utilization and power draw does not follow a linear curve [34]. Instead, many CPUs have a power curve that flattens out with increasing utilization, due to the CPU hitting thermal and power envelope limits. We can approximate this curve with the function $f(x) = x^{\gamma}$, where $0 \leq \gamma \leq 1$ is a CPU-specific constant [21]. Figure 4.2 shows select utilization vs. power curves with both linear and nonlinear behavior.

For a single-process attribution scenario, it is possible to compensate for this behavior by applying γ as a correction factor to the calculated utilization [21]:

$$C_p^{CPU_{corrected}} = \left[C_p^{CPU}\right]^{\gamma} \tag{4.7}$$

For multi-process scenarios, correction is more challenging and the above correction fails with $\sum_{p \in P} C_p^{CPU} > 1$. This can be remedied by "shifting"/"stacking" of processes along the utilization axis, moving subsequently evaluated process further to the right (and into flatter areas) on the curve. This results in the following correction function

$$f(x, p) = \left[x + x_{prev}(p)\right]^{\gamma} - y_{prev}(p) \tag{4.8}$$

where the offsets x_{prev} and y_{prev} are the summed uncorrected and corrected attributions of all processes previously evaluated in the given measurement interval

$$x_{prev}(p) \sum_{j=1}^{p} C_j^{CPU}$$

$$y_{prev}(p) \sum_{j=1}^{p} C_j^{CPU_{corrected}}$$

This shifts the correction curve origin appropriately for each process, using the previously calculated process attribution values to do so. It also ensures that the total energy credit C^{CPU} remains 1. For example, consider three processes with attribution/credit of $C_A^{CPU} = 0.3, C_B^{CPU} = 0.5, C_C^{CPU} = 0.2$ and a nonlinear factor of $\gamma = 0.4$. During the calculation of $C_p^{CPU_{corrected}}$, processes B and C are moved to the right of the range occupied by the preceding processes. The results are shown in below and visually in Figure 4.3.

$$C_A^{CPU_{corrected}} = (0.3)^{0.4} = 0.618$$
$$C_B^{CPU_{corrected}} = [(0.5 + 0.3)]^{0.4} - 0.618 = 0.297$$
$$C_C^{CPU_{corrected}} = [(0.2 + 0.8)]^{0.4} - 0.915 = 0.085$$

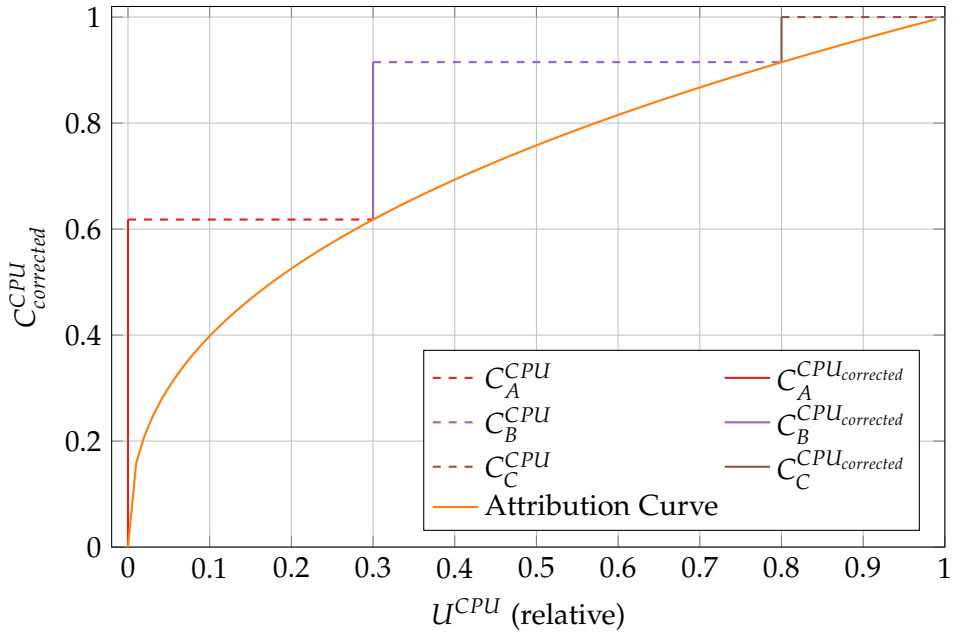


Figure 4.3.: Non-linear power correction with multiple processes, assuming $\gamma = 0.4$

Using this correction approach has a profound impact on the construction of the attribution model: By basing attribution values on the results of previously evaluated processes, individual results are no longer independent. Instead, the order in which processes are

attributed now matters greatly for the final attribution result. Consider process A from the above example. If it is evaluated last instead of first, we find that $C_A^{CPU_{corrected}} = 0.133$, less than a quarter of the original value. The huge influence of attribution order means that for this compensation to be effective and accurate, a process sorting algorithm would be required. This algorithm would need to deterministically map processes to ranges of the nonlinear power band, but there is no obvious correlation that could serve as a metric for this task. Total process runtime may be an acceptable proxy value, with long-lived "base load" processes being evaluated first and short-lived "burst" processes last. This would lead to the "base load" processes receiving a larger share of dynamic power, representative of their longevity. Alternatively, uncorrected attribution C_p^{CPU} could also be used, with the heaviest processes being evaluated first. This would result in disproportionally higher attribution values for large processes, highlighting their heavy load on the system. While both of these metrics have technical weaknesses (such as process restarts resetting the runtime), the main problem is the lack of justification they provide for the position of a process on the curve. In other words: Why should two identical processes have different energy attributions when they are indistinguishable? What decides which of them is responsible for 75% of CPU energy consumption and which only used 25% (assuming $C_p^{CPU} = 0.5$ and $\gamma = 0.4$)?

While it is possible that an intuitive metric for this algorithm can be found (perhaps as a heuristic), the large uncertainty and currently unclear motivation threaten the models requirements of reproducibility and reliability. Therefore, the remainder of this thesis will assume linear curves for all components. The model assumes that there is no clear hierarchy between processes on the power curve and instead treats all processes equally, attributing 50% of consumed energy to each process in the above case.

4.2.2. Dynamic Energy Attribution for DRAM

The power consumption behavior of DRAM is complex and influenced by many different factors in hardware and software. To come up with values for E^{DRAM}, E_S^{DRAM}, and E_D^{DRAM}, one may either use hardware measurement, or make use of the RAPL DRAM domain available on some modern Intel and AMD platforms (which has been shown to provide accurate readouts of consumption data [28]).

To find values for C_p^{DRAM}, a suitable proxy value from the OS-reported statistics is required. In reality, a processes DRAM energy consumption is influenced by its total memory usage, access patterns such as read/write locality and access frequency. However, only memory usage is available as an easily accessible metric on modern Linux kernels. Since a larger amount of used memory results in more DRAM cells that need to be maintained (and thus consume energy), one may assume that memory usage is proportional to energy usage. This does not account for differences in memory access patterns, which may negatively affect the accuracy of this proxy value. For example, a process that frequently and randomly access a relatively small region of memory may consume a disproportionally large amount of energy. On the other hand, a process that stores a large amount of data in memory through a single allocation but rarely accesses it may consume less energy than otherwise predicted.

To calculate C_p^{DRAM}, the same approach as with CPU utilization is used, that is the fraction of memory occupied by a single process divided by total memory usage:

$$C_p^{DRAM} = mem_p / \sum_{p \in P} mem_p \tag{4.9}$$

On Linux, the actual memory usage can again be read from the stats reported in `/proc`, specifically `/proc/<pid>/status` for individual processes and `/proc/meminfo` for the entire system. The physically allocated memory for each process is given as the sum of `RssAnon+RssFile`. `RssShmen` indicates shared memory between multiple processes and is ignored in the model due to unclear process ownership and thus attribution [21]. Total memory usage U^{DRAM} then becomes

$$U^{DRAM} = MemUsed - Buffers - Cached \tag{4.10}$$

where

$$MemUsed = MemTotal - MemFree \tag{4.11}$$

Once again, buffers and caches are not directly attributable to an individual process and are instead handley by the OS, thus they are ignored. This may result in the model providing a value of $\sum_{p \in P} C_p^c < 1$ in Equation 4.4, resulting in some energy usage remaining unattributed.

4.2.3. Dynamic Energy Attribution for Storage

Another significant consumer of energy in a server system is persistent storage, such as HDDs and SSDs. Measuring and subsequently attributing their energy consumption $E_{(S/D)}^{Disk}$ presents a larger challenge, as OS-reported metrics (such as via RAPL) are far less ubiquitous. As disks often have their own connection to the systems PSU, such metrics would rely on disk-reported statistics, which are not widely available. This leaves the option of external measurement, for example by monitoring current of disk power connectors.

Assuming a energy consumption metric is available, attribution still requires a source of C_p^{Disk} values for all processes. Disk energy consumption varies with technology (HDD vs. SSD) and workload (random access vs. sequential, read vs. write): SSDs are able to handle random workloads with much lower energy consumption, mainly due to their ability to return to a sleep state more quickly (race-to-idle), while HDDs had a higher, but consistent energy consumption [35]. Ideally a "disk" activity metric could be present that accounts for this complexity. This would include the amount of data written to each type of disk, the number of IO operations performed and timing information.

In practice, such data is seldom available. For example, the Linux kernel only reports the total amount of data read/written per process without separating by disk in `/proc/<pid>/io`. We can still construct a proxy value for C_p^{Disk} form this by comparing the per-process value with the total number of sectors read/written per disk from `/proc/diskstats` [36].

$$C_p^{Disk} = IO_p / IO_{Total} \tag{4.12}$$

$$IO_p = write\_bytes + read\_bytes \tag{4.13}$$

$$IO_{Total} = \sum_{d \in Disks} \frac{sector^d_{writes} + sector^d_{reads}}{sector^d_{size}} \tag{4.14}$$

While these values do account for OS-level caching, disk-side caches or queuing may further impact the accuracy of energy attribution. While this metric is unlikely to be precise enough for fine-grained attribution where many processes perform an equal amount of writes, it should be sufficient to isolate particularly IO-energy-heavy processes from a limited set.

4.2.4. Dealing with Static Energy Consumption

As discussed previously, the static energy consumption of a system E_S may be considered a constant for the purposes of attribution. However that does not mean that it should be ignored entirely - static energy can be a significant fraction of a systems total energy consumption and thus environmental impact. While it is not possible to attribute static power to single processes due it being load-invariant, there are a few solutions for still including it in attribution:

1. Separate it: Since static energy consumption is not caused by a single process, one may chose to group it into an "other" category, for analysis by other energy optimization initiatives.

2. Even attribution: Static energy consumption is divided equally along all applications running on a system

3. Proportional attribution: Static energy consumption is divided along applications according to the applications fraction of dynamic energy usage.

1. is the simplest approach, but obscures a potentially significant fraction of real energy usage from users. This may reduce the inventive for users to lower their energy consumption by, for example, reducing the number of systems used for operating their application. 2. and 3. address this by including static consumption to highlight the cost of just deploying the application - these approaches are also recommended by the Global Sustainability Initiatives Protocols [37]. The third approach is to be preferred over 2, as it is less prone to over-estimate energy consumption for otherwise low-power applications that only consume a small portion of dynamic energy. This could also motivate tighter packing of applications onto systems, thus reducing the overhead and increasing system utilization.

4.2.5. Summary

The findings from this section can be summarized as a few key steps needed to implement an attribution model on the host layer:

1. For each component c, determine the static energy consumption E^c_S of that component with no load applied. This needs to be done only once per machine configuration.

2. For each component, retrieve the total energy consumption E^c and calculate dynamic energy consumption with Equation 4.2.
3. For each process running on a component, retrieve or calculate its attribution factor C_p^c based on existing metrics. How this factor is determined depends on the component c.
4. For each process, calculate its attributed component energy E_p^c using Equation 4.4.
5. Repeat steps 2 through 4 for all components.
6. Calculate the final per-process attribution E_p by summing the per-component attribution values
7. Optional: Attribute static energy by splitting it across all processes, either equally or proportional to E_p.

4.3. Attribution: Virtual Machines

The previously discussed steps provide the means for attributing energy consumption to processes on a single host, but this is only the first step for attribution in a virtualized k8s cluster. In the given scenario the physical host machine is a hypervisor running several virtual machines. To allow energy attribution inside of these virtual machines, the model must first attribute power to individual VMs by calculating $E_D^c \forall c \in C$. This requires an understanding of how resource/component allocations for VMs are represented on the host. An implementation must then make these values available inside the VM to allow each guest to continue with fine-grained guest-level attribution.

The first step in this process it the identification of host metrics that accurately represent VM resource utilization. Depending on the hypervisor and hypervisor architecture in use, these metrics may take different forms and require different steps to acquire. The main target of this thesis will be Linux-based VMs using QEMU (with KVM for hardware acceleration), but the considerations below will be applicable to other hypervisors as well.

For QEMU-based virtualization, acquiring values for E_D^c is thankfully straightforward: A QEMU VM is represented by a standard Unix process with values for CPU, Memory and I/O usage, just like a standard host process. This process encompasses all the resources utilized by a VM; they can therefore be used for energy attribution to the VM using the principles described previously in section 4.2. In other words, the attribution values for the *VM process* are equivalent to the attribution values for the *VM itself*.

The next task is to make this attribution available inside of the VM guests. Doing so requires passing data between the host and the guest, thus overcoming the isolation barrier introduced by virtualization. This is not a trivial task however, as this barrier presents both a security and operational boundary; a VM being able to break through this barrier presents a significant security risk. For this reason, caution must be taken to preserve as much isolation as possible when selecting the passthrough method. Unlike the basic attribution model this task is mainly

a technical challenge and possible solutions will be implementation-dependent. One can group possible passthrough approaches into several categories by how they deal with this barrier:

The **first** option is to sidestep the barrier altogether through an external service. For example, the host could make attribution data available over the network or even the internet, either directly (say, via a file share) or by submitting it to an external service. Guests could then access this service to retrieve their attribution data. The main appeal of this approach is that it leaves the isolation barrier intact, lessening the security burden on the implementation. This approach also makes no assumptions about the virtualization solution in place and requires no modifications to it, further supporting the goal of compatibility. However, there are also major downsides to this approach: While it leaves the existing barrier unaffected, the service introduces a *new* security boundary that needs to be protected adequately, i.e. with access control measures. Using a networked service also requires that the host and guest are able to communicate with each other, but this may not be the case. For example, the host may only have access to an isolated management interface to prevent external attacks - extra steps would thus be required to publish attribution data in this scenario. In fact, such a solution is likely to have high overhead regardless: Using a networked service results in added latency (generally a least in the milliseconds), not to mention additional energy consumption due to having to host the service. Finally, this approach requires that the guest-side attribution is aware of this service, requiring additional cooperation between the infrastructure operator and VM users to enable service discovery.

The **second** option is to integrate energy attribution *into* the barrier, making the passthrough part of the usual virtualization mechanisms. Hypervisors are already able to emulate hardware such as storage, networking or disc drives - so why not hardware energy interfaces too? This would involve adding virtual hardware for energy interfaces such as RAPL, which the hypervisor would then emulate and populate with attribution data. From the guests perspective, this emulation would look like a regular hardware energy interface, requiring no modifications. Another option would be the use of hypercalls, which allows guests to communicate with the hypervisor directly through a well-defined interface [38]. By implementing an attribution hypercall and adding support for it in major OS kernels, a transparent (for guest userspace) passthrough could be achieved. This transparency is the major advantage of this approach - however it does come with serious costs: For one, adding hardware emulation to hypervisors is a non-trivial task and would likely require addition and/or modification of safety-critical code. Bugs in the emulation implementation could potentially become security breaches, making this a high-stakes operation. Such an implementation would also likely be non-portable, forcing an implementation to target specific hypervisors.

The **third** option presents a middle ground between the first and second: A cooperative passthrough through a shared, local channel between host and guest. In this approach, the host pushes the attribution data into a channel, whose receiving end is then made available

to the guest, for example through shared memory or an emulated serial interface. Unlike the second approach, this reuses an existing mechanism for host-guest communication, lessening the security implications. While the exact mechanism may still be hypervisor-dependent, an implementation would only have to hook up its attribution output to this channel input, without the need to re-implement the communication mechanism itself. Such an approach would not be transparent to the guest and would require some additional configuration similar to the first approach. Latency and overhead are likely to be superior to a networked solution however, and the local approach eliminates the need for a shared network.

Which of these approaches is best suited for a particular implementation depends on the environment in which it will operate. For example, many cloud providers already have special networked services available to their VMs for the purpose of access management [39]. In this scenario, another networked service for energy attribution might be a sensible choice since it can reuse existing infrastructure patterns. Looking at previous work we find that Kepler discusses using the second approach for its future passthrough mode [23]. The third approach is already used by several implementations, using mechanisms such as a virtio serial interface [20] or a `virtiofs`-based shared filesystem [27].

4.4. Attribution: Guest and Kubernetes

The final stage of the attribution model covers the guest-side attribution, that is, processes running on the guest OS inside the virtual machine, including Kubernetes workloads. The basic attribution methodology is the same as for the host, discussed in section 4.2. This requires calculating per-process utilization for all relevant processes and components, just like on the host. The difference to host-level attribution lies in the source and nature of energy consumption data. First, the guest-level attribution of course uses the passed-through attribution data as the basic energy consumption source instead of hardware sensors. Unless this passthrough is transparent to the guest userspace, an implementation must thus be able to communicate over the chosen mechanism to retrieve the VMs energy consumption. Since static power calculation (and compensation) is performed on the host, the passed-through data represents dynamic energy consumption only. The guest attribution model can therefore skip steps 1 and 7 of the host-level model (see subsection 4.2.5).

While the guest-level attribution process itself is similar to the host level, the nature of containerization and Kubernetes workloads means that there are additional challenges to be considered.

There are many ways to define a Kubernetes workload resource (such as a `Deployment`, `StatefulSet` or `Job` resource), however all of these resources ultimately create one or more `Pods`. A pod is a set of coupled containers that are deployed as a unit. When a pod resource is created, the Kubernetes control plane schedules the pod to run on an available node. Said node then starts the container processes defined in the pod and manages their lifecycle from there. From an OS-level view (outside of the k8s cluster), a pod is simply a

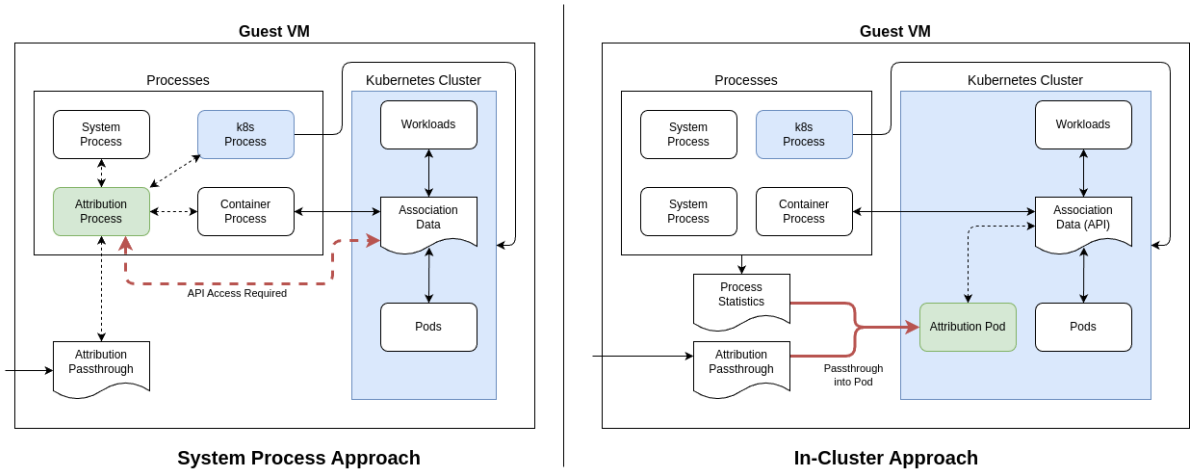


Figure 4.4.: Possible guest-level attribution approaches

number of containerized processes with isolation applied to them. Therefore, if the guest-level implementation runs on the k8s node as a normal system process, it will be able to see these processes and perform energy attribution using the previously described methods. However, the ultimate goal is to perform attribution on a Kubernetes *workload*, not a single container process or even a pod. While attribution of the underlying processes is of course required to do so, an implementation must also be able to link these processes to their Kubernetes pods, and thus workloads. Starting at the container process level effectively requires working backwards, going from container to pod, then pod to workload. The information that ties these elements together does exist, but lives primarily inside the k8s cluster. Therefore, if the implementation operates as a normal OS-level daemon it must somehow be provided with access to the Kubernetes API so that it can associate processes with the k8s resources that created them. This approach is shown in Figure 4.4 on the left - the implementation is able to directly gather the information required for attribution, but not workload association, with the problematic issue of API access shown in red.

One solution to this issue is to move the attribution task *inside* the k8s cluster, turning it into yet another workload. this is because pods have access to a cluster-internal network with a well-known API server location, allowing pods to perform queries against the cluster [40]. This gives the implementation direct access to the information needed to associate processes with k8s resources. However, moving into a container (with isolation enabled by default) also causes it to lose access to the process and utilization information needed for attribution, as well as the passed-through attribution data, as this information is only available to non-isolated processes. Unlike with the difficult-to-penetrate virtualization barrier however, many of the container isolation mechanisms can be selectively disabled as needed. For example, say the implementation uses the `/proc` filesystem to retrieve process statistics. One can simply pass the guests `/proc` directory to the attribution pod, giving the guest a full view of all running processes regardless of whether they are in a container or not, including

other pods and non-k8s processes. Alternatively one can use process capabilities to grant the attribution pod isolation-breaking features such as the ability to run eBPF programs in the OS kernel to collect process statistics. See the right side of Figure 4.4 for a visual representation of this approach: Since the implementation is inside the cluster it has easy access to association data but requires a passthrough of attribution data. Using these mechanisms all needed OS data can be passed into the attribution pod, including the VM passthrough data, process statistics, and utilization information. Ultimately, this means that implementations of the guest model should be designed as a k8s-native solution, rather than a "bare" process.

5. Implementation

This chapter details an implementation of the attribution architecture from the previous chapter, as well as the setup used to run benchmarks discussed in chapter 6. This implementation is designed as a proof-of concept for the attribution model, allowing it to be benchmarked against existing solutions. The implementation is vertically complete and addresses all significant aspects of the model, while remaining extensible for more horizontal coverage. To improve reusability and iteration speed, it extends several existing solutions and combines them to mirror the previously described architecture. The following chapter discusses the details of the implementation, which will in turn form the basis for measurements in the following chapter.

The process of implementing the attribution model begins at the hardware level, specifically in the choice of hardware to target. For the purposes of this thesis the target platform is a three-node hypervisor cluster with specifications shown in Table 5.1, featuring third-generation AMD EPYC CPUs and running Proxmox, a KVM-based virtualization solution built on Debian. Since the target platform is a standard x86_64 system with a Linux-based OS, the implementation promises to be functional across a wide range of commonly deployed systems.

Node Count	3
Platform	Supermicro AS-1114S-WN10RT
CPU	AMD EPYC 7773X (64c/128t)
DRAM	1TB (16x 64GB) DDR4 RDIMM, 2933M MT/s
Storage	4x 6.4TB KIOXIA CM7-V PCIe 5.0 (Ceph)
OS	Proxmox VE

Table 5.1.: Implementation target host specifications

First, the VM attribution solution was installed on the hypervisor hosts, providing energy attribution data for each VM. Next, a virtualized k8s cluster with multiple workers was created, with each Kubernetes node running in its own VM. These node VMs were then provided with attribution data trough a passthrough mechanism to allow guest-level attribution. From there, the use of existing tooling inside the node VMs to attribute energy consumption becomes possible, completing the attribution model and implementation.

5.1. Host Layer and VM Attribution

The host-level implementation uses the data exposed by the Proxmox/Debian OS stack to perform VM attribution as described in the model. This requires the host component to:

1. monitor hardware energy counters (IMPI, RAPL)
2. enumerate VMs running on the host and detect their resource utilization
3. implement the per-resource model to attribute energy consumption
4. make the attribution data available to VM guests using a passthrough method
5. export host-level statistics and attribution data for external monitoring and benchmarking

The target platform exposes power metrics through RAPL, along with an IPMI-based total system power reading. Unfortunately, the RAPL implementation of this particular platform only provides energy readings for the CPU package and the core domain, with no separate measurement of other components, such as DRAM. Due to this lack of data, it was decided to limit the implementation to CPU usage metrics for the time being, though with the aim to keep open the option of future expansion.

Since the focus of this thesis lies on the attribution model (item 3) and its complete vertical integration (item 4), solutions to the other requirements were not reimplemented from scratch where possible. Instead, it was decided to use an existing energy attribution solution as the basis, allowing the reuse of its data acquisition (item 1 and 2) and export facilities (item 5), while reimplementing the attribution logic that is core to the model. Not only does this allow the use of already tested components, reducing the risk of systemic errors, it also provides more insight into existing tooling, helping with answering RQ3 from a qualitative standpoint.

While many of the tools discussed in chapter 3 are able to cover some of the required items, Scaphandre and Kepler aim to provide a comprehensive solution all the way from sampling to export. Table 5.2 provides a detailed evaluation of several tools and their capabilities with regards to the needs of this thesis.

	Scaphandre	Kepler	EnergAt
Energy Sources	RAPL	RAPL, IPMI (Redfish)	RAPL
VM Detection	Yes (QEMU)	No	No
Utilization Measurement	`/proc`-based	eBPF + OS metrics	`perf`-based
VM Passthrough	Partial, Exporter	No	No
Metrics Export	JSON, Prometheus, ...	Prometheus	JSON Files

Table 5.2.: Comparison of host-level attribution implementations

It was ultimately decided to use Scaphandre as the host-level implementation basis for the following reasons: First, it features a modular internal architecture which allows for easy

replacement of individual components, such as the attribution model. Second, it already features support for VM enumeration (item 2) and data export (item 5), as well as partial support for hardware counters (RAPL only, item 1) and VM passthrough (item 4). The latter is of particular interest: Scaphandre includes a data exporter that creates an emulated `powercap` RAPL directory tree for each detected VM. By passing this directory into the respective VM it is possible to reuse existing tooling designed for RAPL inside the VM guests with minimal adjustments. This avoids having to define a custom interface for guest-level power readings and improves interoperability.

5.1.1. Scaphandre Implementation

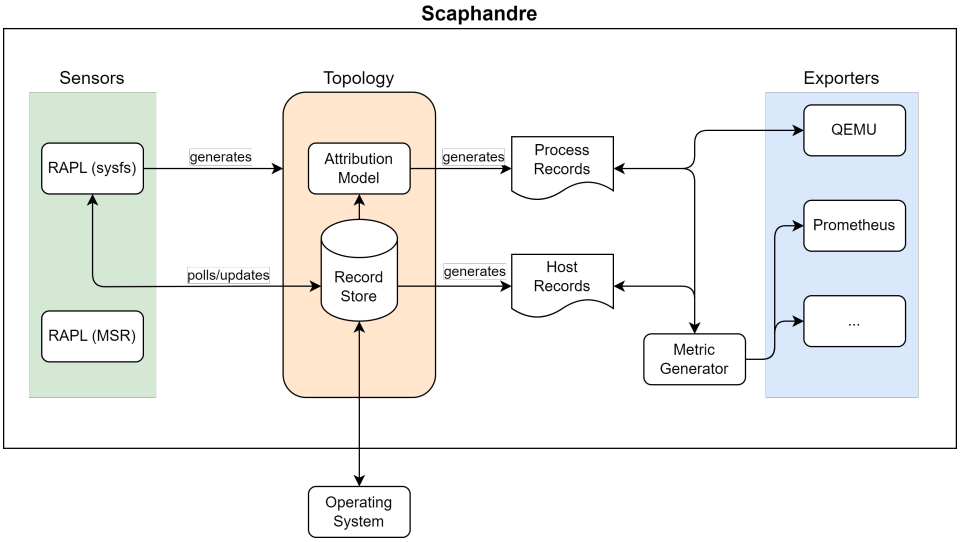


Figure 5.1.: Internal architecture of Scaphandre

The internal architecture of Scaphandre can be seen in Figure 5.1 and consists of three main parts: Data collection using **Sensors**, attribution and data transformation through the **Topology**, as well as data publication using a **MetricGenerator** and **Exporters**. On startup, a Sensor appropriate for the operating system is initialized (such as `powercap` RAPL under Linux). This Sensor then generates a Topology, which models the the host from an energy-consumption perspective. The Topology reads energy data from the Sensor, keeps track of running processes and their resource usage and gathers other system-level metrics. This data is stored as a series of **Records** inside the Topology containing a timestamp, measurement unit and value. For metrics generation, the MetricGenerator polls the Topology for records and transforms them into metrics, such as resource usage or attributed power consumption. These metrics are then in turn made public by an Exporter selected through a command-line parameter, either for ingress into a metrics collector such as Prometheus or for passthrough to VMs. Table 5.3 shows how this structure maps onto the attribution model.

To implement the host attribution layer with Scaphandre, the following steps were per-

Attribution Model	Scaphandre
Energy Measurement	Sensor
Resource Utilization Measurement	Topology
Energy Attribution	Topology + MetricGenerator
Passthrough/Propagation	Exporters

Table 5.3.: Mapping of model concepts to Scaphandre components

formed:

1. Validate Scaphandre energy counter and resource utilization implementations
2. Add IPMI sensor and export for host power metrics
3. Implement attribution model for CPU utilization
4. Setup passthrough between Scaphandre VM/QEMU exporter and Proxmox VMs

Validation

Before implementing the attribution model in Scaphandre, a validation of the sensor readings and OS statistics reported by the existing implementations was performed. In this initial testing, the RAPL Sensor component provided accurate readings in line with direct readouts from the `powercap` interface. For process utilization, Scaphandre already tracked CPU time in jiffies for each process in the Topology, separating between the various CPU time types reported by Linux (`user`, `nice`, `system`, etc.). The generated Records accurately tracked the OS-reported values, however the summed values were found to be incorrect due to double-counting guest CPU time inside of a helper function `total_time_jiffies()`, shown in Figure 5.2: For guest (KVM) processes, modern Linux kernels provide the actual CPU time used by the guest as separate fields (`guest` and `guest_nice`). This is on top of the regular `user` and `nice` fields which already include this guest time [32]. It was found that Scaphandres function for summing CPU time incorrectly added these values together, resulting in inflated values for CPU time. After fixing the sum, Scaphandre now reports accurate values for total and per-process CPU time, as well as energy consumption, sufficient for building the rest of the attribution model.

IPMI Power Measurement

On top of the energy values reported by RAPL, the base platform also provides power readings generated by the PSUs. These readings are accessible through the Data Center Manageability Interface, exposed by via the IPMI interface. The metric reporting system power consumption can be read programmatically using the IPMI Redfish API if available, or through a command-line utility such as `impitool`. This system power value provides a useful reference point for validating per-component (CPU) attribution based on RAPL, but is not

```rust
// in src/sensors/utils.rs

/// Returns the total of active CPU time spent, for this stat measurement
/// (not iowait, idle, irq or softirq)
pub fn total_time_jiffies(&self) -> u64 {
    let user = self.user;
    let nice = self.nice;
    let system = self.system;
    // ... truncated
    let guest_nice = self.guest_nice.unwrap_or_default();
    let guest = self.guest.unwrap_or_default();
    trace!(
        "CPUStat contains user {} nice {} system {} idle: {} irq {} softirq {} iowait
        ↪ {} steal {} guest_nice {} guest {}",
        user, nice, system, idle, irq, softirq, iowait, steal, guest_nice, guest
    );
    // Incorrect
    //user + nice + system + guest_nice + guest
    // Correct
    user + nice + system
}
```

Figure 5.2.: Incorrect and correct per-process CPU time summarization

used in attribution itself. It is also too imprecise to be used as a proxy value for estimating the consumption of other components - it only provides a low temporal resolution power reading (in Watts), rather than the microjoule-level energy counter of RAPL.

The implementation performs IPMI power readings through an additional helper Sensor in Scaphandre. The sensor uses the `ipmitool` utility to read the instantaneous power consumption of the system without the need for a Redfish or similar API. It is then queried directly by the MetricGenerator, bypassing the Topology record store used for the RAPL sensor. This is necessary due to the IPMI reading being a time-dependent power reading instead of an energy counter. From there, Scaphandre can generate an IPMI power consumption metric (`scaph_ipmi_power_watts`), which is then exported by the chosen exporter.

Attribution Model Implementation

Scaphandres existing implementation for attributing power consumption was located in the Topology and used the CPU utilization percentage as its proxy value for utilization. Figure 5.3 shows the core logic of the existing attribution. For any given measurement interval Δt, Scaphandre multiplied the total reported power consumption (`conso_f64`, calculated in `get_records_diff_power_microwatts`) by each processes reported CPU utilization U_p^c in percent. During testing we found this method to underestimate power consumption, especially on lightly loaded systems. Consider a system with only a single process running at 50% CPU

```rust
// in src/sensors/mod.rs -> impl Topology {}
pub fn get_all_per_process(&self, pid: Pid) -> Result<_> {
    let process_cpu_percentage = record.process.cpu_usage_percentage /
    ↪ self.proc_tracker.nb_cores as f32;
    //...
    let conso_f64 =
    ↪ self.get_records_diff_power_microwatts().parse(/*truncated*/).unwrap();
    let result = (conso_f64 * process_cpu_percentage as f64) / 100.0_f64;
    res.insert(
        String::from("scaph_process_power_consumption_microwatts"),
        Record::new(record.timestamp, result.to_string(), units::Unit::MicroWatt),
    );
}:

pub fn get_records_diff_power_microwatts(&self) -> Option<Record> {
    if self.record_buffer.len() > 1 {
        let last_record, last_microjoules, previous_microjoules = // ..record
        ↪ retrieval and parsing truncated
        let microjoules = last_microjoules - previous_microjoules;
        let time_diff = last_record.timestamp.as_secs_f64()
            - previous_record.timestamp.as_secs_f64();
        let microwatts = microjoules as f64 / time_diff;
        return Some(Record::new(
            last_record.timestamp,
            (microwatts as u64).to_string(),
            units::Unit::MicroWatt,
        ));
    }
    None
}
```

Figure 5.3.: Existing Scaphandre energy attribution model, using CPU percentage as its proxy value

utilization. Given this utilization, Scaphandres attribution model will only attribute 50% of the reported energy consumption to the process, despite it being the only process running and thus being responsible for 100% of the CPU energy consumption.

To replace this implementation with the attribution model, a way to calculate C_p^c based on relative CPU time instead of raw percentage was introduced, along with differentiation between dynamic and static energy (E_D and E_S). Since the existing implementation mainly operates on power instead of energy values, the model was adjusted accordingly. This has little to no effect on the actual logic of the model, but results in the final metric being a continuous power reading instead of an energy measurement. Figure 5.4 shows the implementation of C_p^c using CPU time as reported by the OS.

```rust
// in src/sensors/mod.rs -> impl Topology {}
pub fn get_process_attribution_factor(&self, pid: Pid) -> Option<f64> {
    let record, previous_record = //..truncated

    let cpu_time_total = self.get_stats_diff()?.total_time_jiffies();
    let cpu_time_process = (record.process.stime + record.process.utime)
        .saturating_sub(previous_record.process.stime +
        ↪  previous_record.process.utime);

    // Process utilization = fraction of power consumption to be attributed to this
    ↪  process
    Some(cpu_time_process as f64 / cpu_time_total as f64)
}
```

Figure 5.4.: Implementation of C_p^c calculation for CPU power as defined in the model

To account for static power consumption ($\frac{E_S}{\Delta t}$), a command-line parameter for setting a static power value (to be determined via external measurement) was introduced. A new function `get_records_power_diff_microwatts_dynamic()` then subtracts $\frac{E_S}{\Delta t}$ from $\frac{E}{\Delta t}$ (as returned by `get_records_diff_power_microwatts`) to retrieve $\frac{E_D}{\Delta t} = \frac{E-E_S}{\Delta t}$, shown in Figure 5.5.

From there, we can calculate $\frac{E_p^c = E_D^c * C_p^c}{\Delta t}$ (Equation 4.4) by using the values returned from `get_records_power_diff_microwatts_dynamic()` and `get_process_attribution_factor` to arrive at a final value for the per-process per-component power consumption $\frac{E_p^c}{\Delta t}$.

QEMU Exporter

As mentioned before, Scaphandre already includes an exporter intended for usage with VM passthrough. The included QEMU exporter scans the host for running VMs (by checking for `qemu-system` processes) and performs energy attribution with help of the model embedded in the Topology. It then generates a directory tree that emulates a minimal `powercap` RAPL tree, which is intended for passthrough to VMs. This exporter is intended for QEMU VMs managed through `libvirt`, which is not used by the Proxmox hypervisor in use on the test

```rust
// in src/sensors/mod.rs -> impl Topology {}
pub fn get_records_power_diff_microwatts_dynamic(&self) -> Option<Record> {
    let rec_full = self.get_records_power_diff_microwatts_full()?;

    if let Some(static_uw) = self.static_power_microwatts {
        let uw_full = rec_full.value.trim().parse::<u64>(); // error handling
        ↪ truncated
        let uw_dynamic = (uw_full as f64 - static_uw).max(0.0);
        return Some(Record::new(
            rec_full.timestamp,
            (uw_dynamic as u64).to_string(),
            units::Unit::MicroWatt,
        ));
    }
    Some(rec_full)
}
```

Figure 5.5.: Scaphandre implementation of dynamic power calculation as defined in the model

system. To enable the QEMU exporter to find Proxmox-managed VMs, its detection logic was expanded to also find VMs launched by Proxmox, as shown in Figure 5.7.

The directory tree generated by the exporter is spartan by default, only featuring the energy consumption for a single root domain without additional information such as a `name` identifier. This could cause problems with guest-side tooling that may expect a more fully-featured `powercap` RAPL directory. This issue was mitigated by expanding the generated directory with domain name information and an additional "core" subdomain (which may be expected to be present). Said subdomain has its `energy_uj` counter duplicated from the outer "package" domain. Figure 5.6 shows the generated directory before and after this mitigation (compare to the native RAPL interface shown in Figure 2.2).

Since the attribution implementation natively uses power values but the RAPL counters are energy-based, a conversion to eliminate the measurement interval Δt was required. This conversion was notably missing in the original QEMU exporter, resulting in incorrect values for VM attribution if $\Delta t \neq 1s$. An abbreviated version of the corrected QEMU exporter can be seen in Figure 5.7 - note the initial retrieval of E_D and the per-process attribution according to Equation 4.4.

5.1.2. VM Passthrough

Scaphandres QEMU exporter provides a per-VM RAPL directory tree, but does not perform any kind of VM passthrough on its own. To make the directories available inside the guests, the implementation provides a shared directory solution between host and guest. This avoids use of network-based solution due to the high overhead and the requirement of a shared network between host and guest. Instead, the implementation uses the `virtiofs` daemon which implements a high-performance, low-overhead filesystem between a KVM host and

```
// Before
/var/lib/libvirt/scaphandre/<vmid>/
|-- intel-rapl:0
|   |-- energy_uj <-- energy counter
|-- intel-rapl:0:0

// After
/var/lib/libvirt/scaphandre/<vmid>/
|-- intel-rapl:0
|   |-- energy_uj <-- energy counter
|   |-- intel-rapl:0:0
|   |   |-- energy_uj <-- energy counter
|   |   |-- name <-- "core"
|   |-- name <-- "package-0"
```

Figure 5.6.: Scaphandre emulated RAPL directory before and after adjustments

guest on a shared-memory basis. This approach requires `virtiofsd` process per VM running on the host exposing a UNIX socket, as well as VM-specific configuration, using the socket for accessing the filesystem. A `hookscript` that was added to each node VM performs both of these tasks. On VM launch, the script launches the daemon process and adds the required parameters (see below) to the VM processes commandline.

```
-object memory-backend-memfd,id=mem,size=8192M,share=on -numa node,memdev=mem
      -chardev socket,id=char0,path=/run/virtiofsd/<vm_id>.sock -device
                vhost-user-fs-pci,chardev=char0,tag=<vm_id>
```

From there, the guest OS can now mount the shared filesystem inside the VM. A simplified version of this process is shown in Figure 5.8. Note that even non-instrumented VMs have energy attribution performed. This allows VMs to be added and removed as required without having to make adjustments to the Scaphandre configuration.

5.1.3. Host Deployment

We now have all the pieces needed for the host-level attribution layer of the implementation. The actual deployment on the Proxmox nodes is structured as shown in Figure 5.9: Since each Scaphandre process can only run one exporter, two Scaphandre instances must be deployed. The first instance uses the Prometheus exporter, providing general host/process metrics, as well as the IPMI reading. This instance is queried by external monitoring described in section 5.3. The second instance uses the QEMU exporter and performs VM attribution, before generating the RAPL passthrough directories. The Kubernetes cluster itself consists of a single control-plane node and several worker nodes, with each node corresponding to an individual VM. The k8s worker VMs are configured in two steps: On the host side, the aforementioned hookscript starts the passthrough daemon and makes the `virtiofs` instance

```
// in src/exporters/qemu.rs
impl QemuExporter {
    pub fn iterate(&mut self, path: String) {
        let uw_dynamic = self.topology.get_records_power_diff_microwatts_dynamic();
        let Some(t_diff) = self.topology.get_records_time_diff();
        let uj_dynamic = uw_dynamic * t_diff;
        let processes = self.topology.proc_tracker.get_alive_processes();
        let qemu_processes = QemuExporter::filter_qemu_vm_processes(&processes);
        for qp in qemu_processes {
            let vm_name = QemuExporter::get_vm_name_from_cmdline(&qp);
            if qp.len() > 2 {
                let last = qp.first().unwrap(); // ... truncated
                if let Some(proc_utilization) = self.topology
                    .get_process_attribution_factor(last.process.pid)
                {
                    let uj_to_add = proc_utilization * uj_dynamic;
                    match QemuExporter::add_or_create(exported_path, uj_to_add as
                    ↪ u64) {
                        // ... error handling truncated
                    }
                }
            }
        }
    }
    fn filter_qemu_vm_processes(processes: /* truncated */) ->
    ↪ Vec<Vec<ProcessRecord>> {
        let mut qemu_processes = vec![];
        // ... inside for vecp in processes.iter() {
        if let Some(pr) = vecp.first() {
            if let Some(res) = pr.process.cmdline.iter()
                .find(|x| x.contains("qemu-system")
                    // enables Proxmox VM detection
                    || x.contains("/usr/bin/kvm"))
            { /* truncated */ }
        }
        qemu_processes
    }
    fn get_vm_name_from_cmdline(cmdline: &[String]) -> String {
        /* libvirt name detection truncated */
        for pair in cmdline.windows(2) {
            if pair[0] == "-id" {
                return pair[1].clone();
            }
        }
        String::from("")
    }
}
```

Figure 5.7.: Scaphandre QEMU VM exporter after modifications

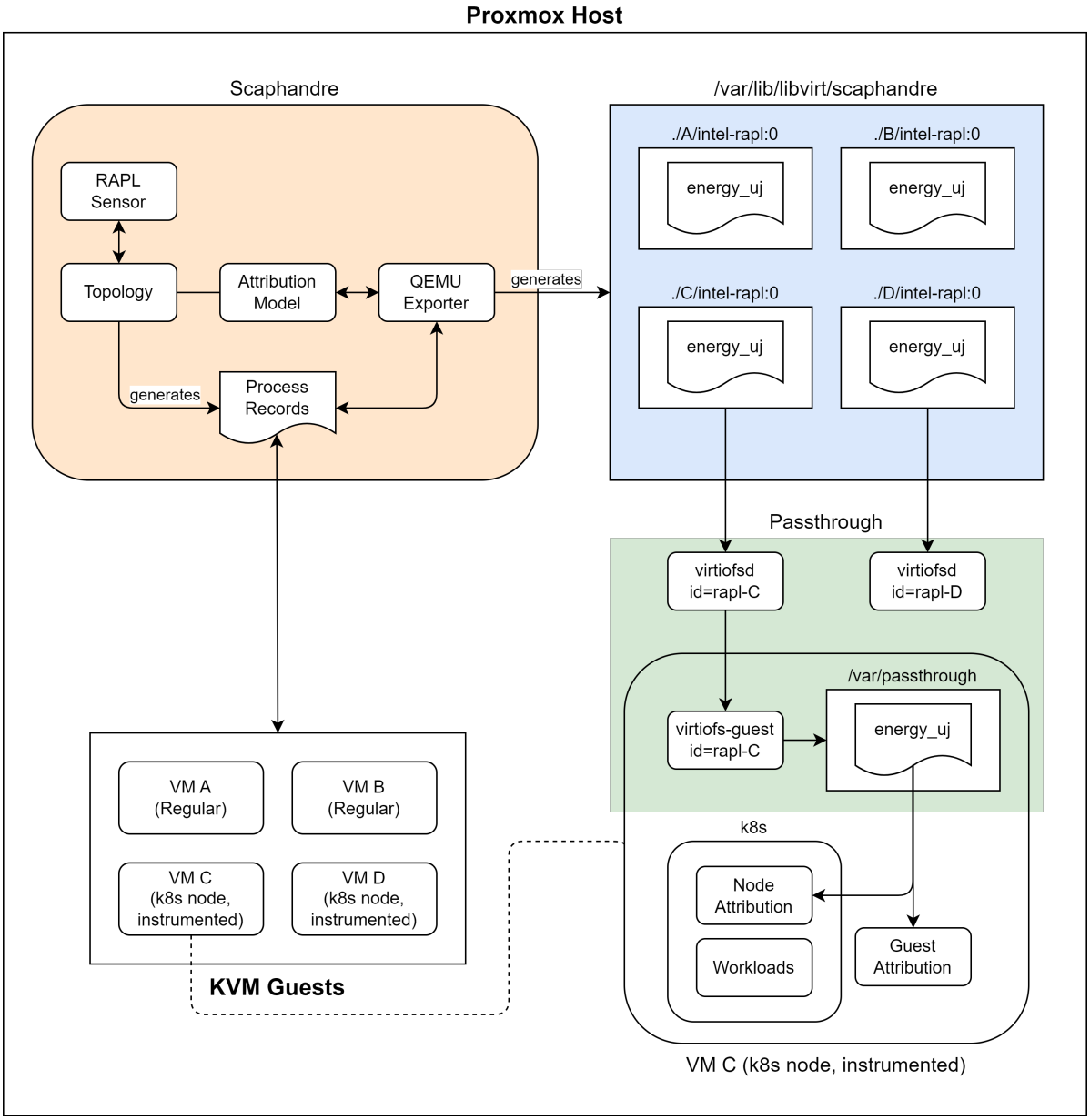


Figure 5.8.: VM attribution passthrough, using Scaphandre and `virtiofs`

available to the VM. Guest side configuration is covered in subsection 5.2.2. Most parts of this process are automated through Ansible to improve repeatability of the deployment and prevent errors caused by manual modifications.

5.2. Kubernetes Nodes and Cluster

The host-level implementation is now able to provide guests with their attributed energy consumption through a virtiofs filesystem emulating the `powercap` RAPL directory tree. From here, the node/guest-level attribution component of the implementation must convert this per-VM reading into a per-workload/container result. The guest-level implementation must:

1. use the virtiofs passthrough as a source for energy consumption
2. enumerate processes running inside the guest, including containerized processes from Kubernetes workloads.
3. associate discoverd container processes with their respective Kubernetes workload API objects
4. use the attribution model to estimate energy consumption for both plain processes and containers
5. export attribution data for external monitoring and benchmarking

These requirements are similar to the host-level requirements, but require a Kubernetes-aware solution. Existing solutions were again examined for their suitability, as reuse of existing tooling would again reduce the scope of potential errors and harness existing synergies. The Kepler project (see subsection 3.2.1) is the premier player in the space of Kubernetes energy attribution and appeared to cover most of the requirements: Kepler is a Kubernetes-native project and fully covers items 2 and 3 in its reported metrics. We found that its attribution model is equivalent to the one described in chapter 4, with the exception of using eBPF for CPU utilization instead of CPU time as reported through `proc`. This sufficiently covers item 4 and provides a point of comparison to the models utilization measurement approach. Finally, Kepler provides comprehensive metrics data through a Prometheus exporter, suitable for external collection.

Additionally, Keplers position as the dominant project in the Kubernetes energy attribution makes it an attractive choice for the implementation, as this will allow comparing its various operating modes. As mentioned before, Kepler does not require the presence of hardware counters for performing energy attribution. The project provides a variety of linear-regression-based models for estimating guest, host and workload energy consumption if no hardware interface is present. These models may be used on cloud provider platforms today to estimate energy usage, so investigating their accuracy is worthwhile. By basing the guest-level implementation on Kepler, directly comparing attribution performance between the passed-through sensor data and Keplers estimation models becomes possible.

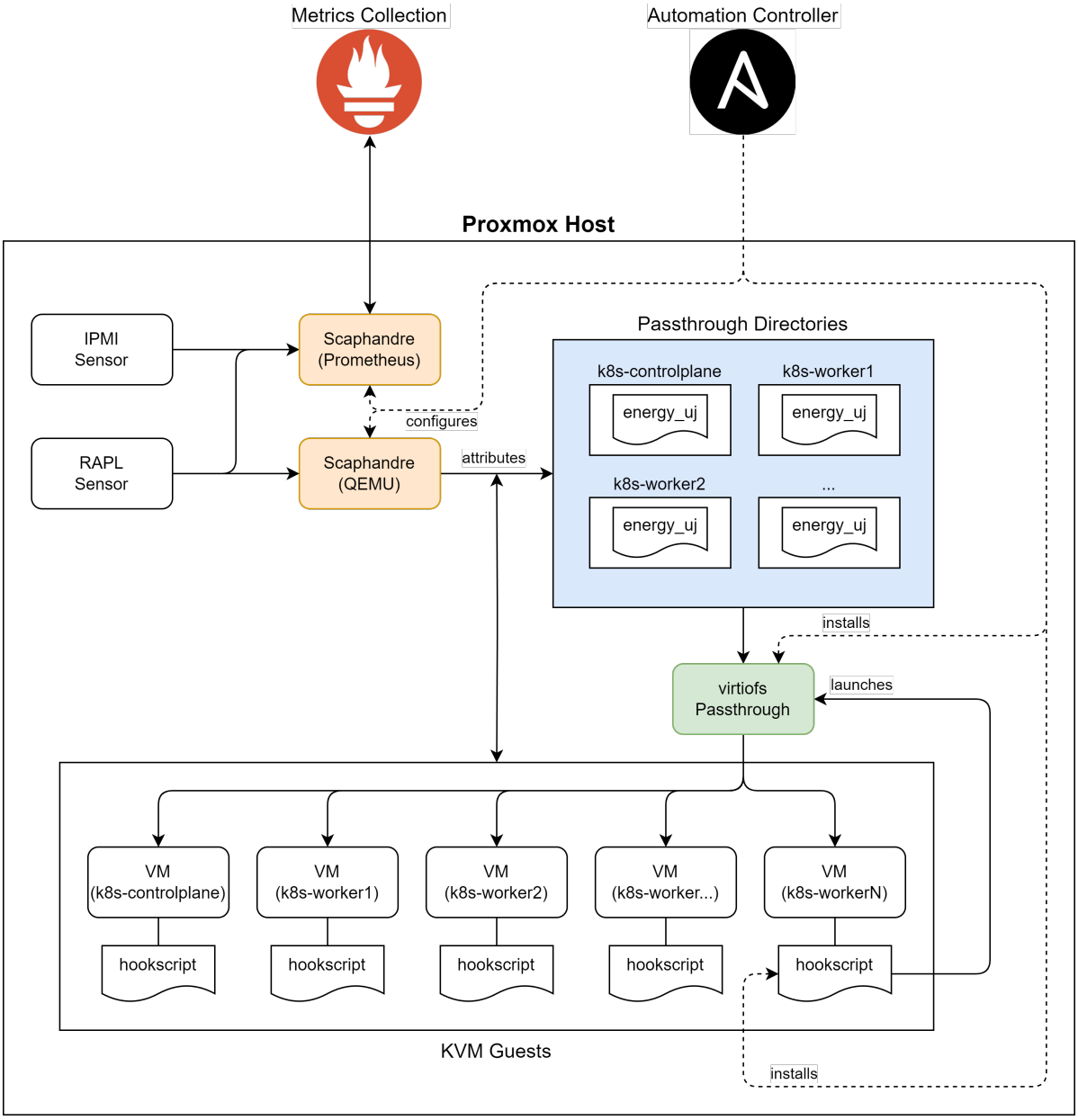


Figure 5.9.: Host-level deployment of implementation

5.2.1. Cluster Configuration

The Kepler exporter is designed to be run inside the Kubernetes cluster it is performing energy attribution for. A minimal Kubernetes cluster environment was chosen to eliminate potential sources of noise, using the `k3s` Kubernetes distribution as the baseline on top of a minimal Ubuntu 22.04 guest operating system. Each worker node runs a minimal `k3s agent` process, providing basic Kubernetes components such as `kubelet`. The control-plane VM node performs all control-plane operations (using `k3s server`) and runs the API server processes. Non-essential services were disabled to reduce noise and the control-plane nodes were tainted to ensure separation between workloads and k8s processes. This approach mirrors managed Kubernetes offerings from major cloud providers such as AWS EKS and Microsoft Azure AKS services. In these services, the control plane is operated entirely by the cloud provider and thus unavailable for energy attribution. The resulting `k3s server` configuration can be found in Figure 5.10.

```
disable:
- traefik
- local-storage
node-taint:
- node-role.kubernetes.io/control-plane=true:NoSchedule
- CriticalAddonsOnly=true:NoExecute
```

Figure 5.10.: k3s server configuration to minimize noise and ensure workload separation

5.2.2. Kepler Implementation

To integrate Kepler as the guest component of the attribution solution, it muse be able read the passed through energy attribution data. Since the Kepler project already features support for reading a `powercap` RAPL directory tree, the passed-through directory simply needs to be made available to it. Since Kepler is itself running inside a Kubernetes-managed container, this involves two steps: First, one must mount the `virtiofs` filesystem at a well-known location on the host. One can then pass through this directory to the container using the Kubernetes `hostPath` volume type, mapping it to the desired path inside the container [41]. Mounting the `virtiofs` filesystem at the usual RAPL interface path (`/sys/class/devices/powercap/intel-rapl`) inside the container would allow for the passthrough to be entirely transparent to Kepler (or other guest-side attribution software), thus avoiding any modifications.

Unfortunately we found that this approach is not feasible due to Kernel-Level protections enforced onto the `/sys` filesystem: Since `/sys` is provided by the Kernel, it is protected from having other filesystems mounted inside the `/sys` directory, even by root. This makes emulating a true RAPL interface impossible and forces the use a custom, non-protected path. To make this work with Kepler, an optional `RAPL_PATH` environment variable was introduced into the application that, when set, overrides the default path of `/sys/class/devices/powercap/intel-`

`rapl`. The passthrough RAPL directory can then be mounted at a custom path with `RAPL_PATH` set accordingly, allowing Kepler to see the passthrough attribution as a regular RAPL interface.

For the first step, the automation controller installed a systemd mount unit inside all guest VMs. Said unit is shown in Figure 5.11 - it is started on boot and mounts the VM-specific `virtiofs` to a predictable path on the host. This means that the passthrough is not transparent on the guest level, but the level of instrumentation required is still minimal.

```
[Unit]
Description=Scaphandre QEMU export virtiofs passthrough
[Mount]
What=<vmid>-<vmid>
Where=/var/scaphandre
Type=virtiofs
Options=defaults
[Install]
WantedBy=multi-user.target
```

Figure 5.11.: Systemd mount unit providing the passthrough RAPL directory

The actual Kepler deployment is performed with Helm using a slightly modified official chart, with added support for defining custom volumeMounts [42]. As shown in Figure 5.12, the passthrough directory mounted on the guest is bound into the container, along with addition of the aforementioned `RAPL_PATH` environment variable. From there, Kepler is able to use the VM attribution data to perform per-process attribution inside the guest. Finally, the generated metrics are made available through a simple `hostPort` mapping. This allows using an external metrics aggregator outside of the Kubernetes cluster, reducing potential noise. The node-exporter utility provided by the Prometheus project was also installed, as Kepler uses it to gather node metrics.

5.2.3. Guest and Kubernetes Deployment

Figure 5.13 depicts the final deployment structure of the guest-level attribution components. Note that the Helm chart installs additional resources into the cluster, but only the `DaemonSet` resource that is responsible for the kepler-exporter pods is shown for simplicity. Both guest-OS and Kubernetes operations are performed by the automation controller. This setup creates minimal overhead on the guest while providing Kepler with direct access to the passthrough data.

5.3. Metrics Aggregation and Visualization

The primary goal of the implementation is to provide operators with detailed information about their systems energy consumption. The metrics generated by the implementation form the basis for this task, but are too primitive/low-level on their own. To make the data

```yaml
#values.yaml
image: # custom image with RAPL_PATH support
extraEnvVars:
  RAPL_PATH: /var/scaphandre
service:
  nodePort: 9002 # used by external metrics aggregator
extraVolumes:
  - name: var-scaphandre
    hostPath:
      path: /var/scaphandre
      type: Directory
extraVolumeMounts:
  - name: var-scaphandre
    mountPath: /var/scaphandre
```

Figure 5.12.: Kepler Helm chart values.yaml for enabling passthrough attribution

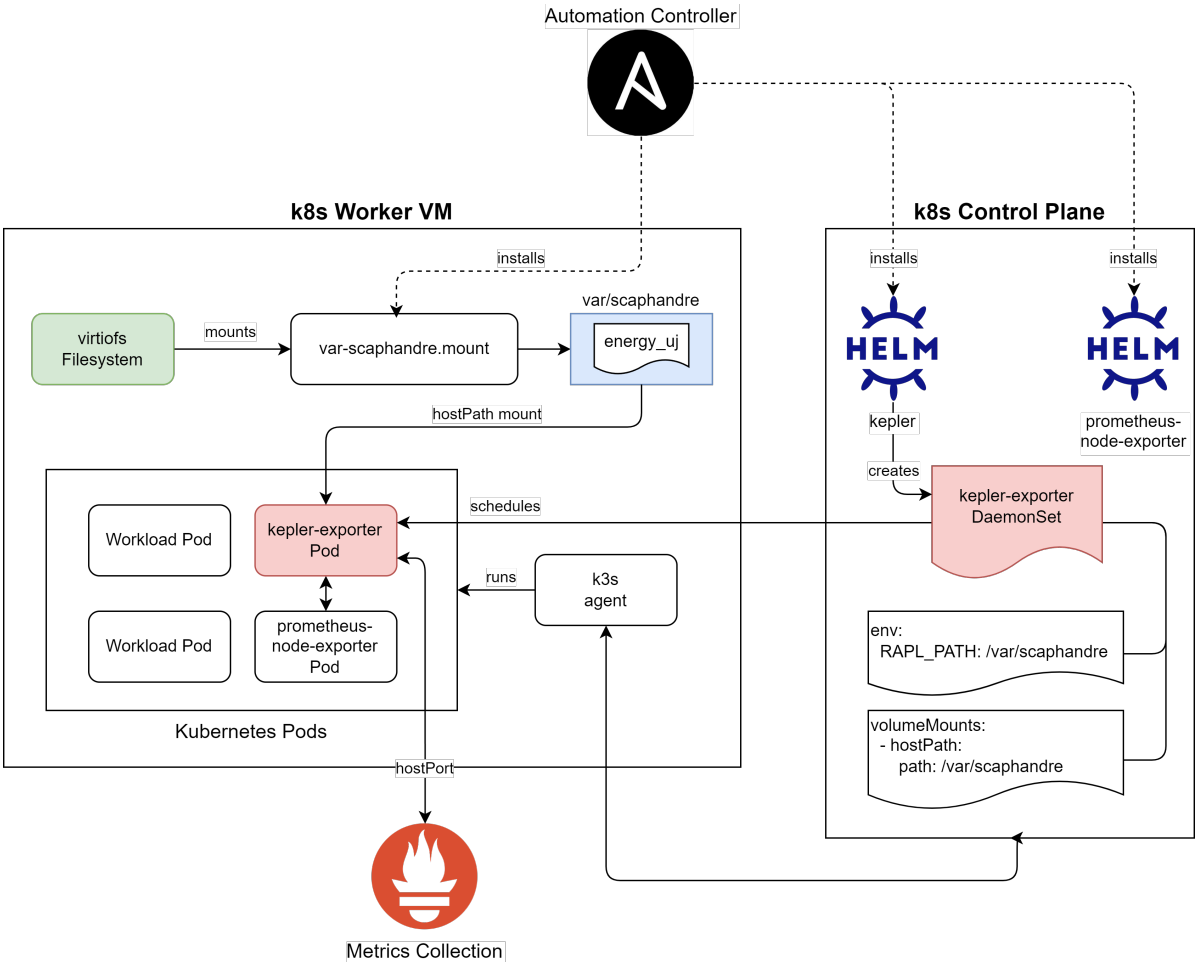


Figure 5.13.: Guest-side deployment, including Kubernetes

contained in these metrics more accessible, one must first aggregate and transform these metrics so that key information can be extracted. The gathered results can then be presented in an intuitive fashion that permits further manual processing.

Both host and guest implementations publish metrics through a Prometheus-compatible exporter over HTTP. An externally hosted Prometheus instance was used to collect these metrics into Prometheus. By using an external instance, any energy consumption caused by the metrics aggregation itself is removed from reported energy values, reducing noise. The Prometheus time-series database was then queried through a Grafana instance, which was also used for final presentation of the reported values. Aside from adhoc queries used for debugging and inspection, the main method for data visualization are two Grafana dashboards - one for host Scaphandre metrics and one for Keplers guest metrics. The values used for these dashboards and subsequent measurements in chapter 6 can be seen below.

Name	Query (without filters)
IPMI Power	`scaph_ipmi_power_watts`
IPMI Dynamic Power	`scaph_ipmi_power_watts - $static_power_ipmi`
RAPL Power	`scaph_host_power_microwatts / 1000000`
RAPL Dynamic Power	`scaph_host_power_microwatts / 1000000 - $static_power_rapl`
VN+Process Attribution	`scaph_process_power_consumption_microwatts / 1000000`

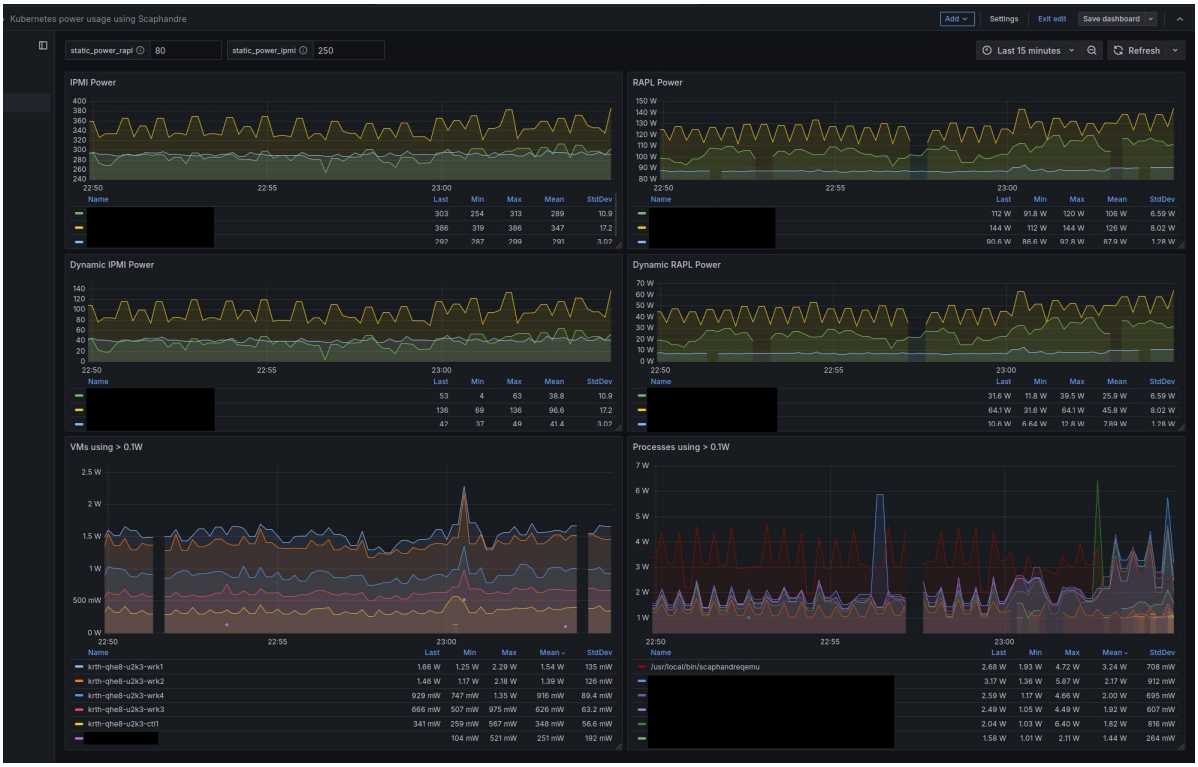
Table 5.4.: Scaphandre metrics used for dashboard and tests

The Scaphandre dashboard can be seen in Figure 5.14 and visualizes the key metrics described in Table 5.4. It displays both IPMI and RAPL power readings, both total ($\frac{E}{\Delta t}$) and dynamic $\frac{E_D}{\Delta t}$ by means of user-provided values for static power consumption. Calculated VM and process attribution is shown next, with the load test figure showing all VMs reporting increased power attribution, as expected.

Name	Query (without filters)
Pod Package Power	`irate(kepler_container_package_joules_total)[1m]`
Node Package Power	`irate(kepler_node_package_joules_total)[1m]`
Pod Other Power (Estimate)	`irate(kepler_container_other_joules_total)[1m]`

Table 5.5.: Kepler metrics used for dashboard and tests

The Kepler dashboard (Figure 5.15) provides both guest node and cluster workload information, using the metrics in Table 5.5: It visualizes attributed package power consumption on a per-pod basis, as well as the nodes total power consumption (equivalent to the VM attribution provided by Scaphandre). Finally, it reports estimates for total system power consumption (OTHER) - these are always generated, even with RAPL data available.



(a) Idle

(b) Load

Figure 5.14.: Scaphandre Grafana dashboard

(a) Idle

(b) Load

Figure 5.15.: Kepler Grafana Dashboard

6. Test Results

With the implementation complete, we can turn to part two of RQ2, the performance of the attribution implementation. To answer this question the implementation was subjected to extensive benchmarks, collecting key attribution metrics for different workload scenarios. Since the implementation is focused on CPU energy consumption, a CPU-heavy task in the form of the `stress-ng` tool was used as the workload. `stress-ng` is a widely used synthetic benchmarking tool, allowing for easy replication and verification of the benchmarking results. To ensure that the implementation is resilient to noisy neighbours in the form of other VMs or multiple in-cluster workloads, attribution was also tested with "noise" workloads added at various layers of the attribution model. Finally, RQ3 is addressed by running the same scenarios against Kepler without passthrough enabled, allowing comparison of the hardware-backed passthrough results to Keplers estimates.

6.1. Testing Methodology

For each of the following tests, the component versions described in Table 6.1 were used, along with following environment setup:

Component	Version
Host OS	Proxmox PVE 8.3.2, Kernel 6.11.0-2-pve
Guest OS	Ubuntu 22.04.05 LTS, Kernel 5.15.0-134
Kubernetes	k3s v1.31.5+k3s1

Table 6.1.: Software revisions used for test runs

First, all cluster VMs were started on a single hypervisor machine. All other running VMs were then removed from said hypervisor to prevent noise. Each of the node VMs was allocated a total of 128 virtual cores and 16 GiB of RAM to ensure sufficient resources were available. The k8s node virtual machines and k8s cluster were configured as described in subsection 5.2.2.

Before each test, all existing workloads were eliminated and the system was allowed to to settle for 5 minutes. Workload pods were then created inside Kubernetes using the deployment manifest shown in Figure 6.1. This deployment creates a container on a given number of nodes (`<nodes>`), with each container running an instance of `stress-ng`. Each `stress-ng` process in turn spawns a given number of CPU workers (`<workers>`), where each worker stresses one CPU thread. Therefore the total number of stress threads is the number

of nodes that the workload will run on, multiplied by the number of stress workers. The `topologySpreadConstraints` ensures that multiple pods are not scheduled on a single node, while still allowing for any number of replicas below or equal to the node count. Once the manifest was created the system was allowed to settle, before metrics were measured over an interval of 5 minutes, not including the initial reconciliation period. For each metric measured the mean value was collected over a 5 minute period, along with the absolute standard deviation. The sampled data was also inspected visually using the dashboards detailed in section 5.3 to catch any oddities or outliers, which are discussed below when relevant.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bench-workload
  namespace: bench
spec:
  replicas: <nodes>
  selector:
    matchLabels:
      app: bench-workload
  template:
    metadata:
      labels:
        app: bench-workload
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: kubernetes.io/hostname
        whenUnsatisfiable: DoNotSchedule
        labelSelector:
          matchLabels:
            app: bench-workload
      containers:
      - name: bench-workload
        image: wdhif/stress-ng
        args: ["--cpu", "<workers>"]
```

Figure 6.1.: Workload manifest template used for tests

6.2. Attribution Testing

The first series of tests aims to measure the accuracy of the attribution model, thus verifying its basic functionality. Each of these tests consists out of a single scheduled workload as shown in Figure 6.1. To be able to validate the attribution results, the hypervisors static (no VMs) and idle (no Workload) power consumption were measured first, shown in Table 6.2.

We can then subtract the static values from the measured total power consumption during the workload test to arrive at a reference value for dynamic (and thus attributable) energy consumption.

	IPMI	RAPL
Static (no VMs running)	Mean 278 W, $\sigma = 5$ W	Mean 80 W, $\sigma < 1$ W
Idle (VMs running, no workload)	Mean 291 W, $\sigma = 7$ W	Mean 87 W, $\sigma = 4$ W

Table 6.2.: Static and idle power consumption of the test hypervisor node

6.2.1. Single-Node Workloads

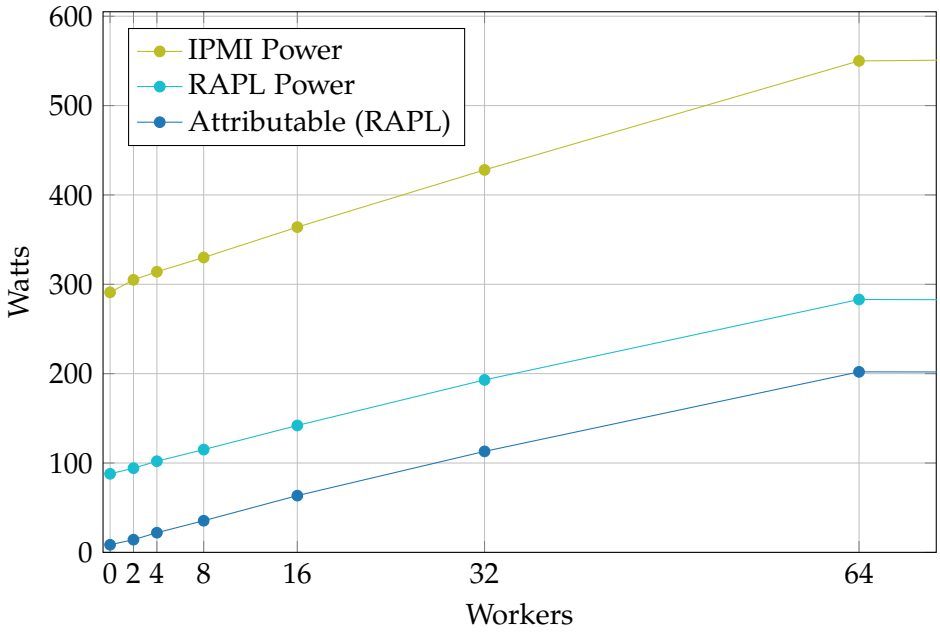


Figure 6.2.: Measured power consumption for single-node workloads. Results for the 128 worker test were nearly identical to 64 workers and have been omitted.

To evaluate basic attribution performance, a single-node workload benchmark was performed, utilizing 2, 4, 8, 16, 32, 64, and 128 `stress-ng` workers. Since these workloads range from low to high load, they also enabled gathering of information about non-linear characteristics of CPU power consumption (γ, as mentioned in section 4.2). Figure 6.2 shows the measured power consumption during these tests, as well as the total dynamic CPU power calculated from the RAPL metrics. At maximum load, 557 W of power consumption were reported by the IPMI interface and 281 W from RAPL, close to the 300 W TDP of the target platform CPU. We can observe that reported CPU power consumption scales almost linearly with utilization between 2 and 64 workers (which is the amount of physical cores in the

system). This is unexpected but helpful, as it means that the choice to ignore non-linear CPU scaling should not affect attribution accuracy. The consistency of these measurements was high, with standard deviation σ for IPMI power ≤ 6 W; for RAPL σ was consistently < 2 W. The dynamic power measured from RAPL will serve as the attribution target for future benchmarks, being the maximum attributable component power for each scenario.

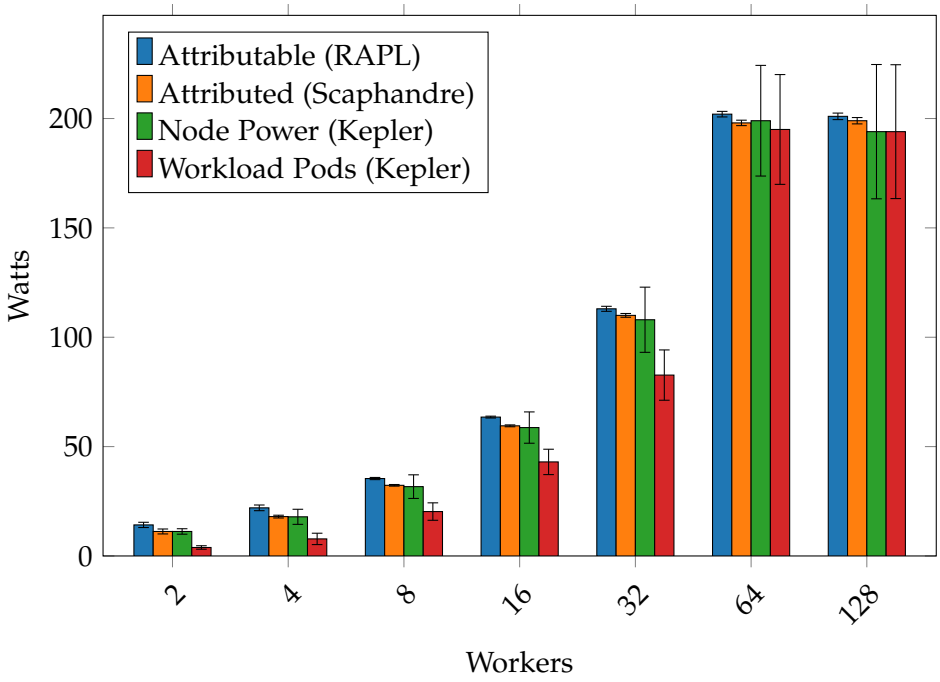


Figure 6.3.: Attribution performance for single-node workloads

The next series of tests examined attribution performance: Ideally, the power attributed to the test workload should be very close to the measured dynamic power (attributable), since the workload is the only load on the system. Overhead from virtualization and k8s may result in an attribution value slightly below this total dynamic power. The results can be seen in Figure 6.3, showing the reference (attributable/dynamic power), the power attributed to the VM by Scaphandre, the node power as reported by Kepler, and the final power attributed to the workload pods. Overall, we can observe excellent results for the host-level attribution, with the Scaphandre model attributing almost all of the dynamic power difference to the node VMs (orange). This performance is consistent across worker counts and shows remarkably low variance throughout the entire test range. Keplers node power (as reported by `node_package_power`) represents the guest-side view of this attribution value (green). Its means are in good agreement with the Scaphandre attribution metric, but show significantly higher standard deviation σ, caused by Kepler periodically under-reporting node package power for short bursts. Figure 6.4 shows the observed behavior, with short dips on the reported consumption. The source of this misbehavior could not be determined, but the passthrough mechanism itself excluded as a cause, providing consistent values over

time. Overall, relative σ was in the range of 15-20%, with the 4-worker scenario being the worst-case scenario at 33%.

Looking at the actual workload attribution performance (red), we can see that Keplers ratio power model under-performs significantly for lower-load scenarios. Despite the host layer providing excellent attribution values, the low-load attribution values are less than half of attributed node power. Table 6.3 shows precise values along with σ values. At higher loads Keplers performance improves, with the full-utilization tests showing excellent agreement with host-level attribution. This shows that Keplers implementation is capable of correct attribution in principle, but fails to do so consistently.

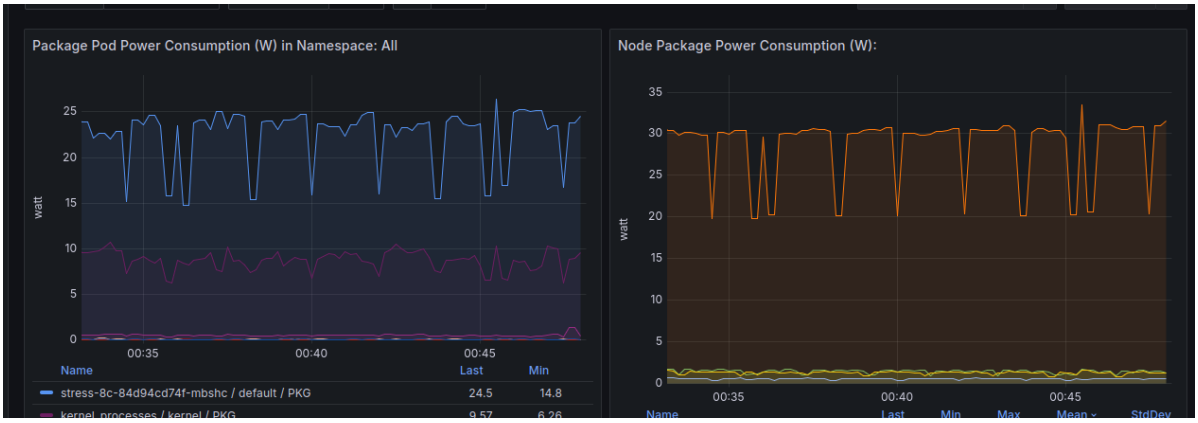


Figure 6.4.: Kepler under-reporting of node power consumption in semi-regular intervals

Workers	Attributable (W)	Workload Attributed (W)	Attributed (rel.)	σ (W)	σ (rel.)
2	14.2	3.88	27%	0.8	21%
4	22	7.8	35%	2.6	33%
8	35.4	20.3	57%	3.99	19%
16	63.5	43.2	68%	5.8	13%
32	113	82.7	73%	11.5	14%
64	202	195	95%	25.1	12%
128	201	194	96%	30.6	16%

Table 6.3.: Kepler attribution performance and error for single-node workloads

While the performance at lower load levels is slightly disappointing, the attribution solution is still able to provide energy estimates that lie within a factor of two in these scenarios accounting for overhead, with accuracy increasing drastically as load goes up, reaching 95% (more likely near-100% with overhead).

6.2.2. Multi-Node Workloads

With basic attribution functionality and performance confirmed, the next series of tests covers multi-node workloads. The series includes both symmetric workloads with a workload pod on each node, as well as asymmetric workloads with less pods than nodes. Workloads are denoted as `XnYc`, where X is the number of nodes that have a workload running on and Y is the number of `stress-ng` workers per node/pod. The total number of `stress-ng` CPU workers can be found by multiplying X and Y for any given workload.

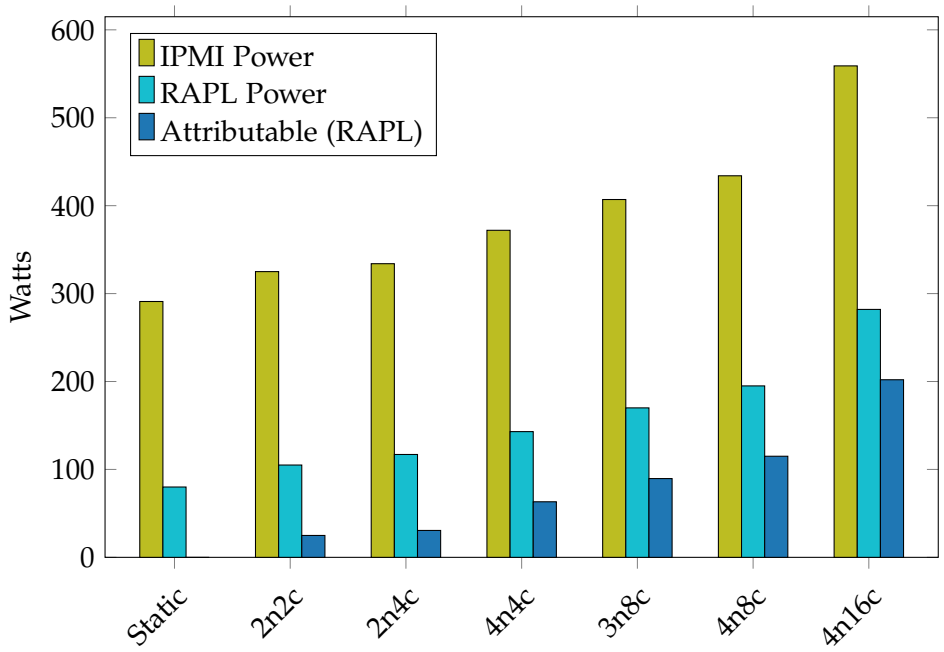


Figure 6.5.: Measured power consumption for multi-node workloads

The total measured power and calculated dynamic power reference is shown in Figure 6.5. These values show good agreement to the single-node power tests from Figure 6.2, with equivalent total `stress-ng` worker counts resulting in equivalent power consumption.

Multi-node attribution performance can be seen in Figure 6.6 - note that Kepler node power and workload pod power are now the sum of all individual node values. Again, attribution values should be close to the total measured dynamic power (attributable), with some minor losses to VM and k8s overhead. We can again observe host-level attribution values very close to the total reported dynamic power, in line with expected results thanks to the small virtualization overhead. This holds true for both symmetric and asymmetric loads, proving that the attribution model is capable of handling such scenarios. Attribution variance is low, with $\delta < 1$ W for almost all scenarios.

Keplers node power means are once again in good agreement with the host-level attribution. There were less "dips" in host power during the multi-node workloads, as evidenced by the lower deviation, shown in Table 6.4. While the single-node workloads showed a fairly

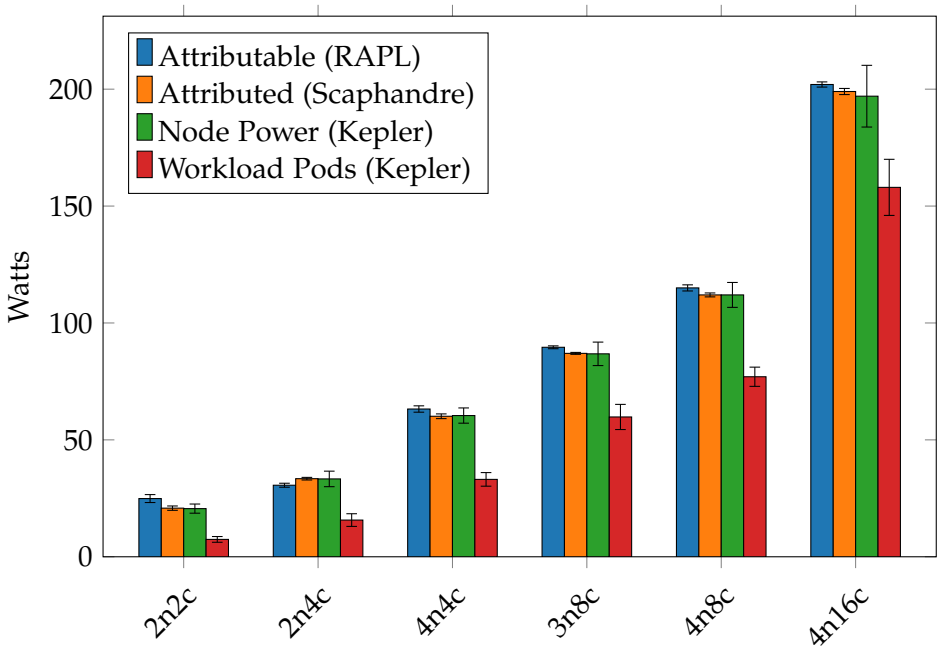


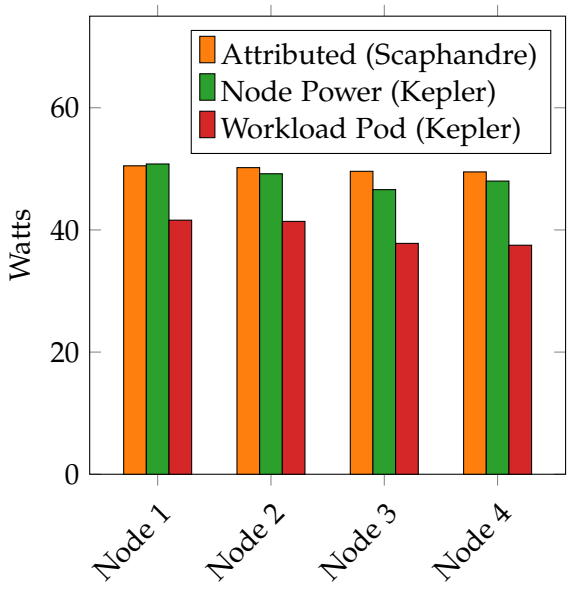
Figure 6.6.: Workload power attribution for multi-node workloads

consistent 15% $< \sigma$(rel.) $<$ 20%, multi-node workloads showed a relative $\sigma < 10\%$, a significant improvement (the low-load tests showed higher error but likely suffered from noise much more than the higher-load tests). It is possible that Keplers host power model is better suited for multiple, evenly-loaded nodes over the highly asymmetric load in the single-node benchmarks. The low error in node power also appears to reduce the error seen in the final workload attribution. While this higher confidence in attribution values is a positive, the power values generated by Keplers ratio power model consistently under-attributed workload power in the tested scenarios. Whereas Kepler was able to attribute over 95% of dynamic power to the 64-worker single-node workload, the best multi-node result was an attribution of 78% in the 4n16c test (which also featured a total of 64 workers).

Workload	Attributable (W)	Workload Attributed (W)	Attributed (rel.)	σ (W)	σ (rel.)
2n2c	24.9	7.43	30%	1.23	16%
2n4c	30.6	15.7	51%	2.71	17%
3n8c	89.6	59.8	67%	5.38	9%
4n4c	63.2	33.1	52%	2.91	8%
4n8c	115	77	67%	4.11	5%
4n16c	202	158	78%	12	8%

Table 6.4.: Kepler attribution performance and error for multi-node workloads

Looking at the per-node attribution results for 4n16c in Figure 6.7, we can see consistent



Node	1	2	3	4
Attributed	50.5 W	50.2 W	49.6 W	49.5 W
Node Power	50.8 W	49.2 W	46.6 W	48 W
Workload	41.6 W	41.4 W	37.8 W	37.5 W

Figure 6.7.: Per-node attribution performance for the 4n16c workload

behavior across all four nodes for the 4n16c workload, with attribution values varying only by a few Watts, within the above attribution error range. Other workloads also behaved consistently, as can be seen in Figure 6.8: For symmetric workloads like `4n4c` we can observe very similar values for each node, as expected given the symmetric load. The asymmetric workloads also show good results: Loaded nodes are attributed similar amounts as in an equivalent symmetric run - compare the loaded nodes in `3n8c` with those in `4n8c`. For the idle nodes, attribution values were equal to the standalone idle tests performed without any load on the cluster. From this, we can conclude that differing load patterns do not seem to affect host attribution significantly. Overall, the attribution model appears stable across multiple nodes, even in light of asymmetric workloads.

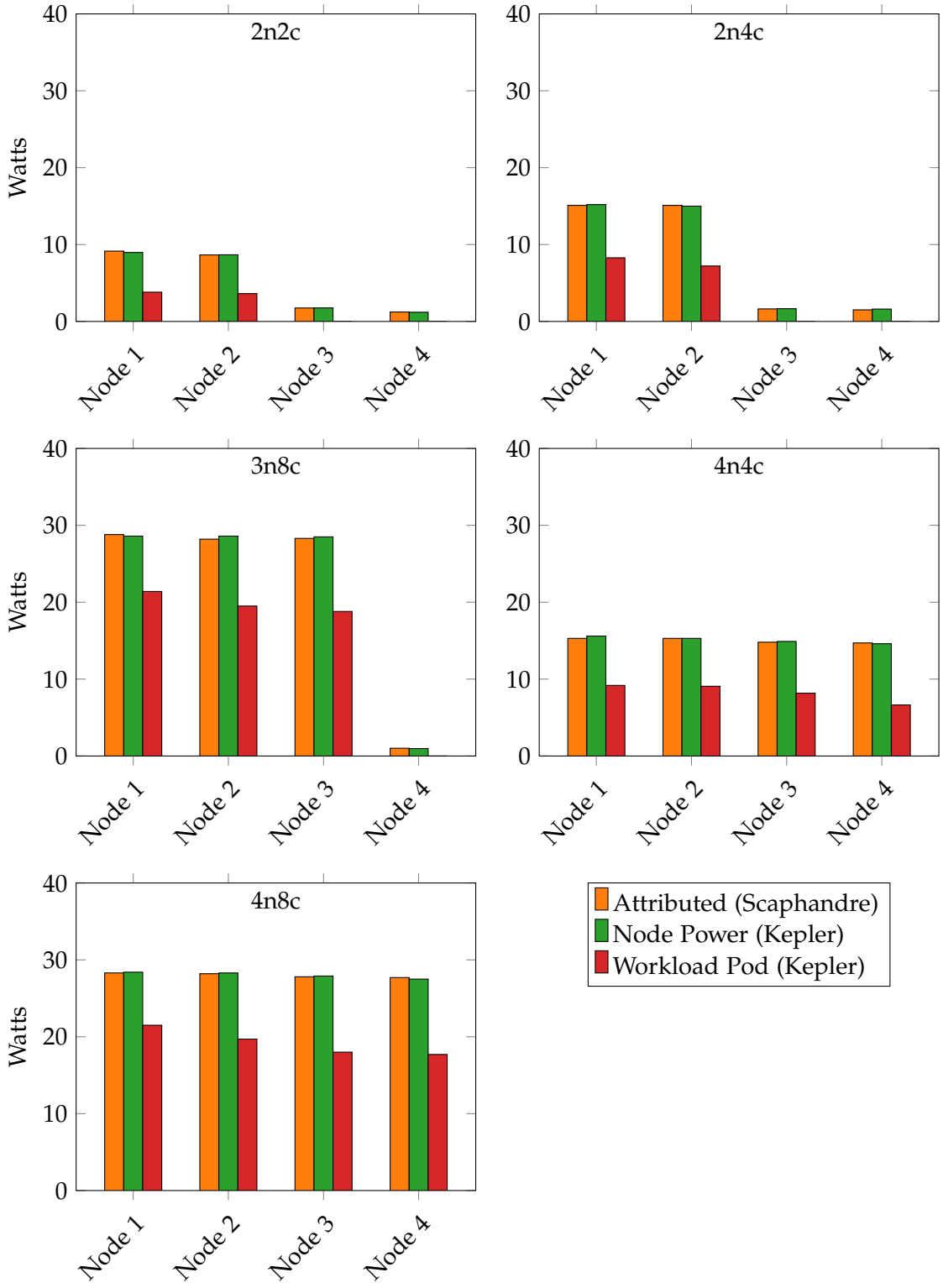


Figure 6.8.: Per-node attribution performance for various workloads

6.3. Noise Tests

As explained in section 1.1, one of the key challenges with attribution in cloud environments is the level of consolidation - a hypervisor may run dozens or hundreds of guests, all utilizing the same components. To simulate how such noise may affect attribution, the next test series feature a "noise" workload added at different attribution layers. The same `stress-ng` benchmark was used for this noise to ensure that results remain reproducible. The first series tests noise on the cluster layer, with varying levels of noise added through a second workload running at the same time as the main workload. The manifest for this noise workload was identical to the one shown in Figure 6.1. The second series then introduces noise by colocated VMs, as may be found in a multi-tenant cloud environment. Noise benchmarks are denoted as `XnYc_AnBc`, where X and Y are equivalent to multi-node tests. A is the number of nodes with a noise workload and B is the number of `stress-ng` workers per node/pod.

6.3.1. k8s Noise

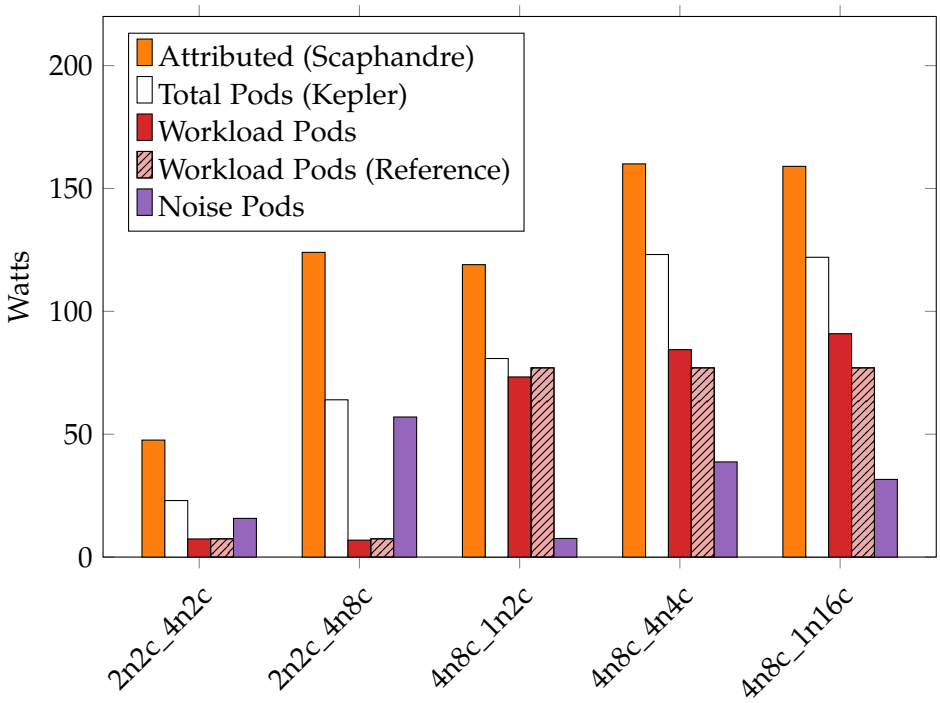
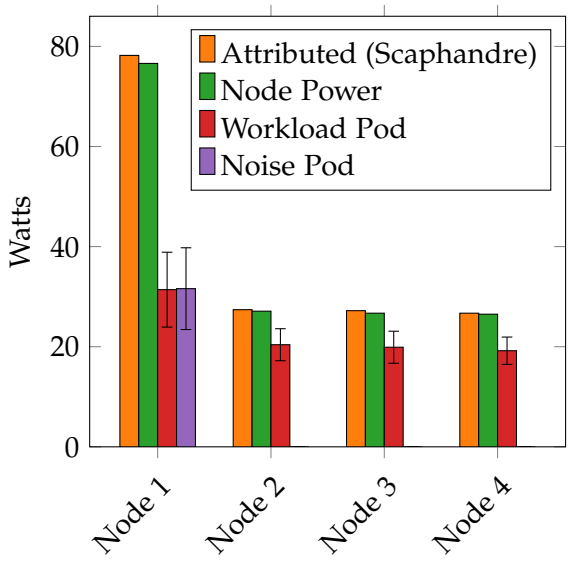


Figure 6.9.: Attribution performance for workloads with Noise in Kubernetes

By running a second resource-intensive (noise) workload in parallel with the main workload, the k8s-level attribution should now divide power between the pods proportionally. The power attributed to the main workload should remain constant when compared to the non-noisy test described above. Total attribution should also remain close to total dynamic power, again accounting for the slight overhead from the abstraction layers. Figure 6.9 shows the attribution

performance during the noisy tests (red), with the white bar presenting the attribution from the same workload without noise. We can see that attribution performance remains consistent across all noise scenarios, with only the highly asymmetric `4n8c_1n16c` scenario showing a slight overestimation for the workload pod. Total attribution performance (including the noise workload) remains unchanged, with values within the previously measured range of 50-80%. While this is still a disappointingly low result, the error at least seems to be well-characterized given the similarity of the results between noise and noise-less workloads.



(Watts)	1	2	3	4
Attributed	78.2	27.4	27.2	26.7
Node Power	76.6	27.1	26.7	26.5
Total Pods	65.7	20.4	19.9	19.2
Workload Pod	34.1	20.4	19.9	19.2
Noise Pod	31.6	0	0	0

Figure 6.10.: Per-node attribution performance for the 4n8c workload with 1n16c noise

Per-node results once again show a strong difference between symmetric and asymmetric load/noise scenarios. Taking a closer look at the asymmetric `4n8c_1n16c` scenario in Figure 6.10 reveals an anomaly on the noisy node: With said node running an 8-worker workload and 16-worker noise pod, one would expect the noise pod to receive proportionally higher power attribution. However, this does not appear to be the case - the noise and workload pod are attributed nearly the same power consumption (34 W vs 32 W) despite their asymmetry. There is also a large deviation in the attribution values, with the workload pod at $\sigma = 7.48$ W and the noise pod reaching $\sigma = 8.17$ W. This is significantly higher than expected and would indicate that Keplers ratio model struggles with asymmetric workloads on the same node. The noise-free nodes perform as established in the previous tests and are unaffected.

Other workload-noise combinations show mixed results, as can be seen in Figure 6.11: The symmetric workload-to-noise test `2n2c_4n2c` performs as expected, with both the workload and noise pods sharing attribution evenly. However noise is again high, casting doubt on the reliability of the attribution. The overwhelming-noise test (`2n2c_4n8c`) shows correct attribution for both the workload and noise pods. While deviation for the noise pods is still high, the results are much more in line with the non-noise benchmarks shown in Figure 6.8. The `4n8c_1n2c` shows good behavior on the workload but suffers from a higher than expected

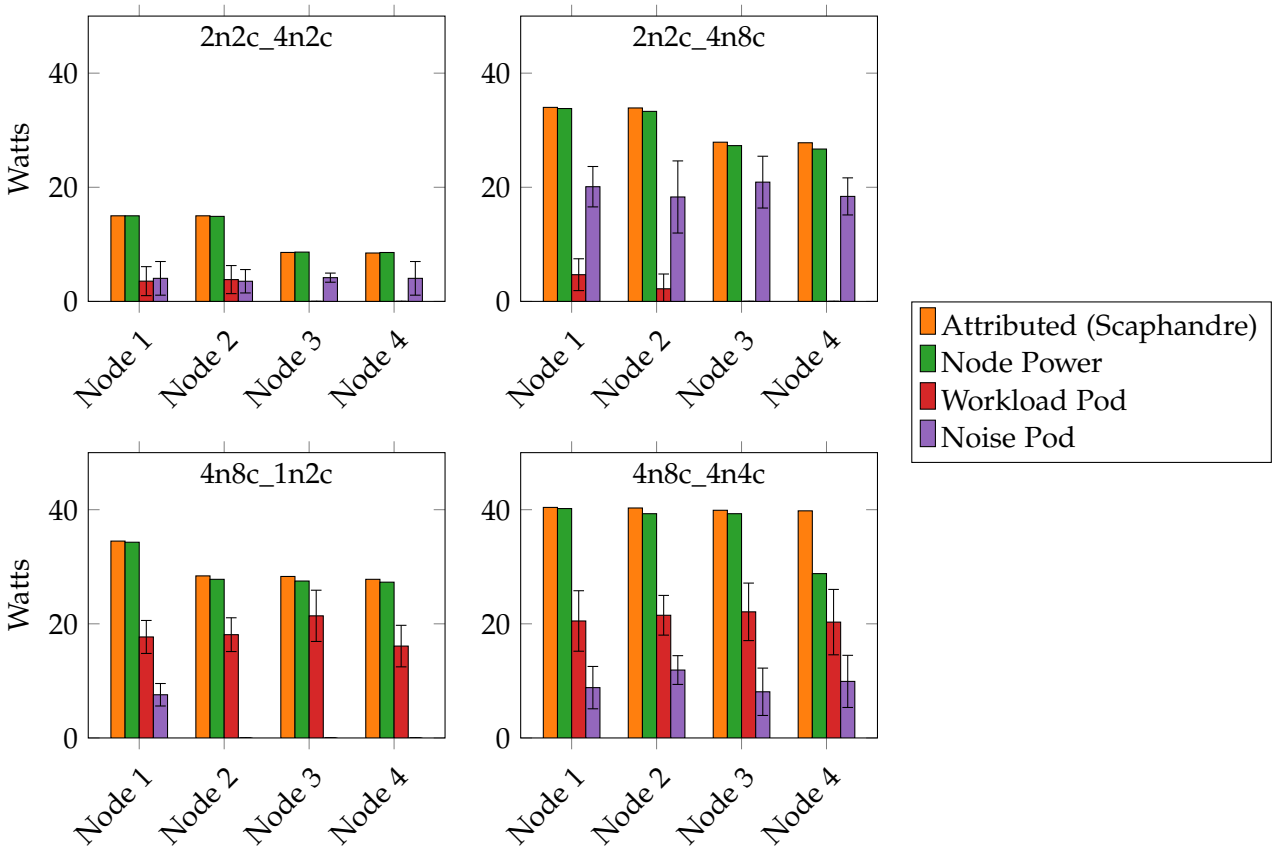


Figure 6.11.: Per-node attribution performance for various workloads with k8s noise

mean for the 2-worker noise pod (compare to the 2-worker pods in the chart above). Finally, the fully symmetric `4n8c_4n4c` test shows good mean values, but very high standard deviation with relative values as high as 50% of the measured value.

Overall, it seems that Keplers ratio power model struggles with reliably attributing power to individual pods in the given scenarios, despite using the same underlying mathematical model as the attribution model defined in chapter 4.

6.3.2. VM Neighbour Noise

Next, several "noise" virtual machines were created on the same hypervisor as the cluster nodes, each with the same resource configuration as the k8s nodes. These VMs were not instrumented and only contained an installation of `stress-ng` on top of the same Ubuntu 22.04 base. Unused noise VMs were shut down for tests where they were not required.

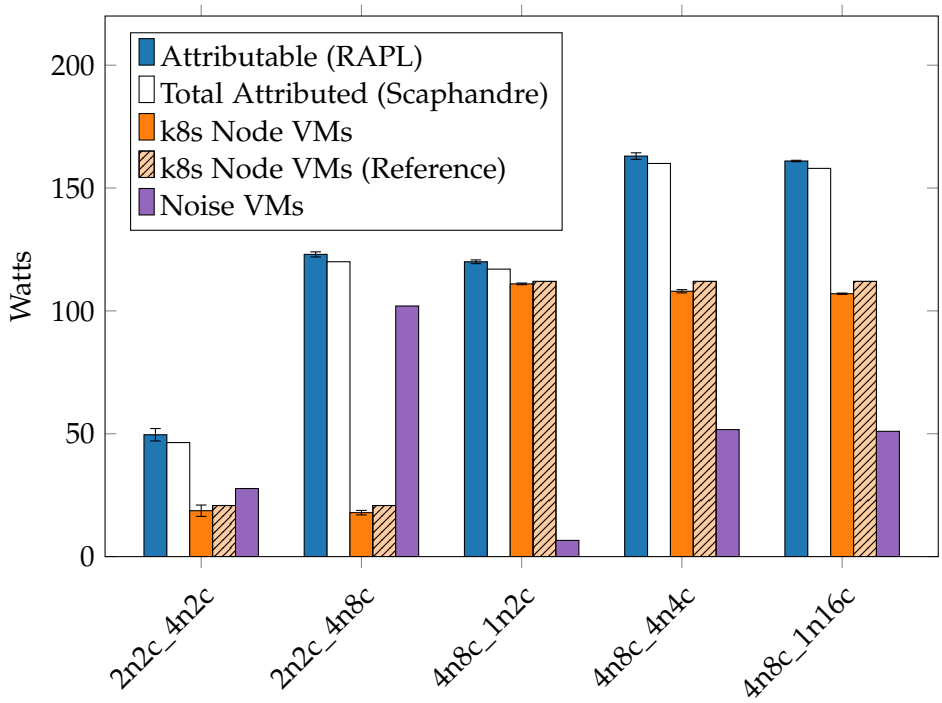


Figure 6.12.: Attribution performance for workloads with Noise in neighbour VMs

Looking at total power attribution in Figure 6.12, we can once again see excellent performance from the host-level attribution model. Attribution values (orange) are very close to the earlier non-noise attribution tests (striped orange) from subsection 6.2.2, indicating that the noisy VMs do not influence attribution significantly. We can observe a slight drop in attribution values across all scenarios, but without further measurements it is difficult to say if this is anything but noise affecting the results.

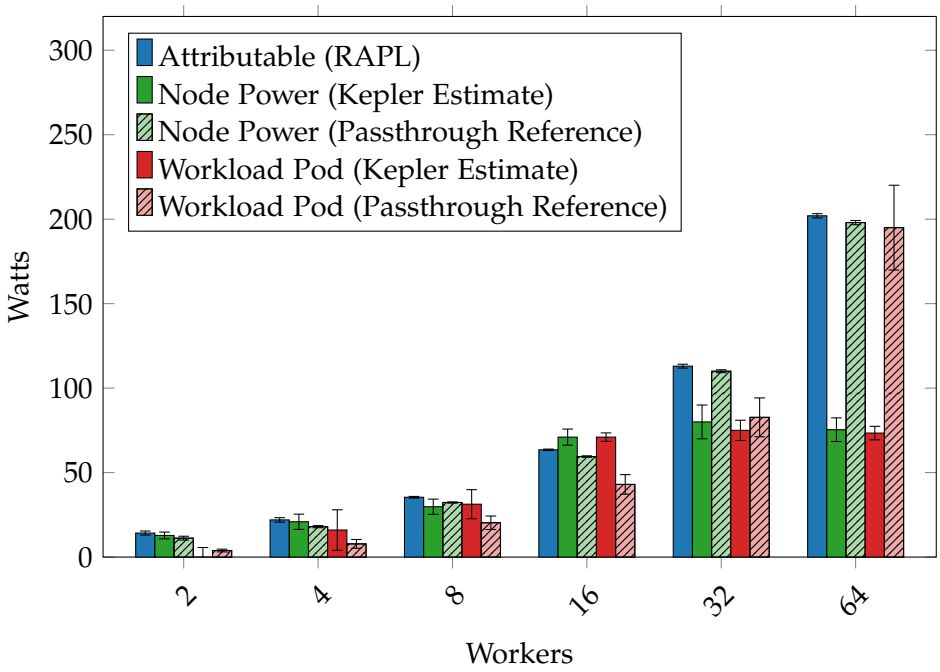
6.4. Kepler Estimation Tests

The final series of tests compares the existing results to Keplers available estimation mode. By configuring Kepler to ignore the passed-through attribution data, it will instead use a pre-trained model to estimate node and component power consumption. It then attributes workload power as normal using its ratio power model. In addition to disabling passthrough support in Kepler, a different set of metrics for estimated node power and pod power was used. For the node power estimate, `platform_joules_total` was used instead of `package_joules_total` and for the pod estimate `container_joules_total` was used instead of `container_package_joules_total`. This was done because because Keplers per-component estimates were found to be unusable and would not have provided for a useful comparison.

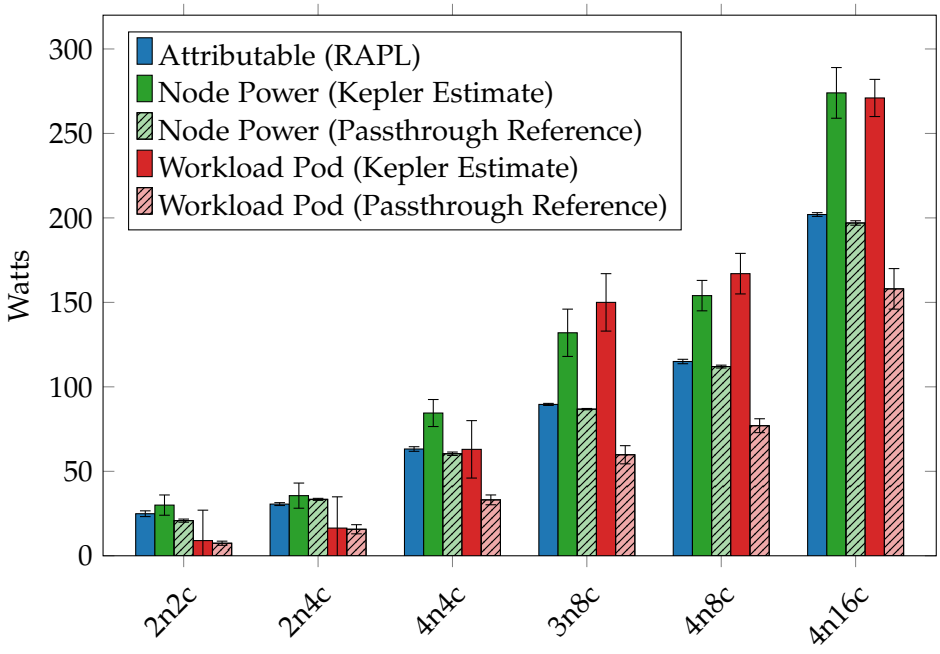
For these tests, handling of static/idle power was also adjusted: Kepler follows a Green-House Gas Protocol guideline to attribute host idle power evenly among running processes [23][37]. This is not an issue for the passthrough setup, as the "host" (the VM) has a miniscule idle consumption of ≈ 1 W as passed through from the host, so attribution values are almost unaffected. However in the estimation setup, Kepler uses its estimated node power as the idle power baseline, potentially inflating attribution values significantly. To address this, Keplers idle power estimate was measured and subsequently removed from each workload pod attribution by subtracting P_{Idle}/n_{Pods}.

Figure 6.13 shows Keplers estimates compared to the previous attribution values from section 6.2 (striped bars). The accuracy of these estimates varies wildly, from severe under-reporting all the way to significant overestimation of power consumption. For the low-load workloads (single-node 2 and 4 workers, `2n2c` and `2n4c`) Kepler is unable to provide a quality attribution at all, with σ larger than the absolute attribution value (note the error bars). In case of the 2-worker test, a mean negative attribution of -2.35 W was measured, along with an unreasonably high $\sigma = 8$ W. It seems that Kepler is unable to compensate for the lack of power passthrough when loads are so low. While this thesis model does under-report power consumption for these scenarios, its results are much more consistent and the error should be possible to correct for once known. Keplers estimation-based performance improves notably in the medium-load scenarios - in fact, Kepler actually outperforms passthrough attribution in some tests: The 8 and 16-worker single-node and `4n_4c` multi-node workloads see Kepler provide attribution values very close to total dynamic power, while the passthrough model still suffers from under-attribution. Consistency of attribution values also improves (with the exception of `4n_4c`), putting Keplers estimates comfortably outside of the passthrough models value range.

High-load estimation performance is drastically different between single and multi-node workloads: For single-worker scenarios, Keplers estimate appears to plateau, thus severely under-attributing power for the 32 and 64-worker tests. For the 128-worker scenario Kepler provided extreme attribution values of over 700 Watts, despite real-world power consumption barely changing. It seems that Keplers power estimate model struggles with loads that exceed the CPUs number of physical cores, but an exact cause for this error was not identified. For these workloads, the passthrough model outcompetes Keplers estimates by a significant



(a) Single-node workload. Reported attribution for the 128-worker scenario was 748 W node power and 747 W pod power.



(b) Multi-node workload

Figure 6.13.: Attribution performance compared to Kepler estimates

margin, more than a factor of two in case of the 64-worker test. While Keplers single-node estimates plateau, multi-node high-load attribution continues to rise with workers and ends up significantly over-estimating power consumption, starting with the `3n_8c` test. This is in contrast to the passthrough model, which under-reports power consumption, albeit to a lesser degree.

The per-node results in Figure 6.14 show that there also is a large variation in attribution values on the node level. Despite being identically configured, Keplers power estimates vary by a factor of over 2 for some workloads, with the `4n_4c` and `4n_8c` workloads showing particularly high variance. This is in stark contrast to the passthrough implementation, which has much more consistent behavior across multiple nodes, as shown in Figure 6.8.

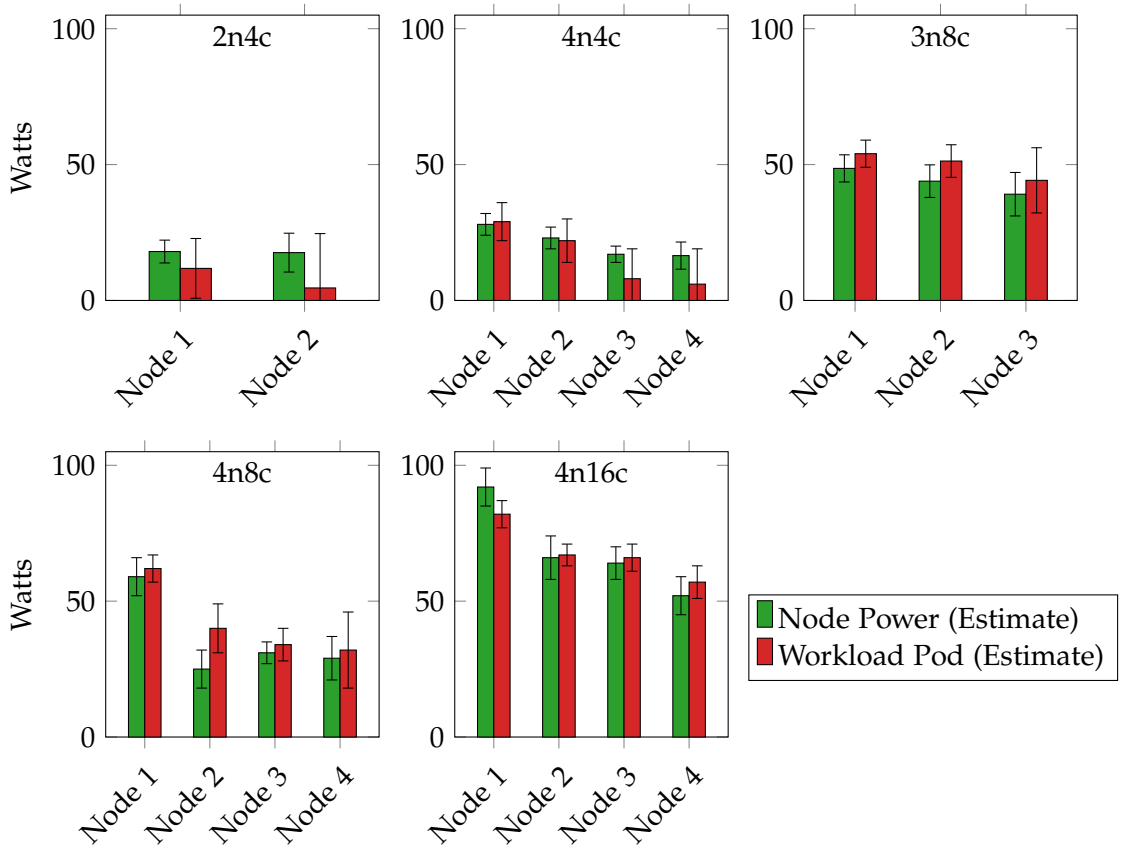


Figure 6.14.: Per-node kepler estimate performance

7. Discussion

To start discussing the gathered findings, we can first turn to the results of the implementation tests gathered in the previous chapter: As set out in section 4.1, the primary goals for the implementation were high precision to enable fine-grained energy attribution and high reliability/consistency of results. On the host layer, the attribution model and implementation is easily able to meet these goals: Not only does the implementation attribute dynamic power nearly perfectly, it also does so confidently with very low deviation. Observed deviation was $\sigma \leq 1$W on most of the tests, easily precise enough to validate application optimizations. This deviation combined with consistent performance throughout the noise scenarios leads to the conclusion that the host implementation is also able to fulfill the reliability requirements.

Sadly, results were a little more mixed on the guest/Kubernetes layer: Despite implementing the same attribution model as described in this thesis, Kepler was unable to match the accuracy shown by the host implementation. The guest implementation consistently underperformed with regard to attribution, with higher-load scenarios showing slightly better results. There are a number of possibilities for why this under-performance occurred:

First, the high deviation observed in these tests suggests that the guest implementation is suffering from noise in at least one area. The unusual node power readings (see Figure 6.4) could hint at the passthrough mechanism being at fault, with the update interval being a possible source of inaccuracy: To reduce CPU load on the host, the passthrough attribution data is only updated every few seconds, compared to the near-instantaneous rate of a real hardware RAPL interface. While the actual values were confirmed to be correct, the coarser increments may be throwing off Keplers attribution implementation. Another possibility for noise ingress is Keplers method for gathering process statistics through eBPF, compared to the `/proc`-based host-level implementation. While eBPF should in theory be able to provide very accurate process metrics, there is a possibility that the used virtualization setup prevented high-resolution measurement of data, causing the poor Kepler performance [23].

Alternatively, the poor guest-level performance could be the result of aggregation difficulties caused by a workload being spread over multiple processes: Since `stress-ng` creates a new worker process for each CPU worker, the guest implementation must properly aggregate the utilization of all these processes into a single attribution value for the pod. This is in contrast to the host layer, where each VM is represented by a single process. It is possible that this aggregation presents additional challenges for the guest layer that the demonstration implementation is unable to deal with.

To identify and subsequently eliminate the source of this guest-side measurement inaccuracy, more implementation work will be necessary. Possible pathways include modifying Keplers attribution implementation to exclude eBPF as a noise source, or retooling the attri-

bution passthrough to allow for more fine-grained data ingress on the guest side.

Of course, the main goal in performing these tests was to answer RQ2 and RQ3. For RQ2, we can confidently say that construction of accurate, hardware-backed attribution is possible in the given scenario, the aforementioned guest-level issue in the demonstration implementation notwithstanding. This is backed up by the noise tests, which show that the implementation is resilient to noise introduced by multi-tenant scenarios. We can conclude that the basic idea of multi-layered attribution is sound and able to provide very high accuracy, both in terms of precision and reliability. Having said that, there are several issues with the concept that must be resolved in future work before it can become truly practical:

First, the demonstration implementation only covered CPU load and ignored other components in the system. This was partially done for scope reasons, but also because modern CPUs are unique in having both high-quality energy consumption *and* utilization data, both of which are required for attribution to work well. Different components have varying levels of instrumentation available, ranging from solid to almost nonexistent. GPUs are of particular interest for future work given their prevalence in Machine Learning/AI workloads. The Kepler project already has some support for reading and attributing energy consumption for Nvidia cards, but adding support for these components across all layers would still be a non-trivial task, especially with regards to passthrough.

Second, even the CPU attribution used by the implementation is imperfect, as it ignores nonlinear power scaling effects described in subsection 4.2.1, along with assuming perfect proportionality betwen CPU time and power draw. The issue here lies in finding a good metric for the horizontal position of a given process on the power curve described in subsection 4.2.1. More research is needed on this subject.

To answer RQ3, we can turn to the estimation test results. While the results for overall accuracy were mixed (in part due to the guest-side noise issue), the passthrough implementation performed significantly better with regards to consistency: The results show that while there is an attribution error in the passthrough implementation, it is rather well-defined and consistent. Kepler meanwhile both over- and underestimated energy consumption depending on the scenario, providing little confidence in its estimates consistency. Ultimately, the passthrough implementation appears to be better-suited for software optimization tasks than Keplers estimates, given its high precision and much lower noise floor.

Keplers poor estimation performance may be the result of its default attribution model being a poor fit for the chosen target platform. In fact, the Kepler project puts significant emphasis on enabling the use of models trained for specific hardware configurations, enabled by tools such as the Model Server and its training pipeline [24]. It is possible that a model trained for a specific platform might be able to match a hardware-backed solution in overall performance, but the creation of such a model is a nontrivial task. The availability of more pre-trained models along with increased accessibility of the training process could help fill this gap to some extent. Even so, pre-trained models will still struggle with some aspects of attribution, such as idle/static power/energy: Even with a good estimate for E_S, a guest

cannot know how many other guests are present on a given host and is thus unable to divide its idle power estimate according to the methods described in subsection 4.2.4 [23].

Overall, the findings in this thesis support the idea that hardware-backed energy attribution is not just possible in cloud-computing scenarios, but also feasible today, using existing tooling (albeit with some modifications). The results prove that the model presented in chapter 4 is valid - the difficulty now lies in the construction and adoption of quality implementations. For the short term, a robust host-layer implementation provided as service to guests could present the most pragmatic approach for cloud providers. By providing attribution as an external service there is less need to touch security-critical virtualization components, while also allowing guests the flexibility in if and how they want to ingest attribution data for their particular workload. Keplers trained models may also provide a viable alternative, although this requires the availability of bare-metal instances equivalent to the virtualization hosts.

In the long term, one can einvision a cloud architecture designed with sustainable computing in mind, where energy data is attributed on the host, before being passed to the guest using true hardware-level emulation of existing hardware energy interfaces such as RAPL. This would present each tenant and each guest with easily accessible, standardized energy data that could then be further attributed by the tenant as needed. This could be considered the ultimate goal of this research and possible future work in this area: increasing the accessibility of energy data for the purposes of reducing resource consumption. After all, more efficient usage of resources is at the core of green computing and the move towards a more sustainable future.

On top of the possible future research opportunities identified above, there are also a few more avenues for improving the results obtained by this thesis: First, the model and implementation are currently only tested on a single platform. While significant effort was taken to ensure that the results are representative, actual hardware verification on different systems would further increase confidence in the model or highlight potential issues with it. Related, the models performance should also be verified with real-world workloads on top of the synthetic load test performed in this thesis. While the simplifications made during the derivation of CPU utilization (see subsection 4.2.1) should hold for other workloads, additional verification would still be beneficial. Finally, extending the implementation to perform static power attribution according to subsection 4.2.4 could allow for further insights into how the different approaches affect attribution results inside the guests, especially in noisy multi-tenant environments.

A. Appendix

A.1. Materials Repository

The materials covered in this thesis, including implementation code, deployment automation and raw results data are available in a public Git repository for further inspection. This repository can be found at `https://github.com/maxhoesel/master-thesis-material`, or using the following QR Code:

List of Figures

List of Tables

Acronyms

k8s Kubernetes 13, 17, 19, 22, 31, 33, 34, 36, 47, 55, 58, 60, 64, 66, 67, 71

PSU Power Supply Unit 2, 7, 10

RAPL Running Average Power Limit 2, 3, 8–11, 17, 18, 21, 22, 24, 28, 29, 32, 37–40, 42–44, 47, 49, 50, 52, 57, 58, 71, 73

VM Virtual Machine 3, 4, 11–13, 16–19, 22, 23, 31–34, 36–39, 42–45, 47, 49, 50, 52, 58, 60

Bibliography

[1] CBRE. "Global Data Center Trends 2023", Accessed: Mar. 26, 2025. [Online]. Available: https://www.cbre.com/insights/reports/global-data-center-trends-2023.

[2] CBRE. "Global Data Center Trends 2024", Accessed: Mar. 26, 2025. [Online]. Available: https://www.cbre.com/insights/reports/global-data-center-trends-2024.

[3] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates", *Science*, vol. 367, no. 6481, pp. 984–986, Feb. 28, 2020, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aba3758. Accessed: Feb. 18, 2025. [Online]. Available: https://www.science.org/doi/10.1126/science.aba3758.

[4] "Constellation to Launch Crane Clean Energy Center, Restoring Jobs and Carbon-Free Power to The Grid", Accessed: Feb. 18, 2025. [Online]. Available: https://www.constellationenergy.com/newsroom/2024/Constellation-to-Launch-Crane-Clean-Energy-Center-Restoring-Jobs-and-Carbon-Free-Power-to-The-Grid.html.

[5] X. Jin, F. Zhang, A. V. Vasilakos, and Z. Liu. "Green Data Centers: A Survey, Perspectives, and Future Directions". arXiv: 1608.00687 [cs], Accessed: Feb. 18, 2025. [Online]. Available: http://arxiv.org/abs/1608.00687, pre-published.

[6] Uptime Intelligence, "Executive summary: Uptime Institute Global Data Center Survey 2023", 2024.

[7] Intel, *Intel 64 and IA-32 Architectures Software Developers Manual*, Intel, 2024.

[8] Intel. "2024.3 IPU - Intelő Processor RAPL Interface Advisory", Intel, Accessed: Feb. 18, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01103.html.

[9] C. Centofanti, J. Santos, V. Gudepu, and K. Kondepu, "Impact of power consumption in containerized clouds: A comprehensive analysis of open-source power measurement tools", *Computer Networks*, vol. 245, p. 110 371, May 1, 2024, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2024.110371. Accessed: Sep. 24, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128624002032.

[10] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, "Methodological Guidelines for Measuring Energy Consumption of Software Applications", *Scientific Programming*, vol. 2019, no. 1, p. 5 284 645, 2019, ISSN: 1875-919X. DOI: 10.1155/2019/5284645. Accessed: Oct. 27, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2019/5284645.

[11] Scaphandre. "About RAPL domains", Accessed: Feb. 18, 2025. [Online]. Available: https://hubblo-org.github.io/scaphandre-documentation/explanations/rapl-domains.html.

[12] M. Liu. "ARM-based Server Penetration Rate to Reach 22% by 2025 with Cloud Data Centers Leading the Way, Says TrendForce", Accessed: Feb. 18, 2025. [Online]. Available: https://www.trendforce.com/presscenter/news/20220329-11178.html.

[13] The Linux Kernel documentation. "GPU Power/Thermal Controls and Monitoring", Accessed: Feb. 18, 2025. [Online]. Available: https://docs.kernel.org/gpu/amdgpu/thermal.html.

[14] Intel. "Running Average Power Limit Energy Reporting CVE-2020-8694,...", Accessed: Feb. 18, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[15] Open Container Initiative. "Open Container Initiative", Accessed: Mar. 1, 2025. [Online]. Available: https://opencontainers.org/.

[16] "Capabilities(7) - Linux manual page", Accessed: Mar. 1, 2025. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html.

[17] S. Choochotkaew, C. Wang, H. Chen, T. Chiba, M. Amaral, E. K. Lee, and T. Eilam. "A Robust Power Model Training Framework for Cloud Native Runtime Energy Metric Exporter". arXiv: 2407.00878 [cs], Accessed: Mar. 2, 2025. [Online]. Available: http://arxiv.org/abs/2407.00878, pre-published.

[18] P. Pathania, R. Mehra, V. S. Sharma, V. Kaulgud, S. Podder, and A. P. Burden, "ESAVE: Estimating Server and Virtual Machine Energy", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Oct. 10, 2022, pp. 1–3. DOI: 10.1145/3551349.3561170. arXiv: 2209.07394 [cs]. Accessed: Sep. 20, 2024. [Online]. Available: http://arxiv.org/abs/2209.07394.

[19] A. E. Husain Bohra and V. Chaudhary, "VMeter: Power modelling for virtualized clouds", in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470907. Accessed: Sep. 20, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5470907.

[20] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, "Process-level power estimation in VM-based systems", in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, New York, NY, USA: Association for Computing Machinery, Apr. 17, 2015, pp. 1–14, ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741971. Accessed: Sep. 22, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/2741948.2741971.

[21] H. Hè, M. Friedman, and T. Rekatsinas, "EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy", in *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, Boston MA USA: ACM, Jul. 9, 2023, pp. 1–8, ISBN: 979-8-4007-0242-6. DOI: 10.1145/3604930.3605716. Accessed: Sep. 19, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3604930.3605716.

[22] J. Phung, Y. C. Lee, and A. Y. Zomaya, "Lightweight Power Monitoring Framework for Virtualized Computing Environments", *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 14–25, Jan. 2020, ISSN: 1557-9956. DOI: 10.1109/TC.2019.2936018. Accessed: Sep. 22, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8811765.

[23] Kepler Project. "Kepler Deep Dive", Accessed: Mar. 2, 2025. [Online]. Available: https://sustainable-computing.io/usage/deep_dive/.

[24] Kepler Project. "Kepler Model Server Architecture", Accessed: Mar. 2, 2025. [Online]. Available: https://sustainable-computing.io/kepler_model_server/architecture/.

[25] Kepler Project. "eBPF in Kepler", Accessed: Mar. 2, 2025. [Online]. Available: https://sustainable-computing.io/design/ebpf_in_kepler/.

[26] Scaphandre. "How scaphandre computes per process power consumption", Accessed: Mar. 2, 2025. [Online]. Available: https://hubblo-org.github.io/scaphandre-documentation/explanations/how-scaph-computes-per-process-power-consumption.html.

[27] Scaphandre. "Qemu exporter", Accessed: Mar. 2, 2025. [Online]. Available: https://hubblo-org.github.io/scaphandre-documentation/references/exporter-qemu.html.

[28] S. Desrochers, C. Paradis, and V. M. Weaver, "A Validation of DRAM RAPL Power Measurements", in *Proceedings of the Second International Symposium on Memory Systems*, Alexandria VA USA: ACM, Oct. 3, 2016, pp. 455–470, ISBN: 978-1-4503-4305-3. DOI: 10.1145/2989081.2989088. Accessed: Jan. 2, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2989081.2989088.

[29] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An Energy Efficiency Feature Survey of the Intel Haswell Processor", in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 896–904. DOI: 10.1109/IPDPSW.2015.70. Accessed: Feb. 18, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7284406.

[30] AMD. "AMD EPYC 7773X", AMD, Accessed: Mar. 2, 2025. [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7773x.html.

[31] "Top(1) - Linux manual page", Accessed: Mar. 2, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/top.1.html.

[32] The Linux Kernel documentation. "The /proc Filesystem", Accessed: Feb. 18, 2025. [Online]. Available: https://docs.kernel.org/filesystems/proc.html.

[33] E. Vasilakis, "An instruction level energy characterization of arm processors", *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450*, 2015. Accessed: Feb. 18, 2025. [Online]. Available: https://www.ics.forth.gr/carv/greenvm/files/tr450.pdf.

[34] R. Sen and D. A. Wood, "Energy-Proportional Computing: A New Definition", *Computer*, vol. 50, no. 8, pp. 26–33, 2017, ISSN: 1558-0814. DOI: 10.1109/MC.2017.3001248. Accessed: Mar. 27, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7999152/?arnumber=7999152.

[35] E. Borba, E. Tavares, and P. Maciel, "A modeling approach for estimating performance and energy consumption of storage systems", *Journal of Computer and System Sciences*, vol. 128, pp. 86–106, Sep. 1, 2022, ISSN: 0022-0000. DOI: 10.1016/j.jcss.2022.04.001. Accessed: Jan. 3, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000022000290.

[36] The Linux Kernel documentation. "I/O statistics fields", Accessed: Feb. 18, 2025. [Online]. Available: https://docs.kernel.org/admin-guide/iostats.html.

[37] Global Sustainability Initiative, *ICT Sector Guidance built on the GHC Protocol Product Life Cycle Accounting and Reporting Standard*, Jul. 2017. Accessed: Feb. 20, 2025. [Online]. Available: https://ghgprotocol.org/sites/default/files/2023-03/GHGP-ICTSG%20-%20ALL%20Chapters.pdf.

[38] The Linux Kernel documentation. "Linux KVM Hypercall", Accessed: Mar. 28, 2025. [Online]. Available: https://docs.kernel.org/virt/kvm/x86/hypercalls.html.

[39] Amazon Web Services. "Retrieve security credentials from instance metadata", Accessed: Mar. 28, 2025. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-metadata-security-credentials.html.

[40] Kubernetes. "Accessing the Kubernetes API from a Pod", Kubernetes, Accessed: Mar. 28, 2025. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/access-api-from-pod/.

[41] Kubernetes. "Volumes", Kubernetes, Accessed: Mar. 21, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/.

[42] CNCF Sustainable Computing Group, *Kepler Helm Chart*, Sustainable Computing, Mar. 3, 2025. Accessed: Mar. 21, 2025. [Online]. Available: https://github.com/sustainable-computing-io/kepler-helm-chart.