university of groningen

faculty of science and engineering

Master's Thesis

Estimating Container-level Power Usage in Kubernetes

Bjorn Pijnacker s4106164

Under the supervision of
prof. Vasilios Andrikopoulos∗ and dr. Brian Setz†

November 10th, 2024

To fulfill the requirements for the degree of Master of Science in Computing Science at the University of Groningen

∗Computing Science, University of Groningen
†Lead of Digital Lab, University of Groningen

Abstract

Energy efficiency in cloud computing, and specifically Kubernetes, has been a major topic of research in the past years as cloud datacenters grow and the importance of minimizing carbon output increases. Much of this research focuses on the total energy usage of a Kubernetes cluster and attempts to optimize this by various methods such as energy-aware scheduling and datacenter- or cluster-wide metrics, without regard for individual workloads. A lesser explored aspect which can provide useful insights into Kubernetes power usage is a measure of power usage at the Kubernetes container level; thereby providing insight into the power usage for each workload in a cluster.

In this research project, we experimentally evaluate the state-of-the-art Kubernetes power estimation tooling: the Cloud-Native Computing Foundation's Kepler. This tool is evaluated on datacenter-grade hardware where its total Kubernetes node measurements and its container power attribution for each of the available configurations and available external sources is considered. We find that this tool does not produce satisfactory power usage metrics in regard to container power attribution and that there are significant limitations in several of the available node power measurement strategies.

To combat the limitations in Kepler, we create our own tool named KubeWatt based on a recently introduced power mapping model. The architecture and implementation of KubeWatt are discussed along with an experimental validation of its features and measurements. During this work, several limitations as well as some interesting findings are discussed which may provide avenues for future research.

Contents

1. Introduction

Datacenters and cloud computing are significant power users. In 2021, cloud computing accounted for approximately 1 % of global power usage, with it estimated to reach 8 % before 2030 [1]. As the leading container orchestration platform, Kubernetes-based workloads play a significant role in this power consumption. According to a 2022 Red Hat report, up to 70 % of IT organizations use Kubernetes in some way [2]. As such, the energy usage and efficiency of applications running in Kubernetes is of significant interest, since significant savings can be made for both cost and carbon emissions in datacenter computing.

Existing research into Kubernetes energy usage focuses on two specific metrics: waste and efficiency. Waste, in the context of computing, is defined as the metric of variation of the allocated resources minus the used resources [3]. In short: waste resources are allocated but go unused; as such, they are wasted. When optimizing deployments to minimize waste, the emphasis lies on achieving 100 % resource utilization as least wasteful. Minimizing energy waste can also lead to reduced power consumption over all. Allocated resources that go unused are not free, as overhead of these resources existing also accounts for some power usage. By more intelligently allocating resources and using as much of these resources as possible, the impact of this overhead is minimized.

The second metric, efficiency, is the measure of energy usage per 'unit' throughput. In 2015, researchers have investigated the efficiency of workloads by relating power divided by throughput against CPU load [4]. The goal of this study was to find the sweet spot of resource utilization where the system operates most efficiently, that is, the most work completed per Joule. They found that striving for 100 % load is not the most efficient use of power that is available to the system. They show a curve of efficiency based on load that recommends approximately 80 % load as the most energy efficient in their experiments. Notably, for a static load, this would yield a small amount of waste as it is defined above.

Based on results from these papers it is possible to provision computers and workloads to optimize them in terms of energy efficiency with a relatively small amount of waste. However, this is only shown for workloads as discussed in [4]: single machines. Many workloads these days do not run on a plain operating system and are rather deployed in Kubernetes. Since a single node in a Kubernetes cluster can run many workloads at once, optimizing its load becomes a more difficult problem. A paper from 2022 [5] has found that in the cloud—specifically Azure—VMs are often running far below the most efficient load. As such, we can assume most Kubernetes instances that run on similar VMs, are likely underprovisioned as well. A significant chunk of deployments actually utilize below 10 % of CPU time, something this paper calls a "red VM" [5]. This phenomenon is called cloud overprovisioning and is a significant waste of resources and drives higher costs for the user of the provisioned infrastructure.

Existing solutions attempt to optimize Kubernetes cluster or even datacenter power usage as a whole [6], [7]. This has yielded some promising results. A relatively unexplored aspect of Kubernetes power optimization is the energy usage at the deployment level. Providing the responsible person with an overview of power used for their applications may provide a useful entry point for energy use optimization, at the very least starting with awareness.

There are existing tools which attempt to estimate container-level power usage for Kubernetes or for generic containers. One such tool is SmartWatts [8]. According to the related GitHub project, SmartWatts is based on power models which relates total power usage to resource usage of containers [9]. It measures total energy consumption using RAPL and attributes this to specific processes using HwPC (Hardware Performance Counters). RAPL stands for *Running Average Power Limit* and is an Intel processor feature. HwPC uses RAPL to provide more granular insights into CPU power consumption.

While SmartWatts seems like a promising tool for power consumption analysis, its main drawback is in its use of RAPL to obtain its metrics, which is unsupported in virtualized environments: something which the majority of the cloud landscape uses. In the case of Kubernetes, we may instead consider Kepler [10]. Kepler is the Cloud-Native Computing Foundation's sandbox level[1] Kubernetes operator which can export energy

[1]Sandbox-level is the first of the three CNCF project maturity levels [11]

statistics to Prometheus[2] on a pod-granular level. While Kepler does use RAPL, it uses it to train a runtime model which does not depend on RAPL. This way, Kepler can be used even in (virtualized) systems without access to hardware counters. Note, however, that Kepler will use such counters on systems that provide them, such as bare-metal Kubernetes deployments, to enhance the accuracy of measurements. Kepler is even able to use other power sources, such as server management interfaces using the Redfish API standard [12], when available. In the following, we present Kepler in more depth.

Kepler

Kepler stands for "**K**ubernetes-based **E**fficient **P**ower **L**evel **E**xporte**r**". It estimates power consumption in Kubernetes, specifically at the process, container and pod-level. At its core, Kepler uses various resource utilization metrics obtained using an eBPF program, a technology to allow running programs in the Linux kernel [13], and collects various real-time power consumption metrics such as RAPL for CPU and DRAM, NVML for NVIDIA GPU power, ACPI, Redfish or IPMI for platform power or regression-based models when no real-time power metrics are available [14].

Using utilization metrics together with platform and component power usage data, Kepler can estimate the amount of energy each process, container or pod uses. It does this by dividing total used energy into idle- and dynamic-mode energy, using, what Kepler calls, their "ratio power model". *Idle power* is used by a system regardless of resource utilization. *Dynamic power* is power which can be directly related to resource utilization and thereby attributed to a specific resource-using process. The idle power is divided over all processes or containers relative to the number of total containers running as described by the Greenhouse Gas protocol guideline [15]; the dynamic power is divided over processes based on resource utilization [16].

Kepler can run in multiple configurations depending on the available information of a system. In its most basic configuration, Kepler estimates power consumption using a pre-trained model, which estimates energy usage based on available hardware counters such as CPU utilization. A user of Kepler may train their own power consumption model and use this instead of the provided one. The authors of Kepler recommend doing this, since any model is trained for a specific system and a custom model will be most accurate. If available, Kepler can also use RAPL to obtain platform or component power consumption data, or use a Redfish API integration to obtain platform data directly from hardware-level components such as the power supply through a server management interface like Dell iDRAC.

Kepler's metrics are divided into node-metrics and container-metrics. The node metrics are, for each node of the Kubernetes cluster, divided between the core, dram, package, platform and uncore components. Depending on the configuration that Kepler runs in, each of the metrics for these components may be derived from a different source. For example, when running Kepler using Redfish, platform power is derived from Redfish while other components' power is derived from RAPL. For the containers, Kepler has similar power metrics available and additionally has certain resource utilization metrics for each container [17].

Previous papers have used Kepler, though none of these validate its accuracy or suitability for the task it performs. Gudepu *et al.* [18] use Kepler to obtain power measurements and indicate it produces similar results to Scaphandre, a tool which predicts power usage per resource; however, neither tool is validated using a source of ground truth. Additonally, the authors do not show the Kepler results. It is only mentioned that the results are similar to Scaphandre. Soldani *et al.* use Kepler as a demonstration of an eBPF use-case. While they show a dashboard of energy measurements, these are not validated as being accurate [19]. Centofanti *et al.* [20] investigate Kepler in comparison to Scaphandre and s-tui; however, for Kepler they only consider the default configuration. They find that all tools performed quite differently in their tests and conclude that further research is required to increase the robustness of these tools. Andringa [21] also investigates Kepler, finding that Kepler's metrics are not accurate as total cluster power is concerned. The author does not investigate per-container attribution further.

The previous work is not completely sufficient to validate Kepler's accuracy. To be convinced of its accuracy, we must validate that the total power reported by Kepler matches the actual power that a system under test uses at

[2]A cloud monitoring system that has graduated CNCF maturity level: https://www.cncf.io/projects/prometheus/

the wall outlet, and we must validate that the power reported per container matches closely to what is expected taking into account resource usage by that workload.

Problem Definition & Contributions

To target energy efficiency and waste in Kubernetes at the deployment level, we must first have (real-time) insight into energy usage of Kubernetes deployments. In this work we aim to make a contribution towards Kubernetes energy measuring. We therefore ask the following question as the main guidance in our research:

> How can we accurately measure or estimate the power usage of Kubernetes containers based on external measurements?

Moreover, as the current state-of-the-art focuses on observability (as we will discuss in Section 2), we will look for a method to measure or estimate the power usage of Kubernetes containers that can export these metrics to an observability platform. An observability platform is a centralized location in which, among others, metrics, logging and traces of cloud environments can be stored. The goal of observability is to give insight into complex systems and to allow for troubleshooting and obtaining insights from system data [22].

The question will be answered in two parts: first, we will evaluate state-of-the-art tool Kepler, which promises to have solved the aforementioned problem. For Kepler, we pose the following additional research questions:

- KPRQ1 How accurate are Kepler's total node measurements compared to a ground truth?
- KPRQ2 How (well) does Kepler attribute energy usage to containers on the node?
- KPRQ3 How do the different Kepler configurations affect the accuracy of the reported metrics?

After evaluating Kepler, we build our own tool named KubeWatt. This tool will serve the same purpose: to measure container-level power usage in Kubernetes. For this tool we ask similar questions:

- KWRQ1 How accurate are KubeWatt's total node measurements compared to a ground truth?
- KWRQ2 How (well) does KubeWatt attribute power usage to containers on the node?
- KWRQ3 How accurately can KubeWatt's base initialization mode report the static power value?
- KWRQ4 How accurately can KubeWatt's bootstrap initialization mode estimate the static power value?

Note that these questions refer to certain modes in which KubeWatt can run. These are discussed in more detail in Section 6.

The rest of this work is structured as follows: in Section 2 we discuss related work, apart from what has already been discussed above, to give context to this research project and to provide an overview of the state of the art. In Section 3, we introduce the experimental setup that is used for all experiments, parts of which we validate in Section 4. In Section 5, the Kepler evaluation is performed. Then, in Sections 6 and 7 KubeWatt's architecture and evaluation are discussed respectively and in Section 8 the research project is concluded and our main research question is answered.

2. Related Work

Energy use in cloud computing and Kubernetes, and its related concepts of energy efficiency and waste, have been the topic of many previous research papers which look at a variety of concepts such as efficiency in terms of CPU load on bare-metal systems [4], using high-level monitors in Kubernetes to obtain information on carbon footprint [23], and energy efficient scheduling for Kubernetes. Besides power measuring and automatic optimization of power use, research into (over)provisioning is also related, since the overprovisioning of resources can lead to significant waste in resources and energy.

In this section we discuss the state-of-the-art of energy efficiency and monitoring in cloud computing and Kubernetes and place our work in its context. It should serve to the reader as a contextualization of this work and will outline how this research project builds and improves on existing research.

Existing papers which concern themselves energy usage in Kubernetes can be globally divided into two categories: (1) energy-aware scheduling and automatic optimization, and; (2) measuring energy consumption and

related metrics. We discuss both of these separately in Section 2.1 and Section 2.2, respectively. Moreover, we discuss the related work of investigating and reducing overprovisioning in Section 2.3. Lastly, we take a brief look at observability in Section 2.4.

2.1. Energy-aware Scheduling and Automated Optimization

The previously completed work in this category aims to optimize the Kubernetes scheduler such that the cluster as a whole is more energy efficient. The exact subject and goal of the research, and methods used by these schedulers, of course, differs between papers. For example, Douhara *et al.* built a workload allocation optimizer (WOA-scheduler) and load balancer (WOA-LB) which allocate workloads such that the energy usage of the Kubernetes cluster under test is reduced by close to 10 %. It uses a neural network-based power consumption reduction function to accomplish this task. The authors do mention a drawback to their energy reduction strategy, namely higher response times for requests [6].

Another strategy entirely is chosen by Kuljeet Kaur *et al.* Instead of a machine learning-based approach they have formatted their scheduling problem using integer linear programming based on a multiobjective optimization problem. It is also not particularly concerned with general-purpose clusters and focuses instead on industrial Internet of Things. Its scheduler, named KIEDS, proposes a reduction in energy usage of approximately 14.5 %, when comparing it to data from a Google compute cluster [24].

A more recent approach from 2023 is proposed by Ghafouri *et al.* Their scheduling approach, Smart-Kube, uses deep reinforcement learning to learn about the cluster it is running on and provide a scheduler that is custom-tailored to the cluster without manual configuration. They perform tests in terms of three "strategies", these being *consolidation* which prioritizes energy saving, *fairness* which prioritizes balancing of utilization between nodes, and *balance* which balances the previous two. While the authors do not specify specific energy saving numbers, they mention that their scheduler is very good at consolidation and that this indicates good energy saving potential [7].

Something that most if not all the papers in this category have in common is that they measure the energy used by some cluster as a whole; however, we should consider a cluster in a more granular view as well. Workloads on a Kubernetes cluster can be quite heterogeneous in nature, and therefore considering the energy usage of smaller components might provide useful insights which are not otherwise obtainable.

What all custom schedulers will have in common is that, while they allow more efficiently utilization of resources using strategies such as consolidation, they still must abide by the resource requests and limits that workloads define [25]. This means that they cannot properly target workload-level overprovisioning without violating certain kubelet enforcements. We think an approach is needed that allows optimizing individual deployments by targeting those that are the least energy efficient directly. Naturally this means we need a way to calculate the energy efficiency of an application running in Kubernetes, for which our work: obtaining a way to measure energy usage of a Kubernetes container, is a precursor.

2.2. Energy measuring and Efficiency

In terms of energy measuring there are many aspects to consider. Many papers introduce tools which may be based on hardware or software, which measure a system as a whole or measure the system in a more granular way. Moreover, tools are aimed specifically at bare-metal systems, virtualized systems or specific container environments. In this section, we consider some papers which look specifically at measuring energy and energy efficiency in virtualized environments, containerized applications, and Kubernetes.

Consider the paper by Lemoine [26]. This paper investigates the energy efficiency of Kubernetes when looking at the Horizontal Pod Autoscaler in a specific deployment scenario named the Nominal:Backup Protection scheme. The paper defines a metric of energy efficiency as

$$EE = \frac{\text{data\_rate}}{\text{power\_consumption}}. \tag{1}$$

In fact, this is the same definition as employed by [4]. They continue to define metrics as related to a load balancing pool of pods and investigate the characteristics of power usage related to horizontal scaling behavior. To

define pod-based power consumption the authors make some optimistic assumptions regarding the utilization and balancing of work over pods, stating that they assume service load is equally balanced over pods in a load balancing pool. They then analyze these for multiple horizontal scaling parameters and provide guidance to engineers attempting to introduce horizontal scaling for the aforementioned N:1 protection setup. It is noteworthy to mention that the authors do not actually measure power but rely completely on their power modelling.

More granular energy measurement techniques are also proposed in some papers. In [27], a model is proposed to increase reliability of fog computing[3] that uses Kubernetes. While reliability is not of particular concern to us, the way the authors measure power consumption per pod is interesting: A power metric is obtained from different statistics and counters of the system running Kubernetes. This provides a global estimate of power used by the system. By multiplying this estimate with time spent, a measure of energy use is obtained. To then obtain a measure per container/pod they consider the rate of pod instantiation within the system. While this does not give the exact power used by a system, it does provide a framework while facilitates the identification of potential issues related to reliability and power usage.

The above work references [8], which introduces the SmartWatts energy measuring tool. Recall that this was already discussed in Section 1. As discussed, SmartWatts main limitation is in its use of RAPL, which is not available in virtualized environments. For this reason we consider Kepler instead, which has more options to work with software models that are trained on actual measurements [16].

Andringa [21] has also taken a look at the power-measuring problem described above. In this study, existing solutions for energy-measuring at multiple levels are evaluated and a new model for power mapping between bare-metal, virtual machines and Kubernetes pods is introduced. This model has promising results; however, it is not yet made available as a functioning tool. Moreover, as this approach focuses on multiple levels of the Kubernetes stack, an in-depth study for Kubernetes specifically is not performed.

As seen from the related work on energy measuring, existing models exist which are not readily usable as a tool, and existing tools exist which are not validated exhaustively to work accurately and consistently. Hence, this is what our research project will undertake.

[3]Fog computing is a technique in which a substantial part of computing for some task is carried out on edge devices [28].

2.3. Overprovisoning and waste

Besides the research which aims to improve or provide insight into how systems use energy and how to optimize this, research into overprovisioning is also related. It looks specifically into how cloud infrastructure is provisioned and if the provisioned resources are actually used. Unused resources may lead to significant waste. While most research to waste in cloud and Kubernetes is not directly related to our goal, it is important to consider it to underline the importance of this research project.

In 2022, Everman *et al.* looked at cloud waste and cost using an analysis of 2019 Microsoft Azure traces [5]. They look specifically at the allocation of provisioned VMs and find that the average CPU utilization of VMs is around 10 %. They also find that approximately 75 % of VMs have an average utilization of <10 %. This is obviously a significant waste of resources and can be costly for the users of these VMs. As potential reasons for this, the authors mention that users may simply be prone to overprovisioning, users may be unaware of improvements in hardware, and users may be provisioning a multicore system for non-parallel workloads.

This analysis is expanded upon by Huang *et al.* [29]. This paper specifically targets overprovisioning and excessive energy consumption by proposing two VM migration algorithms and two VM optimization algorithms: Core Reduction (CR) and Shutdown (SD). To estimate the power consumption of the VMs in question, they use the SPEC CPU Power dataset [30] and information from Azure traces. Their CR algorithm finds potential wasteful VMs and simply reduces its core count to reduce the overprovisioning while minimally affecting performance. The SD algorithm identifies VMs with extremely low utilization and shuts them down. It is important to note here that, while not mentioned in the paper, these algorithms are likely in conflict with most cloud provider's SLA, which will usually guarantee availability of resources as requested with some minimum uptime percentage [31]–[33]. That makes these algorithms, while effective, particularly aggres-

sive. Of course, such algorithms could be offered as an autoscaling/cost-saving measure that is opt-in, thereby not conflicting with guarantees.

The drawback of this approach is similar to that which we discussed in Section 2.1, where a cluster-wide or datacenter-wide optimization strategy still must abide by provisioning rules set by the user, meaning that overprovisioning is still very much possible. While we can create a rudimentary metric of provisioning factor for Kubernetes by simply dividing the CPU utilization by CPU request, this does not give a complete picture. As we know from [4], 100 % utilization should not be the goal as it is not most energy efficient where throughput is concerned. Moreover, energy usage in Kubernetes is usually guessed at instead of measured real-time, which makes it harder to draw conclusions. Therefore, we must explore energy usage in Kubernetes first.

2.4. Metrics and Observability

Kepler, which we discussed in the previous section and which we will be exploring more, exposes its metrics in Prometheus format. Their documentation recommends using a Grafana dashboard to gain insights into these metrics [34]. It is important for us to understand the types and the impact of metrics in the context of observability.

A survey on metrics and measurement tools for sustainability in cloud was performed in 2018 [35]. This paper highlights the importance of metrics at various levels and discusses also which types of metrics are important. This survey explored the types of metrics and the measurement tools that existed at the time related to sustainability in cloud computing. From fine- to course-grained they consider multiple metric scopes: component-level, equipment-level, network resource-level, facility-level and corporate-level. For sake of brevity we won't discuss all discussed metrics here. According to the survey, there is no current consensus on which metrics should be used. They mention that there is significant overlap in metrics as well, which makes comparing metrics very difficult[4].

From their extensive survey the authors make a few observations. For instance, the vast majority of metrics discussed consist of a performance part, a power/energy part and some emissions part. The only exception to this is metrics which consider the number of users of a system. The authors also emphasize the importance of updating metrics to consider new technologies, as, for example, software-defined networking has influence on how per-port metrics are calculated. We have seen something similar earlier where many power-measuring tools use RAPL which isn't available on (somewhat newer) virtualized machines.

A similar survey was performed in 2017. This survey specifically looked at energy-efficiency metrics at the datacenter facility, rack or equipment level [36]. From both this and the previous survey we see a distinct lack of deployment-level metrics where energy usage and efficiency is concerned.

A systematic literature study was performed in 2023 which does a state-of-the-art analysis on cloud observability [37]. It specifically looks at, among other questions, the motivation for equipping cloud(-native) applications with observability capabilities. They mention in their conclusion that they found the main motivations to be provisioning focussing on run-time tracing, measuring system overhead, and scheduling. From a managerial perspective they mention motivations as root cause detection, SLA management and end-to-end management. Lastly, they mention observability is highly motivated by the distributed nature of cloud applications, its integration with many other technologies, its heterogeneity, as well as testing concerns. The authors mention Prometheus to be the most popular monitoring solution.

While observability methods and their usage in practice are not directly related to our work, it is important to consider them in context. As observability tools become the de facto standard for cloud monitoring, it is important that a Kubernetes energy measuring tool also abides by this standard and produces its metrics in Prometheus format.

3. Experimental Setup

In the rest of this research project, several experiments are run. For these experiments, of which the details are discussed later, a Kubernetes cluster and observability platform is required. Additionally, we wish to measure ground-truth power of our Kubernetes cluster as well as several other metrics.

[4] https://xkcd.com/927/

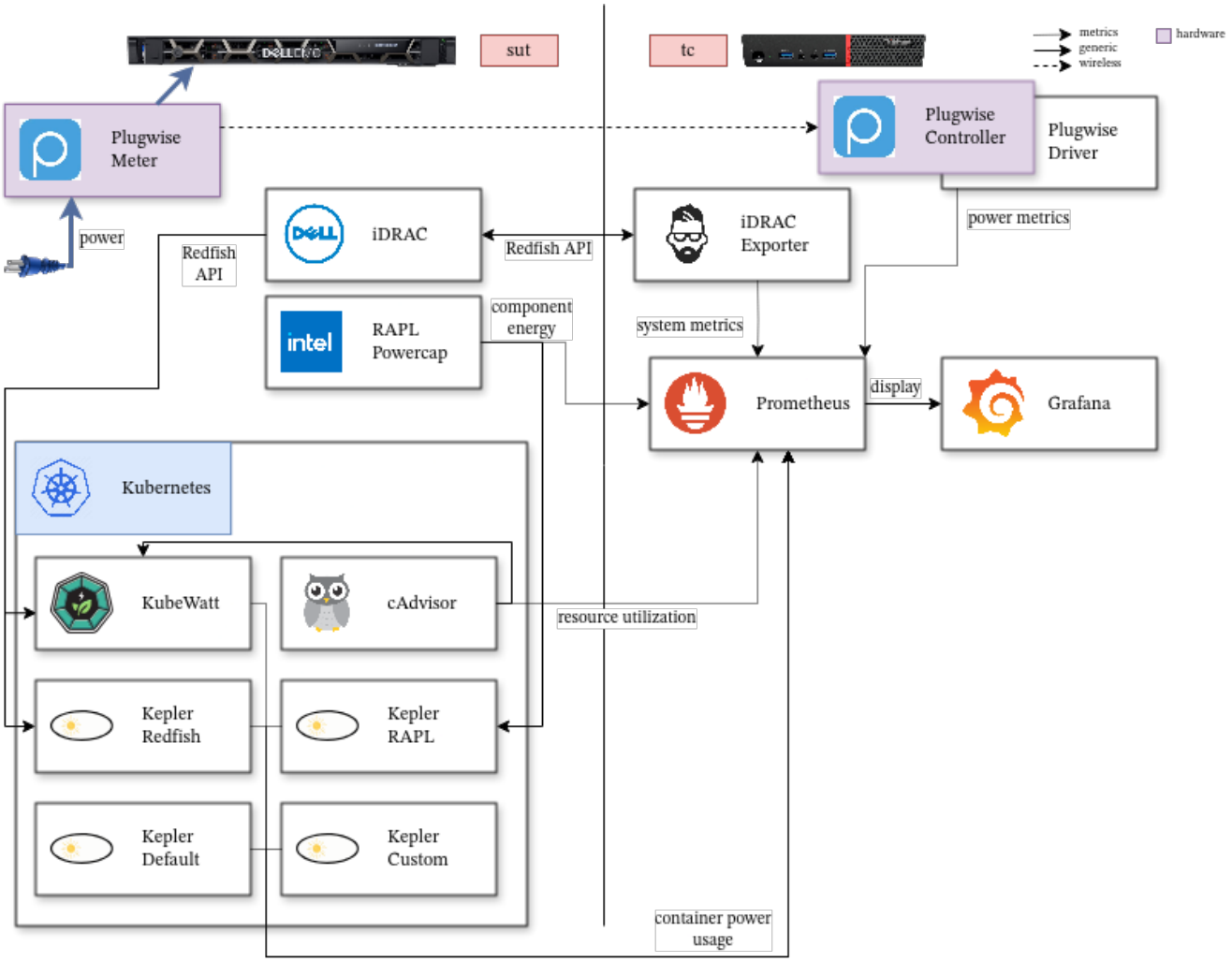


Figure 3.1: Complete testing setup, including Kepler and KubeWatt

In this section we discuss the experimental setup used for the aforementioned experiments. In Section 3.1 we outline the hardware that is used; in Section 3.2 we discuss the software and metrics gathering aspects of the setup. Both Kepler and KubeWatt are deployed in this experimental setup. Their deployments are discussed in the respective sections: Sections 5 and 7. A complete diagram of the testing setup, including Kepler and KubeWatt, is shown in Figure 3.1.

In the GitHub repository for this research project[5], configuration files are available for all components of the experimental setup. Additionally, test scripts are available for the tests that are performed and datasets in CSV format are available for all figures.

3.1. Hardware

The hardware that will be used for the experiments consists of a few different components: our main system under test is a Dell PowerEdge R640 server. It is equipped with two Intel Xeon Gold 6226R processors totaling 32 cores and 64 threads, 96 GB of RAM and ~256 GB of RAID1 SSD storage. Fedora Server 40 is installed. The server is also equipped with Dell iDRAC9 from which we can obtain, among others, power metrics using the Redfish API integration. We will refer to this server as `SUT` (for system under test) in the rest of this project.

A secondary machine is deployed alongside `SUT`. This is a Lenovo m910q with an Intel Core i3-6100T, 8 GB of RAM, and 256 GB of NVMe storage running Fedora Server 40. This machine is meant for supporting workloads required for measuring and analyzing the tests we perform on `SUT`. These supporting workloads are on a

[5]https://github.com/bjornpijnacker/msc-thesis

separate machine so that they don't influence test results of SUT. We will refer to this machine as tc (short for ThinkCentre) in the rest of this project.

The power supplies of SUT are externally power-monitored at the wall outlet in addition to the internal power monitor that iDRAC provides. To this end we use Plugwise Energy Monitoring plugs of which the controller is connected to tc.

3.2. Software

On SUT, we deploy a single-node Kubernetes cluster. This cluster is bootstrapped using Rancher Kubernetes Engine (RKE) [38] version 1.5.8. RKE allows us to administer the cluster without running specific software on-node, as it uses SSH to set up the cluster on the node. The choice to use RKE was an arbitrary choice and was chosen due to prior experience with the technology. For the purposes of this research, the exact Kubernetes distribution should not matter as long as it can be installed on bare-metal Linux. The cluster is set up in its most basic form, and we do not run any workloads on here that are not necessary for our tests, such as nginx-ingress-controller, in order to reduce the amount of noise that our power and CPU metrics will see.

To gather and store metrics from various aspects of the system under test, an observability stack is required. In this case, Grafana and Prometheus are used both due to prior experience with the technologies and because Prometheus-format metrics are supported by all components of which metrics are collected. Additionally, Prometheus has had CNCF graduated maturity since August 2018, making it the recommended metrics platform for our use-case [39]. Grafana and Prometheus Server are deployed and configured as services on tc, so their processing does not influence the power used by SUT. A list of all collected metrics is available in Appendix A.

To extract metrics from SUT at a hardware level, we use an iDRAC exporter for Prometheus [40]. This exporter also runs on tc and uses the iDRAC Redfish API to extract available metrics. When Prometheus scrapes the exporter endpoint, it uses the Redfish API to gather the information and transform it to Prometheus format. Of particular interest will be the power supply metrics which will allow us to measure power going into the system. Additionally, metrics from the Plugwise plugs are imported using a Prometheus exporter driver [41]. It exposes the current wattage used by one or more Plugwise plugs in Prometheus format. This software must run on the same system as the Plugwise controller is connected to, this being tc.

Since Kepler uses RAPL as a source of power, we also import RAPL data into Prometheus. For this purpose we write custom software[6] that can read the RAPL Powercap information and expose this to a Prometheus scraper. While RAPL is an Intel processor technology, the Linux power capping framework exposes the underlying RAPL information as files in the Linux filesystem [42]. For SUT, RAPL exposes—for each CPU—platform energy usage and DRAM energy usage as a counter in μJ.

Lastly, we import Kubernetes cAdvisor metrics for our Kubernetes node. This allows us to see per-container metrics such as CPU and memory usage.

4. Verifying Power Measuring

Before running experiments which verify power usage, we must consider how power is measured, as an important aspect of the upcoming experiments is to accurately measure power usage as a ground truth. As explained in Section 3, we have deployed a Plugwise power measuring plug to measure the total server consumption, and we have iDRAC reporting power consumption as reported by the power supplies. To verify that power measuring is accurate, these two sources of power metrics are compared to see whether their reported values match up. If both sources produce the same measurements then we can safely assume that they are accurate. Moreover, as Kepler also uses RAPL as a power source, we compare RAPL to a ground-truth power measurement to find whether it is accurate, and investigate whether there is some quantifiable relation between the ground-truth power measurements and RAPL power measurements.

4.1. Hardware measurements

On SUT, we run a script which uses `stress-ng` to stress the system to induce power consumption we can measure. We run several CPU stressors. Since the exact stressors do not matter, we choose a random sequence

[6] The source code is available at https://github.com/bjornpijnacker/msc-thesis/tree/main/setup/sut/rapl-prometheus

of stressors: $[13, 5, 4, 13, 4, 2, 20, 10]$. We run each for a minute and sleep 30 s in between. We see the results of this in Figure 4.1. This test is repeated three times, with errors indicated by the shaded regions in the figure.

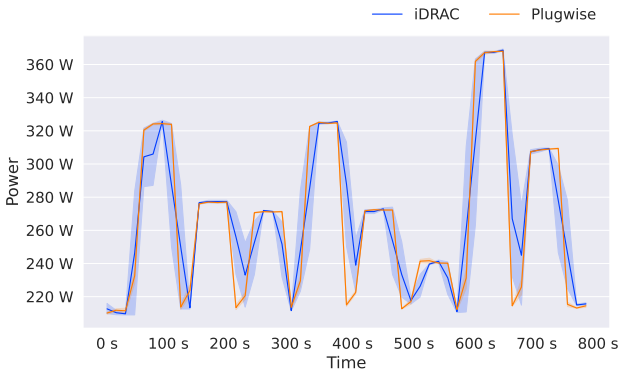


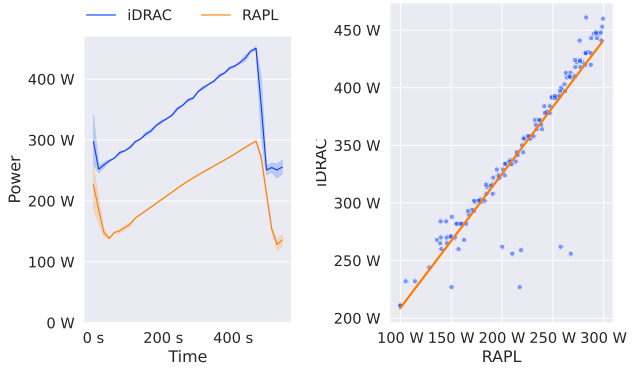
Figure 4.1: Power metrics of iDRAC and Plugwise compared

As we can see from this figure, the two metrics are very similar. The main difference that we observe is time-based, where the iDRAC measurements have some more variance in time than the Plugwise measurements. Value-wise the measurements are very close. Since both measurement tools seem perfectly adequate for our case, we will be using the Dell iDRAC measurement for future tests. We prefer iDRAC over Plugwise due to the inclusion of the Redfish API in iDRAC, which allows both Kepler and KubeWatt to use the power supply metrics.

4.2. RAPL and iDRAC

To measure the relation between RAPL and iDRAC, we run a CPU stressor which slowly ramps up over the range of CPU utilization to get a complete range of measurements. The measurements over time should give us an indication of RAPL's accuracy as it pertains to CPU power usage. We can then scatter the iDRAC against RAPL values to find whether there is a relationship which we can easily quantify. The test is repeated four times.

Figure 4.2 shows the result of this test, where Figure 4.2a shows the runtime of the four tests and Figure 4.2b shows a scatter of the two metrics for all four tests combined. From the first plot it is obvious that RAPL does not produce an accurate reading of total system power compared to what the power supply is actually reporting, with RAPL underreporting power utilization



(a) Power usage timeseries during test

(b) The two timeseries scattered over the duration of the test

Figure 4.2: Total power as measured by RAPL Powercap (sum of all components) and iDRAC during the ramp-up test

by approximately 100–150 W. However, as displayed in the second plot, the relationship seems very linear and well-behaved. A linear regression on the data in Figure 4.2b finds a model with

$$\text{iDRAC} = \alpha \cdot \text{RAPL} + \beta,$$

with $\alpha = 1.16$ and $\beta = 93.0$, with an R^2-value of 0.85, indicating that this fits the data quite well. Note that this does not take into account the multiple components which RAPL measures, where one component could require a different transformation to another; the test we have run only stresses the CPU cores, not the other components. This result simply indicates that RAPL could be a good candidate for power measurement if tuned to the system in question. Since we have iDRAC available, we will use it as the single source of truth where possible, and we will not explore the idea of RAPL tuning further.

5. Validating Kepler

Previously, we have seen Kepler as the state-of-the-art energy-measuring tool for bare-metal and virtual-machine based Kubernetes. Additionally, in Section 1 we have explained the need to experimentally validate Kepler, and have asked a series of research questions to guide this validation. Recall from this section:

KPRQ1 How accurate are Kepler's total node measure-

ments compared to a ground truth?

KPRQ2 How (well) does Kepler attribute energy usage to containers on the node?

KPRQ3 How do the different Kepler configurations affect the accuracy of the reported metrics?

To be able to accurately attribute energy usage to specific containers, Kepler must be able to accurately measure (or estimate) the total amount of energy that a node is using. It is easy to compare this to a ground truth. Since Kepler aims to supply granular information on energy usage in regard to pods and containers as well, we will also validate how Kepler attributes energy to specific containers. We cannot measure a ground truth in this aspect; however, we can investigate Kepler's behavior in certain scenarios and assess whether results are as expected based on resource utilization that we manipulate. Lastly, we aim to investigate the different configuration options Kepler has available and discover what effect these have on the metrics which Kepler exports.

In this section, we propose a number of experiments that validate Kepler's measurements as consistent and accurate. The rest of this section proceeds as follows: in Section 5.1 we discuss the experiment design and Kepler deployment, in Section 5.2 we showcase the result of the experiments, and in Section 5.3 a discussion of the findings is provided.

5.1. Experimental Setup and Design

5.1.1. Deploying Kepler

To answer the posed questions, the testing setup as discussed in Section 3 is used. On the Kubernetes cluster, four instances of Kepler are deployed, each with a different configuration. We deploy Kepler version 0.7.2 as newer versions suffer from an issue where incorrect values are sometimes measured at random [43]. The Kepler configurations that are used are:

1. KEPLER-DEFAULT: This instance uses the default estimator model. Note that by default Kepler does not support disabling RAPL in favour of using the estimator, so for this deployment a custom Helm chart is used which overrides `/sys/class/powercap/intel-rapl/`, the location at which the Linux Power Capping framework exposes RAPL data [42], with an empty directory thereby making Kepler assume RAPL is unavailable. This can be confirmed by checking the logging, which indicates "Unable to obtain power, use estimate method".

2. KEPLER-CUSTOM: This instance is equal to KEPLER-DEFAULT except that it runs a custom power model. More detail on training and using this custom model is given below. Note that, again, a custom Helm chart was used since the provided Helm chart does not have configuration options or manifests for running the required sidecar deployment.

3. KEPLER-RAPL: This instance runs Kepler with default settings and uses RAPL as a source of platform and component power. There are no external sources of power provided.

4. KEPLER-REDFISH: This instance runs Kepler using the Redfish iDRAC integration for platform power. Since Redfish cannot provide component power, RAPL is used for component power.

Custom Power Model

Kepler supports training a custom power model. This model can be used when no other source of power is available and is trained on the specific features of the system it will be used on. According to Kepler documentation, it is necessary to train a custom power model to overcome limitations of the pre-trained power model and to tailor it to be system-specific [16]. The limitation that is targeted here is that the default power model is not tailored to the system in question. A power model works only on the combination of hardware it was trained on, as it estimates power consumption from utilization counters.

To train a custom power model, we use the Kepler model server [44]. This extracts information from a running Kepler instance through Prometheus and uses these metrics to train a model which can estimate energy based on available counters. In the model server, the Kepler authors provide Tekton pipelines which contain the neces-

sary steps to train the model[7]. The training can be configured by choosing the correct pipeline and configuring options such as the power source and feature group to use. The single-train pipeline allows for training a model on a single source, such as Redfish or RAPL, and the complete-train pipeline attempts to use all available sources to train a more complete model.

In our case, the model is trained on the Redfish instance and Redfish data, as it provides a ground-truth measurement of power usage, which is the most truthful source available to us. We have additionally validated its accuracy in Section 4. Moreover, our attempts on training a model on the RAPL instance were unsuccessful. Specifically, a single-train pipeline run on rapl-sysfs source with the CounterIRQCombined feature group failed with `failed: [extract : extract] AttributeError: 'NoneType' object has no attribute 'to_csv'` and a complete-train run on combined sources redfish and rapl-sysfs failed with `failed: [train-from-query : pipeline-train] cannot get pipeline`; both errors that are not mentioned in Kepler documentation and for which we could not find an explanation.

To train the model, we run the default single-train pipeline that the Kepler Model Server provides. The pipelinerun is updated to use Redfish as an energy source and to use CounterIRQCombined as feature group, since this is the most complete group that Kepler model server reports to be available. After the training is complete, the model can run in the Kepler sidecar estimator. Our model is suitable to be used as a node platform power model, since it was trained on a source of platform power.

5.1.2. Experiment design and system load

To answer the research questions that were stated earlier, we run multiple tests. In this section we explain which tests are performed and what results are expected. Each test is repeated three times to ensure consistency.

For each test, stress-ng is used to perform load generation. This script allows for load generation with various parameters and is customizable in that it lets the user select which subsystems of the computer are stressed, and to what capacity [45].

Single-stressor CPU test

First, we run a simple arbitrary workload on the test cluster, and observe the total system power usage of each Kepler instance compared to the ground truth of power and raw RAPL measurements. We investigate power measurements and total energy usage over the test interval. While running the test we investigate not only the measurements that Kepler gives for our running test-container but also look at the power/energy usage of other containers in the cluster to gain insight into container power attribution.

To verify total system energy measurements, we run a single stress workload on our Kubernetes cluster. This is invoked using `stress-ng --cpu 32 --timeout 5m`. Afterwards, we let the system sleep for 5 minutes before running the test load again. We repeat this three times.

Before running this test, we also set up 16 idle containers that use no energy. They will simply run the `date` command then exit and not restart. These containers will have the 'Completed' status while our main workload is running. We turn off the pods, so they do not interfere with the 'dynamic'-mode power attribution by actually using power. Installing these containers should yield a situation closer to a real-world cluster, where a container is not the only container on the cluster. However, as we do not want these container to influence CPU utilization and power usage, they remain as 'Completed' containers. We expect these containers to not be attributed any power during the runtime of our tests since they are not running.

Since our test is running in a container, we can track the energy that Kepler attributes to each container in the cluster. Since we run just a single workload that will not use all cores of `SUT`, we expect the energy usage of all containers to stay mostly static during the test, except for the testing container, which should show a peak similar to the peak we expect to see in total power usage. We also expect all other containers in the cluster to be using very little power compared to the test container since the cluster should be mostly idle.

We obtain the total system energy from Kepler using the `kepler_container_joules_total` Prometheus metric and summing this for all containers, grouped by Kepler instance. Since Kepler attributes all power used

[7] https://github.com/sustainable-computing-io/kepler-model-server/blob/main/model_training/tekton/

in the system to containers, this will give a measure of the total power per node. How Kepler decides the total node power differs by configuration and depends on the power metric source(s) that are available to that configuration. Kepler will, for instance, use Redfish over RAPL for platform power when both are available, and RAPL component power over the estimation model when RAPL is available.

With this test we attempt to verify both Kepler's total system power measurements and gain insight into the container attribution.

Node component tests

Besides total system power and per-container power, Kepler also specifies power used per node-component. Kepler has five different node-based measurements that it provides as metrics: core, dram, package, platform and uncore, where platform is the total node energy usage and core, dram, package and uncore are all components of the system. Specifically, the core metric is the power usage of the CPU cores, dram is the power usage of the system RAM, uncore includes the power usage of all CPU components which are not the core itself, and package is the power of core and uncore combined. Of interest to us is running a stressor such that we expect just a single of these components to change, and observe what happens.

We will be running three tests: one test that stresses just CPU using `stress-ng --cpu 16`, one test that stresses just RAM using `stress-ng --vm 8` and one test that stresses both as a combination of the earlier two tests. We will let each run for five minutes with a two-minute pause between each. Note that the stress-ng RAM stressor will also somewhat stress the CPU as CPU utilization is required to manipulate RAM.

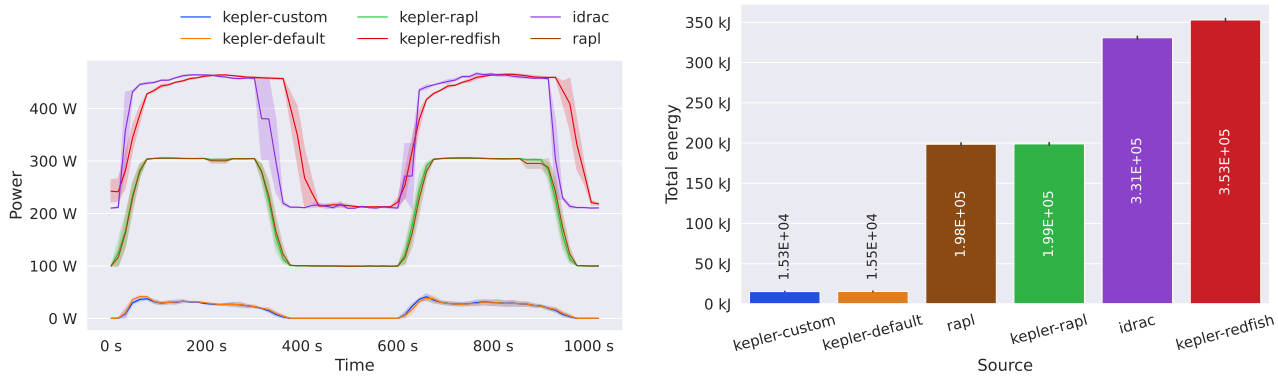
What we expect to see is a spike in 'core' for the first test, 'dram' in the second, and both in the third test. We also expect the 'dram' component to not report much power usage in the first test, and 'core' to report little power usage compared to the CPU-stressor tests when only running memory stressors.

5.2. Results

In this section we discuss the results of the several tests that we have outlined earlier. The results are organized per test and are used in Section 5.3 to answer the research questions we posed in Sections 1 and 5.1. All tests are run multiple times. Each figure in this section will indicate with a solid line the mean test results and will indicate with a shaded region the error as a 95 % confidence interval.

5.2.1. Single-stressor CPU tests

Figure 5.1 shows the power and total energy measured during the test for each of the different Kepler deployments as well as the raw RAPL powercap and iDRAC power measurements. As evident from these figures, the different Kepler deployments report significantly different findings. In Figure 5.2, the RSME between each Kepler instance, iDRAC and RAPL is shown, to further showcase this fact. Between KEPLER-REDFISH and KEPLER-RAPL there is an RSME of 165.3 W, between KEPLER-REDFISH and KEPLER-DEFAULT there is



(a) Power per Kepler instance, iDRAC and RAPL

(b) Total energy used per Kepler instance, iDRAC and RAPL

Figure 5.1: Power and energy as measured during the single-stressor test

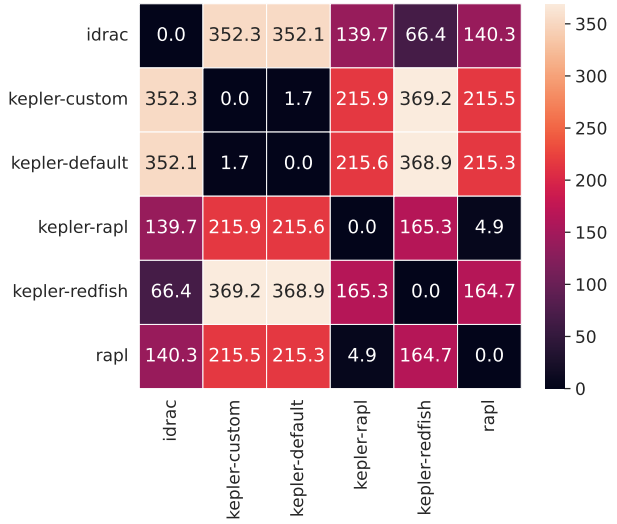


Figure 5.2: Internal RSME of power measurements for the single-stressor test

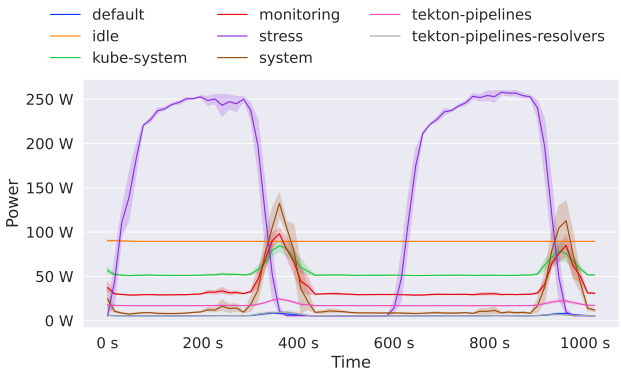


Figure 5.3: Attributed power usage per namespace by KEPLER-REDFISH during the single-stressor test

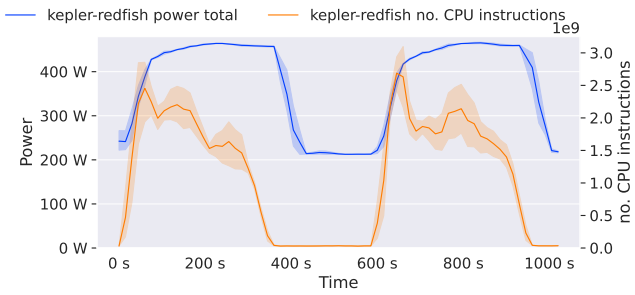


Figure 5.4: Timing of CPU load vs. power consumption measurements of the single-stressor test

an RSME of 368.9 W, between KEPLER-REDFISH and KEPLER-CUSTOM there is an RSME of 369.2 W. Then between KEPLER-RAPL and KEPLER-DEFAULT and KEPLER-CUSTOM there is RSMEs of 215.6 W and 215.9 W respectively and between KEPLER-DEFAULT and KEPLER-CUSTOM the RSME is 1.7 W. This indicates that each Kepler deployment has considerably different measurements to one-another, except for KEPLER-DEFAULT and KEPLER-CUSTOM. It is notable that the values for KEPLER-DEFAULT and KEPLER-CUSTOM correspond quite closely, since we had expected the custom trained model to perform closer to the KEPLER-REDFISH instance, as this was trained on KEPLER-REDFISH.

We also see that each Kepler strategy reports values which are very close to its external source of energy: the power graphs (Figure 5.1a) and energy totals (Figure 5.1b) line up very well for KEPLER-REDFISH and the iDRAC power, and for KEPLER-RAPL and raw RAPL powercap values. The RSME for KEPLER-REDFISH and its power source iDRAC is 66.4 W and between KEPLER-RAPL and RAPL itself is 4.9 W. This alone would indicate that KEPLER-RAPL is more true to its source of power than KEPLER-REDFISH though as evident from Figure 5.1a: the iDRAC measurement is somewhat misaligned from KEPLER-REDFISH which will influence the RSME.

Figure 5.3 shows the attributed container power of the KEPLER-REDFISH instance summed by Kubernetes namespace. The results are partially what was expected: the power attributed to the 'stress' namespace, which houses exclusively the stress-test container running our workload, corresponds to the generated load which was expected. Two things in this figure are not as expected: (1) there is a peak of power usage in other namespaces after the stress-test pauses; (2) the 'idle' namespace consistently uses ~160 W, even though it should not be using energy. This peak is easily explained by considering that iDRAC may measure slower than Kepler measures CPU usage. We verify this in Figure 5.4, where we indeed see that power consumption metrics lack behind CPU load metrics by up to 1 min when CPU load quickly decreases. This finding is in-line with the specification, as the Redfish API itself specifies power metrics are updated on a one-minute interval[8]. We saw a similar situation in Section 4. In such a case, Kepler needs to attribute more power usage among containers than is

[8]Available through redfish/v1/Chassis/System.Embedded.1/Power/PowerControl API endpoint.

actually occurring at that given moment, therefore artificially spiking the power usage of all workloads as CPU usage suddenly changes.

We see unexpected power usage of the 'idle' namespace. To investigate this, we determine which containers are using power according to Kepler. Looking into the power attribution of all containers in namespace 'idle', we see that there are containers that are reported to be using ~6 W each according to KEPLER-REDFISH. Recall that we created these containers as part of the test, and that the containers are not running. All power attribution to these containers is of the 'idle' mode, where the power attributed to our stress container was 'dynamic'-mode power. In KEPLER-RAPL, each container is attributed ~2.5 W and in both KEPLER-DEFAULT and KEPLER-CUSTOM, the containers do not have reported power usage at all.

Consider first KEPLER-REDFISH since this is the same Kepler instance also depicted in Figure 5.3. While Kepler reports the containers to be using power, according to `kubectl`, the Kubernetes management CLI, these pods are all in a completed status. There are no pods that are actually running in the namespace 'idle'. This means that Kepler is reporting power usage for containers which are not running which indicates that it is not able to properly attribute 'idle'-mode power. Since the total power usage of all containers was correct in regard to the iDRAC measurement, this also indicates that Kepler is in turn underreporting the idle power used by other containers in the RAPL and Redfish implementation. The KEPLER-DEFAULT and KEPLER-CUSTOM instances do not report this erroneous power usage. In fact, these estimator-based Kepler deployments show attribution data much closer to what we would expect given the cluster load, in terms of power ratio. The only factor which differs between deployments is the method used to obtain total node and component power, since the ratio power model is the same for each of them.

To find out why we observe this difference, let's consider each Kepler deployment and each power component. Even though we have simplified our deployments as KEPLER-REDFISH, KEPLER-RAPL, etc, this pertains only to the total system energy usage; what Kepler calls 'node-platform'. Kepler also measures node components, such as DRAM, package and uncore, which may use different sources depending on availability. By reading labels from the Kepler metrics we can find these sources for each node-metric. This is shown in Table 1.

kepler-	**rapl**	**redfish**	**custom**	**default**
core	rapl	rapl	rapl	rapl
dram	rapl	rapl	rapl	rapl
package	rapl	rapl	rapl	rapl
platform	none	redfish	none	none
uncore	rapl	rapl	rapl	rapl

Table 1: The source of Kepler node-metrics for different deployment configurations

We see that, except for the `kepler_node_platform_joules_total` metric, RAPL is used as source everywhere. This makes sense, because Redfish can only provide total system power at the power supply level; it does not know how much of this power is CPU, DRAM, or other components, hence RAPL is used to obtain this information where available. For the systems that do not use Redfish, there is no source of total platform power; therefore, `kepler_node_platform_joules_total` is obtained by using the available models. For the KEPLER-DEFAULT and KEPLER-CUSTOM, RAPL is also named as a source; however, this is simply because the model was trained using RAPL as a source with regards to component power.

Since KEPLER-DEFAULT and KEPLER-CUSTOM use the same models for component power, and KEPLER-RAPL and KEPLER-REDFISH use the available RAPL data for component power, we can attribute that the observed difference to (at least) one of the component metrics.

To see whether Kepler can correctly attribute power once the inactive pods have been deleted, consider the following. First, we start 64 idle pods that run `date` and complete. We choose a large number so that their presence and absence has a large and thus easy to observe effect on measurements. After these complete we give the system one minute to stabilize. We then run a small (8 CPU) stressor and after two minutes we delete all inactive pods simultaneously, then observe how the container attribution of Kepler changes. We expect that Kepler reallocates the idle-mode power usage to all other running containers, and that the dynamic-mode power attribution does not change.

The results of this test are presented in Figure 5.5. In Figure 5.5a the total power reported by the Kepler deployments as well as iDRAC and RAPL directly are depicted.

As evident and as expected, deleting the inactive pods does not have an effect on total system power, since these pods were idle. Figure 5.5b shows the power attributed by Kepler to each namespace. Figures 5.5c and 5.5d show this for 'dynamic'-mode and 'idle'-mode power respectively. Recall that the 'stress' namespace houses solely our stressor pod and that the 'idle' namespace only has the 64 idle pods. As the test starts, the total power goes up for the 'stress' namespace as expected. As the idle pods are deleted, the 'idle'-mode power for the 'idle' namespaces quickly goes to zero as expected. Power is re-attributed throughout the other namespaces over all remaining containers.

After deleting the idle pods we also see the dynamic-mode power usage for 'stress' go down and the dynamic-mode power usage for 'system' go up. Note that the CPU usage of the workload remained at 100% and consistent throughput was indicated throughout the test as per the stressor logging, as also indicated in Figure 5.5f. The change in power attribution here is unexpected, since the 'system' namespace is not running any workloads. The upward trend of this namespace goes matched with a downward trend in reported dynamic power usage for the 'stress' namespace. This power is being attributed to pods named 'system_processes', which is a reserved name in Kepler for processes that cannot be attributed to a pod. After stopping the testing workload running in the 'stress' namespace, we see that the 'system' namespace also reports using less power. Note that while Figures 5.5b to 5.5d show namespace power usage for the KEPLER-REDFISH instance, the same behavior shows for all other deployments of Kepler as well. We conclude that Kepler was not able to properly attribute the power used by the stressing container to that container.

Figure 5.5e gives insight into why Kepler is reporting this power usage for 'system_processes'. In this figure, we see a similar situation as Kepler reported: cAdvisor reports CPU utilization for containers without a namespace as well as for containers *with* a namespace. Looking at the raw data, all the values without a namespace are one of: (1) actual system processes; (2) slices, where these slices are grouped metrics of multiple processes. For instance, the `id=/kubepods.slice/kubepods-besteffort.slice` labelled metric gives aggregated CPU utilization for all best-effort pods. When considering only the actual containers, such grouped slices should be ignored. Filtering the cAdvisor output to only include actual containers gives the result as shown in Figure 5.5f, which is exactly as expected given the stressors that were run.

From the several tests performed above, we can conclude that Kepler is unable to properly attribute power usage based on resource utilization.

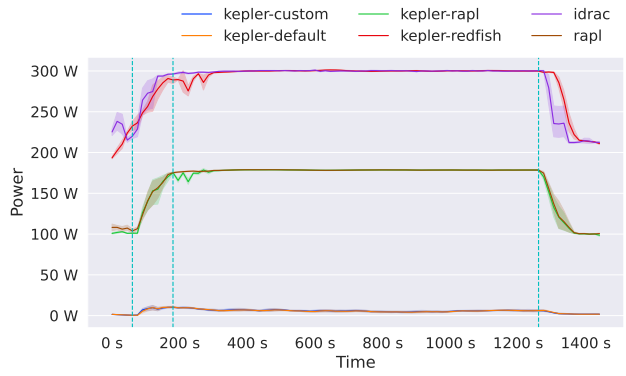
5.2.2. Node component tests

The results of the node-component tests are shown in Figure 5.6. We consider the reported package consumption and DRAM consumption, since our RAPL deployment does not support specifying core/uncore package components separately. The blue shaded regions specify the runtime of the CPU-only test, memory-only test and combined test, respectively.

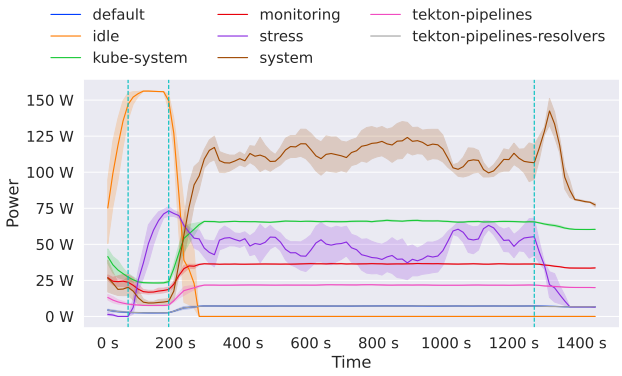
In Figure 5.6a we see the package power that Kepler reports. This includes the CPU cores as well as uncore[9] components such as the memory controller. For the RAPL-powered Kepler deployments, we can see that there is lower reported package power usage during the non-CPU test. Some power usage is reported, however, which can be attributed to the memory controller and the CPU overhead of reading/writing to the memory often. For KEPLER-DEFAULT and KEPLER-CUSTOM, the Kepler instances which use estimation instead of RAPL, we see results which are not as expected. For each of the tests, the power usage is reported higher than the previous one. However, we would have expected the second test to use less power than the first one at the CPU-level, which is also what RAPL reports.

In Figure 5.6b we see the reported DRAM power. As expected, RAPL does not attribute any power to DRAM when the CPU-only test is running. For the second and third test we see that the memory is indeed using some power, and that this is equal for both memory tests. For the KEPLER-DEFAULT and KEPLER-CUSTOM instances, we see a similar result to the package power, where consumption increases for each consecutive test.

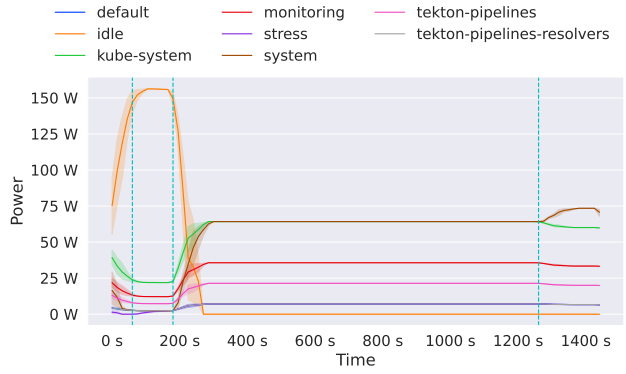
[9]Uncore refers to the functions of the CPU which are not in the core but are closely connected to the core for performance reasons.



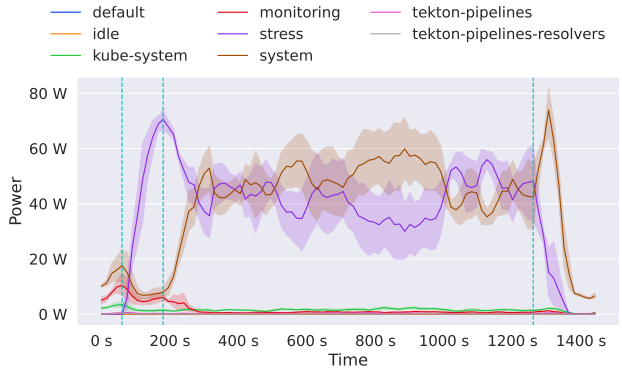
(a) The total system power usage



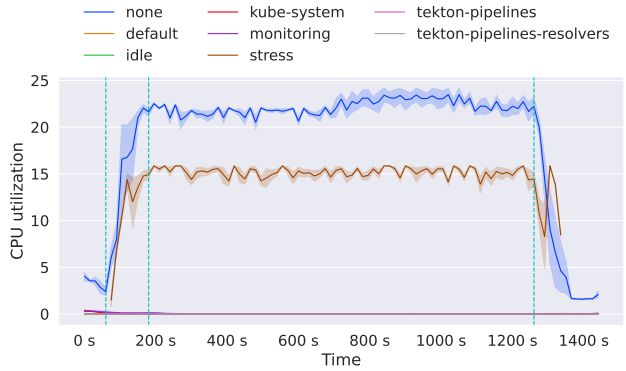
(b) The total power per namespace



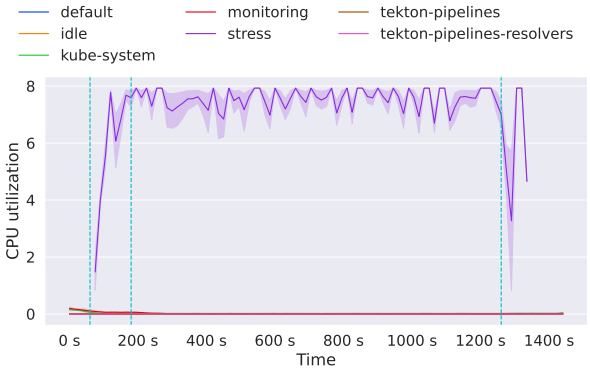
(c) The idle-mode power per namespace



(d) The dynamic-mode power per namespace



(e) The CPU-utilization per namespace; all cAdvisor values



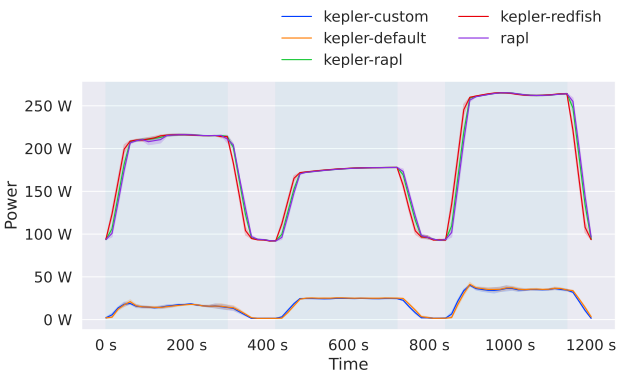
(f) The CPU-utilization per namespace, containers only

Figure 5.5: The result of deleting inactive pods. The blue markers indicate the times at which (1) the stressing load was started; (2) the inactive pods were deleted; (3) the stressing load was stopped, respectively. The tests were repeated four times.

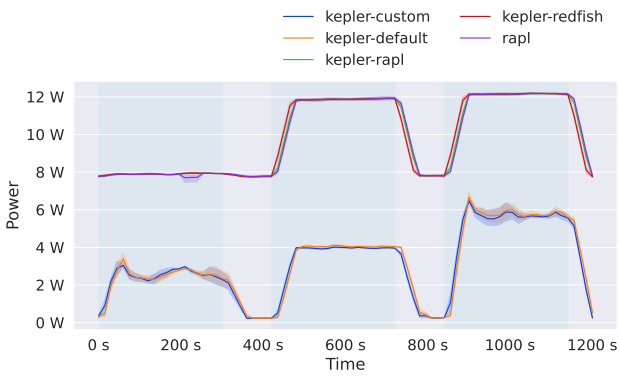
For KEPLER-RAPL and KEPLER-REDFISH we see that it corresponds exactly to RAPL.

In Figures 5.6c and 5.6d we see the matrix of root mean squared error such that we can properly compare the different metric sources. We see that for both package and DRAM components, KEPLER-RAPL, KEPLER-REDFISH and RAPL itself have a very small error to one another. The same holds for KEPLER-CUSTOM and KEPLER-DEFAULT. Between these two groups the error is, however, quite significant with an ~172 W error for package component and ~7 W error for the DRAM component. These errors are approximately 64 % and 55.5 % of the maximum measured value.

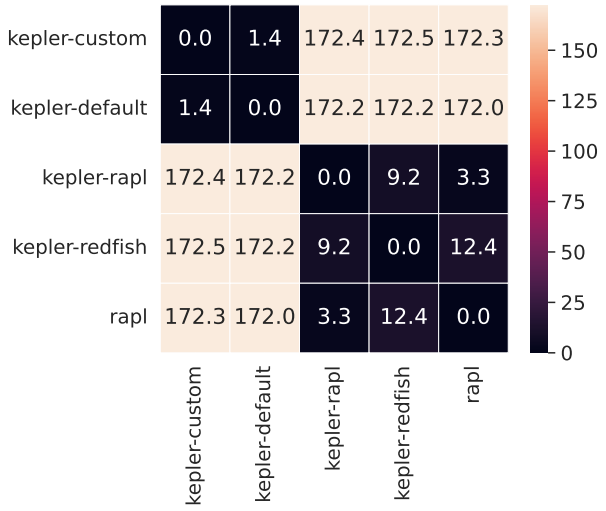
Conclusively, we have seen that Kepler deployments which can read RAPL values to obtain component power, KEPLER-REDFISH and KEPLER-RAPL in our case,



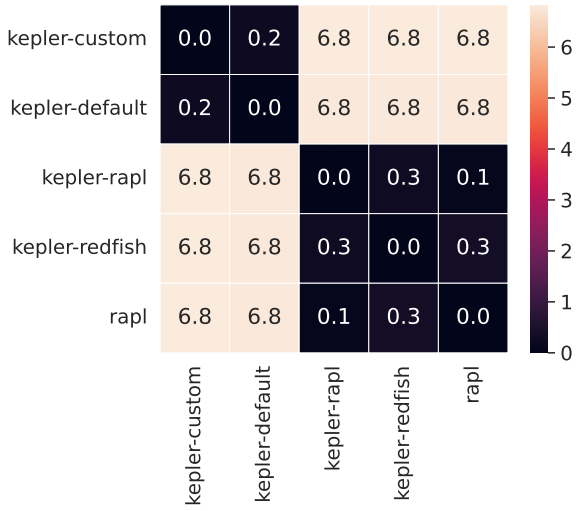
(a) Kepler readings for the node package component



(b) Kepler readings for the node DRAM component



(c) RSME for the node package component



(d) RSME for the node DRAM component

Figure 5.6: Kepler node component measurements for Package and DRAM. Shaded regions indicate respectively: (1) a 16-stressor CPU test; (2) an 8-stressor memory test; (3) the previous two combined. The bottom figures indicate root mean squared error heatmaps for the top figures.

do so accurately, as the measured values do not significantly deviate from what RAPL reports. Recall that the RAPL values are *not* accurate to the ground truth, as seen in Section 4. For the KEPLER-DEFAULT and KEPLER-CUSTOM instances, which do not have direct access to RAPL, the reported power is not as close and does not follow the same trend as RAPL does. For KEPLER-CUSTOM, however, this is as expected, since we were unsuccessful in training a custom node component model based on RAPL.

5.3. Discussion

With the tests we have performed above, we can now answer the research questions posed in Section 1.

KPRQ1: How accurate are Kepler's total node measurements compared to a ground truth?

In regard to external power sources we consider only the KEPLER-REDFISH and KEPLER-RAPL instances, as these subscribe to an external source of power usage. In general we have seen that Kepler is able to accurately use an external power source in its metrics. KEPLER-REDFISH reported a node power value RSME of 66.36 W to iDRAC itself, most of which is caused by measurement latencies; KEPLER-RAPL reported an RSME of 4.91 W to RAPL itself. We have seen a good example of this in the KEPLER-REDFISH deployment, where total energy usage over time corresponds closely with that of iDRAC itself. Similar results hold for KEPLER-RAPL which corresponds closely with RAPL values even though these are not ac-

curate to the actual energy that is used when not properly calibrated.

The KEPLER-DEFAULT and KEPLER-CUSTOM instances, which used estimation instead of a power source, were not able to properly estimate the total power that a node uses. In the single-stressor test, these instances were off by over 99% when compared to the iDRAC value, which represents the true power that the system used. The default estimator model is obviously not suitable for the specifications of SUT; however, training a custom model for this goal did not appear to be a workable alternative.

It should be noted that training ones own Kepler model is not an easy task. There is technical and architectural documentation available on the model server, but there is no concise documentation available that helps the user in choosing the correct configuration options, debugging the model training when it fails, validating the trained model or using the trained model with the Kepler Helm chart. The available documentation is also inaccurate in some places; for instance, the documentation specified only RAPL or ACPI are available to train the model; however, the model server training parameters also allow using Redfish and other such sources [46], [47], though using Redfish to train the model did not work in our case.

KPRQ2: How (well) does Kepler attribute energy usage to containers on the node?

The container power attribution leaves much to be desired. In the tests we have seen multiple examples of container power attribution that were clearly inaccurate and sometimes inexplicable, both in idle- and dynamic-mode power. In regard to container power attribution, Kepler does not produce proper measurements. As we have seen in the inactive-container deletion test, Kepler misattributes some power to system processes, where it should not be doing so. Additionally, as we have seen in the single-stressor test, Kepler attributes power to non-running containers, which is also not correct.

KPRQ3: How do the different Kepler configurations affect the accuracy of the reported metrics?

The total node power differed significantly from configuration to configuration. As already discussed, with iDRAC having the true source of power, the KEPLER-REDFISH instance was able to accurately access and portray this value. The KEPLER-RAPL instance had a RSME of 18.7 % compared to iDRAC; however, we saw that the relation between RAPL and therefore KEPLER-RAPL and iDRAC and therefore KEPLER-REDFISH is a linear one. The accuracy of KEPLER-DEFAULT and KEPLER-CUSTOM was much worse where they reported an RSME with an error of over 99% when compared to the iDRAC value.

When attributing energy to containers we saw identical behavior in each of the four Kepler deployments. This makes sense, as they are all running the same power attribution model against the available node metrics, and are all able to access the same source of resource utilization counters. The differences that were observed are explained by considering that idle- and dynamic-mode power are attributed differently and that the ratio of idle-to-dynamic-mode power is obtained from node metrics whose values differ by configuration.

6. KubeWatt Architecture and Implementation

In Section 5 we have shown that Kepler is not a suitable tool for producing container-level power metrics in Kubernetes. As an alternative, we will create our own tool: KubeWatt. It will be based partially of the power attribution model which is proposed in [21], which yielded promising results in container/pod power attribution.

In this section we discuss the implementation of a proof-of-concept version of KubeWatt. It is implemented based on some assumptions, which are discussed later, to make it suitable for the testing environment we use for its evaluation. This means the tool might not be suitable for all environments, though we aim to discuss these details where necessary. Assumptions that we make are numbered for this purpose and are discussed in Section 6.4.

Functionally, there are only a few simple requirements for KubeWatt:

1. It must be able to read Kubernetes CPU utilization for both node and container resources.
2. It must be able to obtain node power usage from a source of power.
3. It must be able to split power usage into static and dynamic parts.

4. It must produce Prometheus-style metrics for container power usage.

The first two requirements are obvious: in order to calculate power usage of a container, KubeWatt must at least know the total power usage of the node that container is running on, and it must know the amount of resources that container uses on the node.

The third requirement is one that is of considerable importance to the workings of KubeWatt, and is something that we also have seen in Kepler. Dividing the power usage into 'static' (or 'idle' in Kepler) and 'dynamic' parts, aims to account for the power usage of a Kubernetes cluster and server when no workloads are running. The simple act of turning on a server and running a Kubernetes cluster uses some amount of power which cannot be attributed to any specific container. KubeWatt therefore splits the total power into two components, such that the static power can be indicated in total, and the dynamic power, which is the difference between static and total power, can be attributed amongst Kubernetes containers. KubeWatt will indicate static power as a single number in its output, since it indicates overhead that is not easily attributed to any specific container. Note that this definition of overhead includes Kubernetes control plane containers. These are therefore excluded from the dynamic power attribution. Any additional power usage incurred by the control plane will instead be attributed to other running containers causing the control plane to use power.

6.1. KubeWatt modes

KubeWatt runs in one of three modes. The first two modes, 'base initialization' and 'bootstrap initialization' are initialization modes. They run as a one-off job to initialize KubeWatt parameters. These modes analyze the cluster which KubeWatt will run on and find the static power value for each node in the cluster. This value is calculated once and not updated without manual intervention; therefore, we assume that this does not change over time.

Assumption 1 *The static power of a server does not significantly change over time.*

The third mode of KubeWatt is the 'estimator'-mode. It receives the result of the initialization and produces the Prometheus metrics for containers. We will discuss each mode in more detail below.

6.1.1. Base initialization mode

The base initialization mode is the simplest mode that KubeWatt can run in. It expects an empty Kubernetes cluster running no more than the Kubernetes control plane. KubeWatt expects its user to specify which pod names are part of the control plane as a set of regular expressions, such that it can validate the cluster is indeed empty before starting.

Over a period of 5 minutes, KubeWatt measures the power usage per node every fifteen seconds, which is averaged to produce the static power value per node. We expect that this mode will produce the most accurate results for the static power value, as it directly measures the idle cluster.

6.1.2. Bootstrap initialization mode

The bootstrap initialization mode is an alternative to the base initialization mode. It attempts to find the static power value from a cluster which is already running workloads that cannot be turned off for testing. Since this mode makes an estimation of the static power value based on measurements, the base initialization mode should be preferred if possible.

This mode gathers both CPU-usage data for each node and power usage data for each cluster node. Data is gathered every fifteen seconds for half an hour. Afterwards, KubeWatt checks whether the data has enough variability and a sufficient distribution to draw conclusions from. If not, data collection is repeated. If the data is sufficient, KubeWatt continues with data analysis.

To gauge whether data is sufficient to perform analysis with, KubeWatt checks the rough distribution of the data. The collected CPU-usage values are placed in buckets. KubeWatt then checks the amount of measurements in the largest bucket, and validates that no bucket has fewer values then some factor of this. By default, each bucket should have at least half as many values as the largest, to ensure a relatively uniform distribution. The buckets are 10% in size between 20% and 80% CPU utilization by default. All of these values are configurable in the KubeWatt configuration files. Changing the bucket parameters may be useful in a cluster which never reaches

as low as 20% CPU. This does not hold for the upper bound, since the user can run an artificial stressor to generate higher CPU-load. During the evaluation of KubeWatt we will explore what effects such a restricted range of measurements may have on the output of the bootstrap initialization mode.

As previously named, the static power should include the Kubernetes control plane utilization at idle. To achieve this, the CPU-usage of the control plane containers is gathered at the same time as the node CPU-usage and power usage. The set of control plane containers is known as the user is required to specify these, as also named above. The control plane CPU utilization is averaged to give an indication of stable control plane CPU-usage. Note that we cannot expect the control plane CPU utilization to remain stable when the cluster has load. As the idle control plane utilization is encapsulated in the static power value, any power usage caused by higher utilization in result of cluster load will be attributed amongst the containers causing this load.

To finally derive the static power usage, a third-degree polynomial regression is performed on the collected data. From initial testing, a third-degree polynomial tended to fit the data best, though this will be evaluated in the next section. The polynomial coefficients are subsequently used to find the estimated power usage at the average CPU-usage of the control plane. This then gives us the static power for each node in the Kubernetes cluster.

6.1.3. Estimation mode

The estimation mode is what we consider the 'main' mode of KubeWatt. This mode takes the output from either initialization mode as input and actually estimates the amount of power that each container in the Kubernetes cluster uses. When running in this mode, KubeWatt exports metrics to Prometheus (see Appendix A).

To achieve this, we use the model proposed in [21], named 'pod mapping'. We do, however, make a few changes. In [21] the following model is proposed to map bare-metal power consumption to virtual machines, and virtual machine power consumption to pods:

Define $\text{power}(\cdot)$ as the power usage of some component and $\text{cpu}(\cdot)$ as the CPU utilization of some component.

Let b_i be some bare-metal machine of which $\text{power}(b_i)$ and $\text{cpu}(b_i)$ are known. Let $v_{n,i}$ be some virtual machine identified by n running on b_i. Then, given $\text{cpu}(v_{n,i})$ as known we can define

$$\text{power}(v_{n,i}) = \text{power}(b_i) \cdot \frac{\text{cpu}(v_{n,i})}{\text{cpu}(b_i)} \tag{2}$$

as the power usage for each VM. Then, let $p_{m,n}$ be some pod identified by m running on a Kubernetes cluster on the set of v where the pod runs on $v_{m,n}$. Given $\text{cpu}(p_{m,n})$, then:

$$\text{power}(p_{m,n}) = \text{power}(v_{n,i}) \cdot \frac{\text{cpu}(p_{m,n})}{\text{cpu}(v_{n,i})}. \tag{3}$$

Together, Equations (2) and (3) map the power of a bare-metal machine through a virtual machine to the Kubernetes pod level, based on CPU utilization.

We make some changes for our case. We define $\text{power}_d(\cdot)$ and $\text{power}_s(\cdot)$ as the dynamic and static fractions of power as described above, respectively, with

$$\text{power}(\cdot) = \text{power}_d(\cdot) + \text{power}_s(\cdot).$$

Then let n_i be some Kubernetes node whose power $\text{power}_d(n_i)$ and $\text{power}_s(n_i)$ are known. Let $c_{m,i}$ be a Kubernetes container running on node n_i, and identified uniquely by m. Given the CPU utilization of $c_{m,i}$, we have

$$\text{power}(c_{m,i}) = \text{power}_d(n_i) \cdot \frac{\text{cpu}(c_{m,i})}{\sum_i \text{cpu}(c_{m,i})}. \tag{4}$$

Importantly, we make the distinction between $\sum_i \text{cpu}(c_{m,i})$, the CPU usage of all containers on a node combined, and $\text{cpu}(n_i)$, as the metrics API of Kubernetes includes overhead CPU usage such as system processes in the latter metric, which is already included in static power and should not be attributed to Kubernetes containers [48]. Equation (4) gives us a metric of power usage for each container in a Kubernetes node as derived from the node power usage and container CPU utilization.

Both the model proposed in [21] and our own make an important assumption:

Assumption 2 *Any Kubernetes cluster node is the only workload running on the underlying machine whose power is measured, that being a virtual or bare-metal*

machine.

This is an important distinction since this constraint makes it possible to measure the power usage of some Kubernetes node accurately by gauging the power usage of the underlying virtual or physical machine.

6.2. Power collector

The 'power collector' component of KubeWatt is responsible for providing a measure of power usage in Watts per node in the Kubernetes cluster. In our implementation, the only implemented version of this interface is the `RedfishPowerCollector` class, which uses the Redfish API of iDRAC in `SUT` to obtain power usage from the power supply. KubeWatt does not care about the source of power and the implementation is abstracted behind the `PowerCollector` interface, meaning that it is easily extended to use other power sources.

The `RedfishPowerCollector` collects power information from server management interfaces which implement the Redfish API. For each Kubernetes node named in config field `collector.node-names`, KubeWatt expects a corresponding Redfish entry in the `collector.power.redfish` map. Each of those must contain a host, username, password and list of Redfish ComputerSystem names. Since a single Redfish API can return multiple systems [12], we require that the user of KubeWatt specify which system(s) correspond to which Kubernetes node. The power readings for each system are summed per Kubernetes node to provide the final power values.

6.3. Kubernetes metrics collector

The Kubernetes metrics collection component is responsible for obtaining CPU usage metrics of both nodes and pods in the Kubernetes cluster. It uses the `metrics.k8s.io/v1beta1/pods` and `metrics.k8s.io/v1beta1/nodes` Kubernetes API endpoints for pods and nodes metrics respectively. The nodes endpoint returns CPU utilization in ns for each node. The pods endpoint returns CPU utilization in in ns for each container in each pod [48]. In the Kubernetes Java API implementation, which KubeWatt uses, the pods metrics are available per namespace [49].

6.4. Assumptions

In this section, we have made several assumptions which influence the architecture and implementation of KubeWatt. We now discuss these assumptions, show why it is safe to make these assumptions, and discuss what situations may occur in which these assumptions do not hold.

6.4.1. Assumption 1: The static power of a server does not significantly change over time.

We assume that the static power consumption of a server does not significantly change over time. To validate this, let's consider the energy usage of `SUT`. Note that this server is deployed under less-than-ideal conditions, in an archival room with fluctuating temperatures.

In Figure 6.1 the power usage of `SUT` between 2024-08-05 11:00 and 2024-10-14 11:00 is shown. The large peaks in this figure indicate the periods in which tests were run, leading to high power usage for a short period of time.

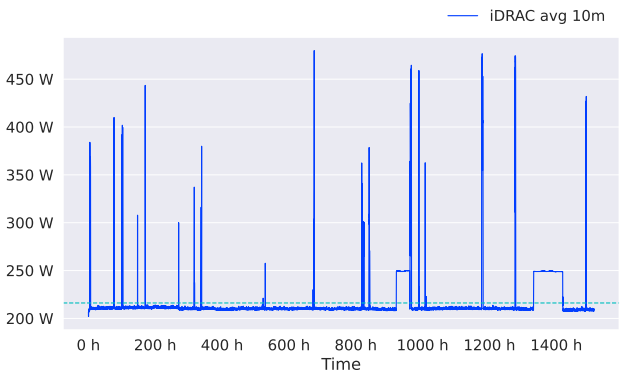
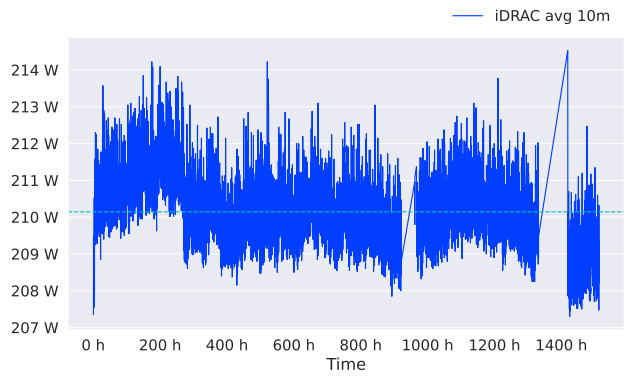


Figure 6.1: Power usage of `SUT` over ~2 months

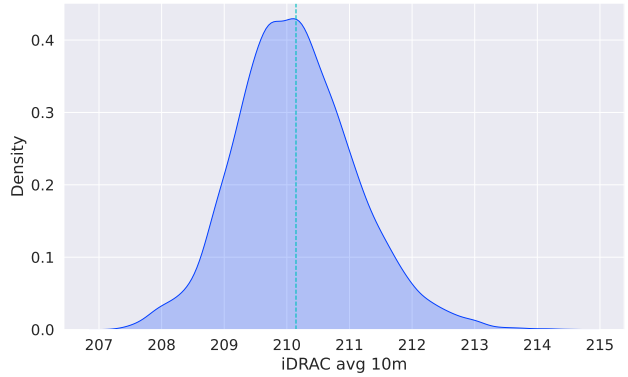
Power usage at idle, the static power, is indicated in Figure 6.2a. The data was filtered by removing measurements equal to and above 215 W and equal to and below 207 W as this removes the peaks without affecting the data at idle power usage. These filter values were obtained by visual inspection of the data. Note that these metrics include the Kepler deployments as discussed in Section 5.1. This results in more dynamic and noisy data than otherwise expected with a completely empty cluster. For all the KubeWatt tests, these Kepler deployments have been removed, and we will subsequently observe a lower static-power value that we see here. These deploy-

ments were still active during the two-month observation shown above, as this was the period in which the Kepler experiments (Section 5.2) were running.

From Figure 6.2a, we see that power usage is quite consistent over the measured period. The adjusted data has a mean of 210.15 W with a standard deviation of 0.95, which is an error of just 0.45 % from the mean. This distribution is also shown in Figure 6.2b. This indicates that the static power usage of the server does not significantly change over time.



(a) Power usage at idle of SUT over ~2 months



(b) Histogram of power usage at idle

Figure 6.2: Power usage at idle of SUT as measured by iDRAC

In a situation where this assumption does not hold true and the static power load of a server *does* change over time, the behavior of KubeWatt depends on how this value changes. If the value is very noisy, but the mean value over time is consistent, then the readings over time of KubeWatt will remain consistent. When considering small time-windows, KubeWatt will be somewhat inaccurate in this situation as its exact values of static and dynamic power will not be completely correct. As the actual power usage drops below the static power value as KubeWatt knows it, dynamic power will become zero and no power is allocated to containers. If the value is not noisy but changes, for example, based on room temperature over a longer period of time then KubeWatt will under- or overestimate the amount of static power and thereby over- or underestimate the amount of dynamic power, respectively, and container power correspondingly.

6.4.2. Assumption 2: Kubernetes lives alone

We have assumed that a Kubernetes node is the only measurable workload running on some underlying machine whose power is measured. In the case of KubeWatt's proof-of-concept implementation, this is very important. Since power is measured through Redfish, KubeWatt only knows how much power the entire server in question is using. If there are other significant workloads running on that server besides the Kubernetes cluster under test, KubeWatt does now know how much of this power usage is attributable to Kubernetes and how much is not.

This assumption does not need to hold in general. A potential workaround to this requirement is to use the power mapping model discussed in Section 6.1.3 to map the used power of an entire system to separate workloads as if these were, for instance, their own virtual machines. Such mapping is extensively discussed in [21], so this work does not explore that possibility.

7. KubeWatt Evaluation

To evaluate KubeWatt we ask similar questions to how we evaluated Kepler. Recall that these are the same questions as posed in Section 1.

KWRQ1 How accurate are KubeWatt's total node measurements compared to a ground truth?

KWRQ2 How (well) does KubeWatt attribute power usage to containers on the node?

Obviously, these questions hold for KubeWatt's 'estimator' mode. As we evaluate KubeWatt by considering each of its modes separately, we additionally ask for the initialization modes:

KWRQ3 How accurately can KubeWatt's base initialization mode report the static power value?

KWRQ4 How accurately can KubeWatt's bootstrap initialization mode estimate the static power value?

The testing setup as discussed in Section 3 is used; however, the four Kepler deployments have been removed as these are no longer necessary. As depicted, KubeWatt is now deployed.

7.1. Experiment design

For the initialization modes, we want to verify that the output that KubeWatt gives is equal to the static power of the system. We have seen the actual static power value in Assumption 1. Note that this value includes the Kepler deployments that could not be filtered from the data.

To evaluate the base initialization mode, we run the KubeWatt job on `SUT` with nothing else running on the cluster. The following list of pod names is provided to KubeWatt as control plane: `nfs-.*`, `calico-.*`, `canal-.*`, `coredns-.*`, `metrics-.*`, `tekton-.*`, `kubewatt-.*`. Note that `tekton-.*` is not technically part of the control plane; however, it could not be easily removed and as it is idle, it should not have a significant impact on findings.

We repeat this six times in sequence, then verify that the output value closely corresponds to the expected power value. To obtain the expected power value, the power use reported by iDRAC is tracked during the runtime of each of the initialization job runs.

For the base initialization mode we evaluate whether the resulting values are consistent over multiple runs and whether the resulting values are accurate to the expected value.

To evaluate the bootstrap evaluation, we run a best-case test, where the cluster will be stressed with a random stressor. Stress-ng is used to create stressors at a random CPU level between 1–64 that last three minutes. This creates a CPU load that should be uniformly distributed across the entire CPU range of `SUT`.

Since it is not possible to use a synthetic load to *decrease* the CPU utilization below what a running system normally has, KubeWatt allows modification of the minimum CPU value it checks for when validating the data. Tuning this value avoids a situation where KubeWatt's bootstrap initialization never finishes due to insufficient data in a bucket which will never get any data. Note that KubeWatt does *use* the data outside of buckets, it only does not validate that it exists prior to continuing with the analysis. To show the effect a limited range of data has on the output of KubeWatt we make cuts of the dataset obtained by the earlier tests, then run the regression as KubeWatt would to observe how the output would change.

To evaluate the estimation mode we repeat some of the tests that were also performed for Kepler (see Section 5.1.2). We repeat the single-stressor test as a baseline validation. We do not repeat the node component test, as KubeWatt only measures CPU utilization to determine power attribution ratios. This limitation and its implications are discussed in Section 7.3. Additionally, we perform a test with multiple stressors that start and stop at different times, to investigate how KubeWatt handles a more dynamic environment. Lastly, to validate that KubeWatt indeed works better than Kepler, we repeat the test where we deleted inactive pods with Kepler, as this is the test where we saw Kepler perform worst.

7.2. Results

7.2.1. Base initialization mode

Figure 7.1 shows the measured power values by iDRAC during the runtime of the test. Recall that KubeWatt is able to read these values directly, using the Redfish API. The figure shows the aggregation of data during six sequential runs of KubeWatt's base initialization mode. Note that this value is lower than what was observed in Section 6.4.1, as the Kepler deployments have been removed.

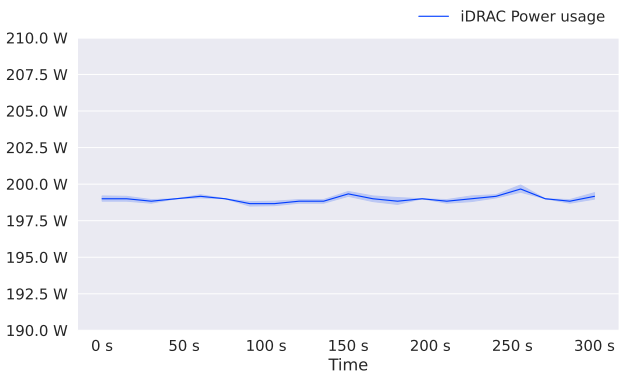


Figure 7.1: iDRAC power measurements during the base initialization tests

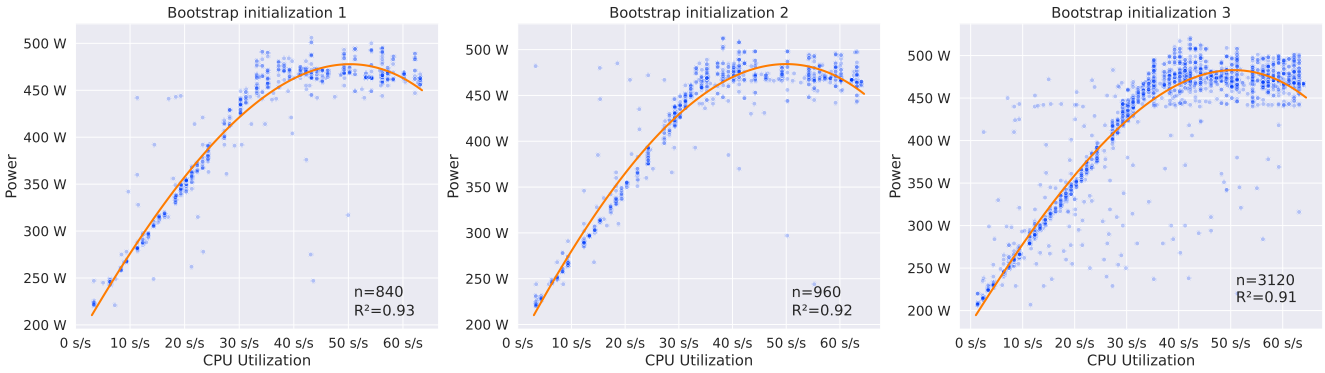


Figure 7.2: The measured values by KubeWatt bootstrap initialization mode, for each of the three best-case tests

KubeWatt reported a static power of 198.9 W, 199.15 W, 199.1 W, 199.1 W, 198.75 W, and 199.15 W. Compared to the raw data we collected in Figure 7.1 these observations have a z-score between -0.47 and 0.31. This indicates KubeWatt is able to consistently and accurate report the static power using base initialization mode.

7.2.2. Bootstrap initialization mode

To evaluate the bootstrap initialization mode the best-case test as defined in Section 7.1 is performed three times sequentially. The three tests resulted in a static power value for the tested node of 180.8 W, 178.2 W and 185.2 W. While consistent with each other, these results indicate that the bootstrap initialization mode slightly underestimates the static power value compared to the base initialization mode, which we have confirmed to be accurate. The estimated values deviate from the mean expected value by at least 6.9 % to at most 10.5 %.

To explore the data that is collected by KubeWatt, consider Figure 7.2. This figure shows, for each of the three test runs, the data for CPU utilization and power usage. Note that each test has a considerably different number of samples; this is because each test took a different amount of time to run to completion. Test 1 took ~3.5 h, test 2 took ~4 h, and test 3 took ~11 h to finish. This difference can likely be attributed to 'unluckiness' in the data distribution when dividing data into the buckets, where buckets may get filled non-uniformly for a low number of samples, even if the sample distribution is uniform.

For each of the tests, Figure 7.2 shows a scatter of measured values (blue) and the associated third-degree polynomial regression line (orange). KubeWatt evaluates this regression line at the CPU utilization of the control plane to find the static power value. The measured CPU utilization values of the control plane were, respectively, 0.0406 s s^{-1}, 0.0394 s s^{-1} and 0.0393 s s^{-1}[10]. According to the calculated regressions, this indicates the control plane accounts for 0.43 W of power usage.

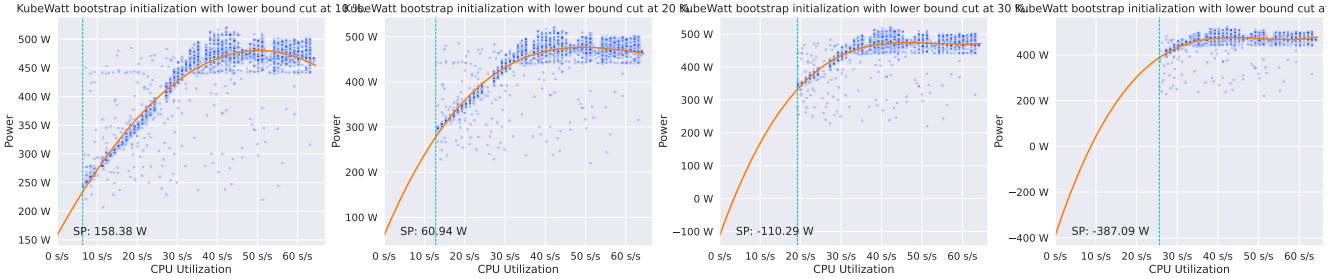
From the above we can conclude that the bootstrap initialization mode works moderately well when provided with good data. We found that it deviates at most approximately 10 % from the expected value in our measurements. Due to this error we recommend using the base initialization mode over the bootstrap initialization mode wherever possible.

Influence of minimum CPU-value

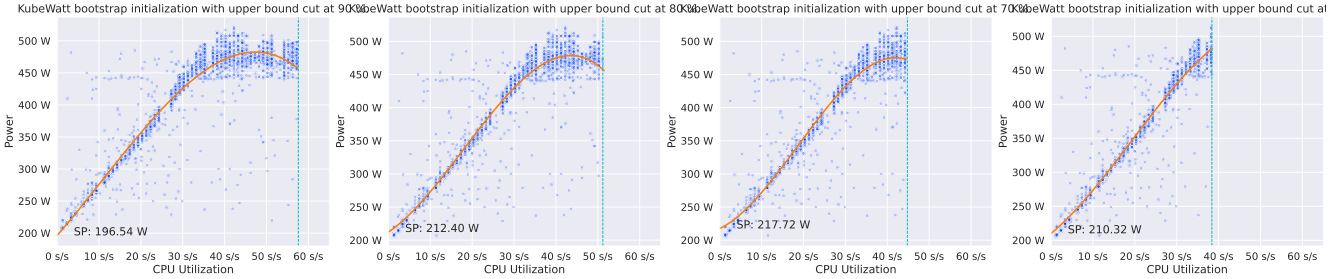
As explained in Section 7.1, to evaluate the effect of a narrower data range on the output of the bootstrap initialization mode, we take cuts of the data that was acquired for the best-case scenario test, where any data under a specific amount of CPU-utilization is removed. These cut datasets represent the data that KubeWatt *would* have obtained in such situation where the CPU utilization range is limited. On these cut datasets, we perform the regression as KubeWatt would and explore how its coefficients change based on how much the data domain is limited.

Figure 7.2 shows the original data. This data has a full CPU utilization range. We combine the three tests into a single dataset for these tests. For the lower bound cuts, the data is cut at 10 %, 20 %, 30 %, and 40 %. This is illustrated in Figure 7.3a. For the upper bound, the

[10]The unit s s^{-1} indicates a unitless measure of CPU utilization that is not dependent on the amount of cores in a system. A 1 s s^{-1} utilization means a single core of the machine is working at 100 % of its capacity (or two cores at 50 %, etc).



(a) Combined dataset of bootstrap initialization (best case) cut at a lower bound of 10 %, 20 %, 30 %, and 40 % CPU utilization.



(b) Combined dataset of bootstrap initialization (best case) cut at an upper bound of 90 %, 80 %, 70 % and 60 % CPU utilization.

Figure 7.3: Different domain cuts of the bootstrap initialization test. Orange shows the third-degree polynomial regressions, SP shows the associated static-power value.

data is cut at 90 %, 80 %, 70 % and 60 %; illustrated in Figure 7.3b.

From these figures as well as the resulting static power values as indicated in the figures, we see that the lower bound of the collected data has a significant influence on the outcome of the bootstrap initialization. At a minimum of 10 % CPU-utilization we see that the static power value drops to 158 W. For 20 % this is 60 W. Both are sufficiently far from the expected value that they cannot be considered accurate. For the 30 % and 40 % lower-bound cuts we see that the expected static power value becomes negative. This is obviously not possible. By considering the shape of the data in this figure as well as in Figure 7.2 we see that these results do make intuitive sense. Above ~32 s s^{-1} CPU-utilization, utilization does increase while power usage does not. When we consider more of this non-changing data in our regression, it cannot properly quantify the relation between CPU-utilization and power usage in the lower domain. A cut in the upper bound does not have such a significant influence on the result as did cutting the lower bound. This is illustrated in Figure 7.3b. We see that the more aggressive the cut, the higher the intercept value tends to be.

The knee in data that we have observed in Figures 7.2, 7.3a and 7.3b might be caused by simultaneous multithreading (SMT), a CPU feature where one physical core is presented as two logical cores to the operating system. It is more commonly known by its Intel implementation named Hyper-Threading [50]. Since this feature allows 32 threads to saturate all cores if one thread runs on each CPU, we could expect to see a knee such as the one observed, if the processor optimizes performance when not all 64 threads are in use. To confirm that this is indeed why we observe the knee, the tests above are repeated with Intel's Hyper-Threading turned *off* on SUT. These results are presented in Figure 7.4.

As predicted, the knee in the data disappears, and we observe a result more in-line with what was expected based on [4]. A linear regression of the combined data of the three tests gives an R^2-value of 0.94, indicating a very good fit of the linear regression. The three bootstrap initialization tests resulted in outputs of 201.7 W, 200.0 W and 199.9 W which have an error to the expected value of 199.1 W of between 0.4 % and 1.3 %; much closer than the polynomial approach used with SMT enabled and the full range of values considered.

A repeat of the 'cut' tests are shown in Figures 7.5a and 7.5b. The predicted power usage values now stay much more consistent as the data range is limited. In

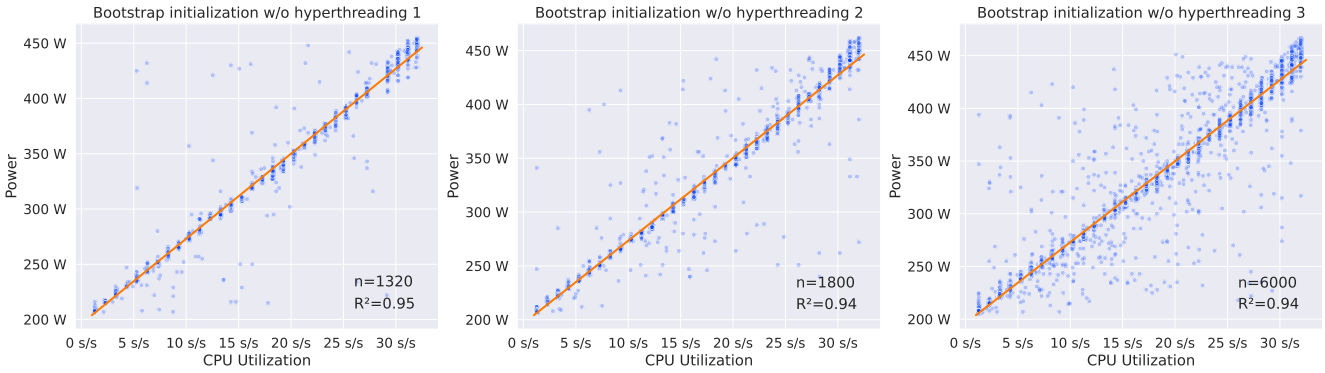
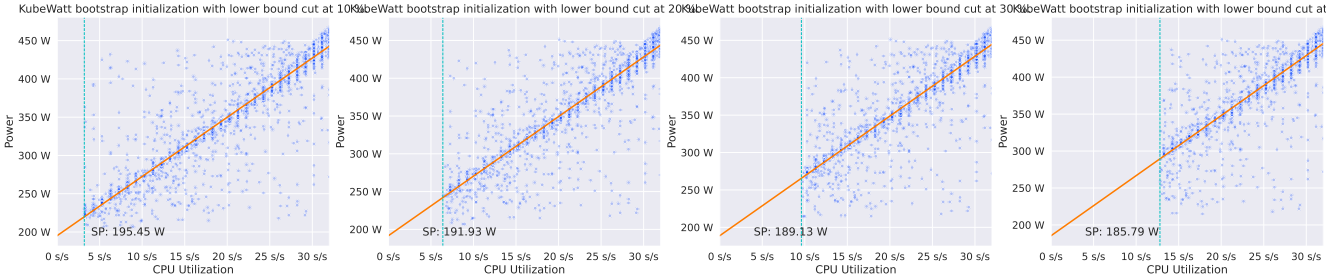
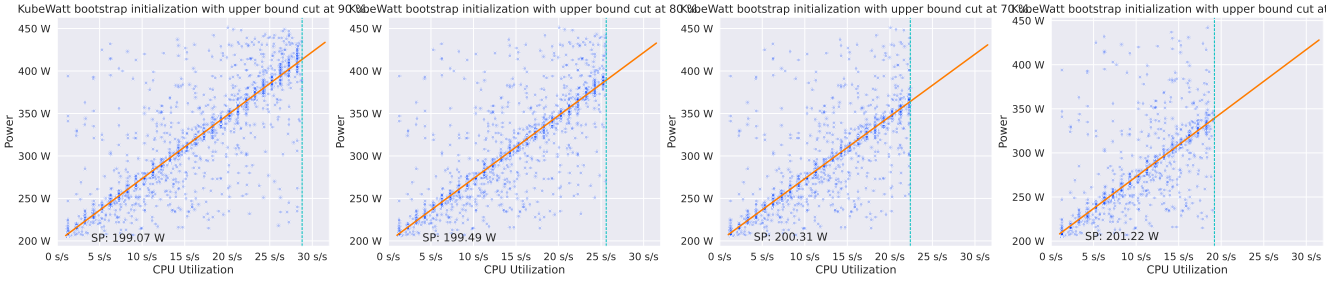


Figure 7.4: The measured values by KubeWatt bootstrap initialization mode with SMT disabled, for each of the three best-case tests



(a) Combined dataset of bootstrap initialization (best case without simultaneous multithreading) cut at a lower bound of 10 %, 20 %, 30 %, and 40 % CPU utilization.



(b) Combined dataset of bootstrap initialization (best case without simultaneous multithreading) cut at an upper bound of 90 %, 80 %, 70 % and 60 % CPU utilization.

Figure 7.5: Different domain cuts of the bootstrap initialization test, with SMT disabled. Orange shows the third-degree polynomial regressions, SP shows the associated static-power value.

Figure 7.5a we see that the lower-bound cuts do result in a deviation in the static power value as the data range decreases. A 10 % cut yields a 1.83 % deviation from the expected value, a 20 % cut yields a 3.60 % deviation, a 30 % cut yields a 5.01 % deviation and a 40 % cut yields a 6.67 % deviation. Notably, each of these is much better than even a 10 % lower-bound cut was for the third-degree polynomial regression without considering SMT. As before, we observe very little change in the static power value when changing only the upper bound, as illustrated in Figure 7.5b. The most aggressive upper bound cut of 60 % resulted in a value 1.06 % higher than expected.

As the no-SMT test has a much better result than before, a simple change is introduced in KubeWatt: when SMT is enabled on a node, the bootstrap initialization ignores the top 50% of CPU utilization data. When it is disabled, all data is used. KubeWatt then runs a linear regression as opposed to a third-degree polynomial regression as we now expect the relation between CPU-utilization and power-usage to be linearly quantifiable.

7.2.3. Estimator mode

For the estimator-mode tests, Hyper-Threading has been re-enabled on SUT. This should result in a more realistic

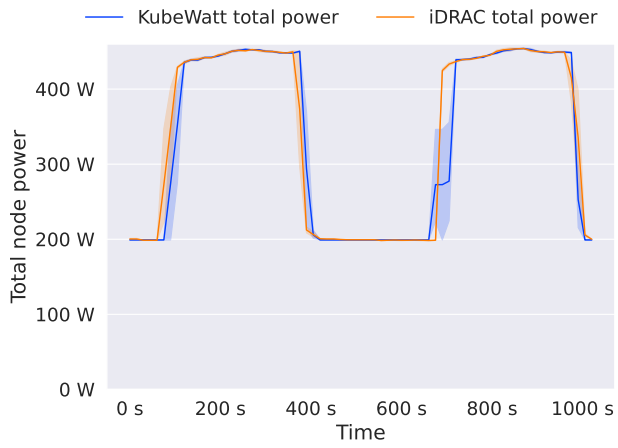


Figure 7.6: Sum of container+static power for KubeWatt (blue) and iDRAC node power (orange), combined for the three performed single-stressor tests.

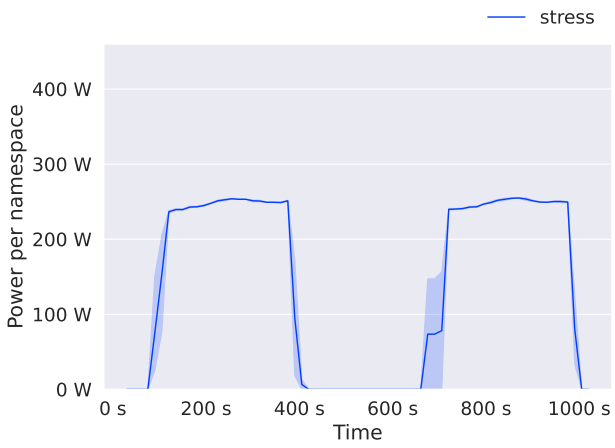


Figure 7.7: Sum of container power per namespace for KubeWatt, combined for the three performed single-stressor tests.

evaluation since we expect most servers to use some form of SMT. Hyper-Threading is enabled by default on the Intel CPU that SUT uses [50].

Single-stressor tests

Results of the single-stressor tests are shown in Figures 7.6 and 7.7. The first figure shows the total power that KubeWatt thinks our node SUT is using as well as the direct iDRAC measurements. The second figure shows the dynamic power usage of each namespace according to KubeWatt. Both figures are the aggregate results of three repeated tests.

In Figure 7.6 we see that the total power reported by KubeWatt is very similar to what iDRAC reports our server to be using. The total power usage has an RSME of 10.56 %, where the error is biggest when a load is just starting up. In this case this error occurs as there is a delay in CPU utilization and Kubernetes metrics API reporting this utilization. As KubeWatt measures only a single container, the total CPU utilization is reported at zero while waiting for metrics, leading to KubeWatt not attributing the dynamic power to any container for a short period. If we instead consider the area under the curve which indicates the total power used, the total error is less than 1 %.

The observed error during this test is smaller than the error we have seen for Kepler's best-case scenario, where an RSME of 18.70 % was measured, though here too, the error was influenced by measurement latencies.

Figure 7.7 contains some noteworthy differences compared to the same test performed for Kepler in Figure 5.3: only the dynamic power is shown for each container by KubeWatt; this is, as explained above, by design. Static power is not attributed to specific pods and is reported separately (not shown). Moreover, not all namespaces are shown. KubeWatt does not indicate power usage by control plane pods, as their power usage is (partially) included in the static power value when this is measured or estimated. Higher control plane power usage is instead attributed to workload containers that are not part of the control plane.

Multi-stressor tests

In addition to the simple single-stressor test which verifies the basic functionalities of KubeWatt, we also consider a test with multiple stressors. This will be particularly interesting considering the SMT results that we have seen for the bootstrap initialization mode in Section 7.2.2.

This test runs four stressors, each taxing sixteen CPUs. In total this will stress the whole system at 100 % CPU-utilization. We run this in four phases, where each phase has one more stressor than the previous thereby ramping up the CPU utilization.

The results of this test are presented in Figure 7.8. In the figure we see the four phases of the test. The left figure indicates the power that KubeWatt attributed to each of the 1–4 containers running stressors during the test. As each of the containers should have exactly the

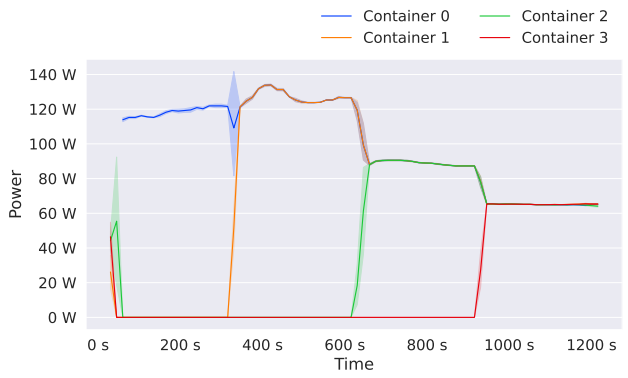


Figure 7.8: Power usage over time of the three repeated multiple-stressors tests.

same amount of CPU utilization, we expect KubeWatt to report the same power value for each of the containers. This is indeed confirmed. We see that the power per container goes down from before when stressing 48 or 64 CPU cores.

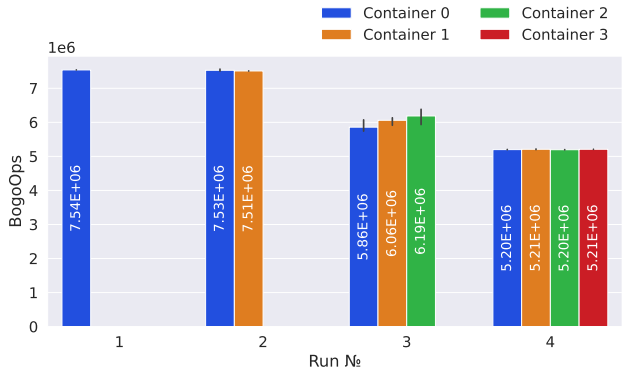


Figure 7.9: Total benchmark throughput per phase of the three repeated multiple-stressors tests.

Figure 7.9 shows the total throughput per stressor per phase in 'bogus operations' for each of the five-minute stressors as reported by stress-ng. We observe a similar trend in throughput as we do in power consumption per container. This explains why we see the power decrease, as the throughput per container also decreases when running above 32 CPU threads in total. This finding is inline with the SMT-findings in Section 7.2.2, where we also observed that 32 threads are able to fully utilize the 32 CPU cores even though 64 threads *can* also run simultaneously. Note that cAdvisor, the source for Kubernetes container metrics, does not report this difference in throughput. Since it only reports the higher-level metric of CPU seconds, it cannot report whether a thread is using a full core or is sharing a core with another thread when SMT is enabled. The reported CPU seconds per container are shown in Figure 7.10, for reference.

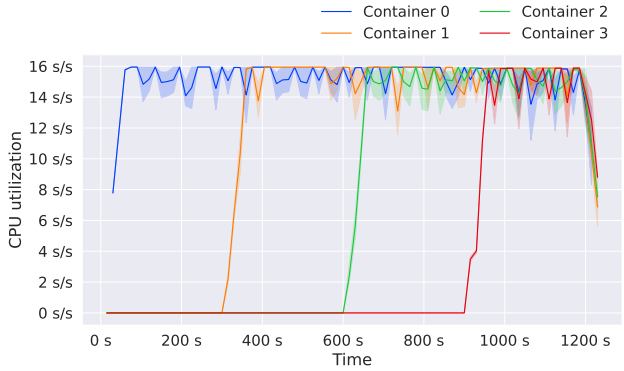


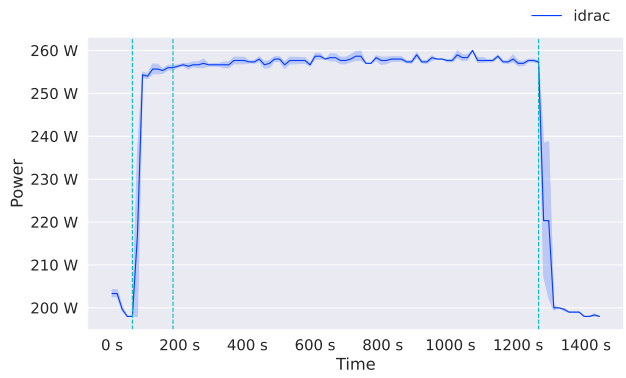
Figure 7.10: The reported CPU utilization in CPU seconds for each of the four stressors the multiple-stressors test.

Note that in Figure 7.8, KubeWatt reports power usage when the containers are all created for containers 1, 2 and 3 even though these are idle. As per KubeWatt debug information, this is caused by the power collector reporting power usage because of container 0, while the Kubernetes metrics pipeline is not yet reporting container 0. As containers 1, 2 and 3 are the only containers KubeWatt knows about, they are attributed all the dynamic power that is reported, even though they have very little CPU-utilization. This problem corrects itself as soon as Kubernetes metrics includes all newly created containers.

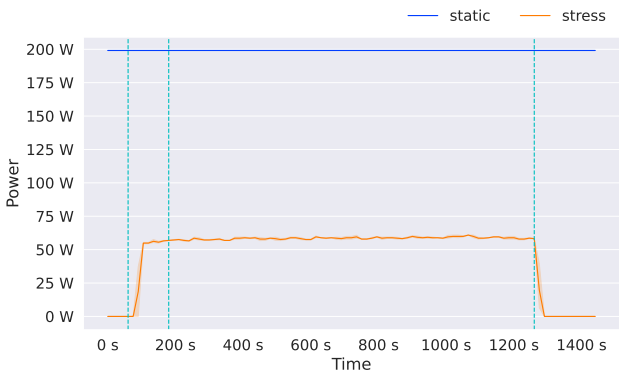
Deleting inactive pods, repeated

To validate that KubeWatt does not suffer from some of the more severe limitations we saw in Kepler, we repeat the inactive pods deletion test (Kepler result depicted in Figure 5.5). The result of this test for KubeWatt is shown in Figure 7.11.

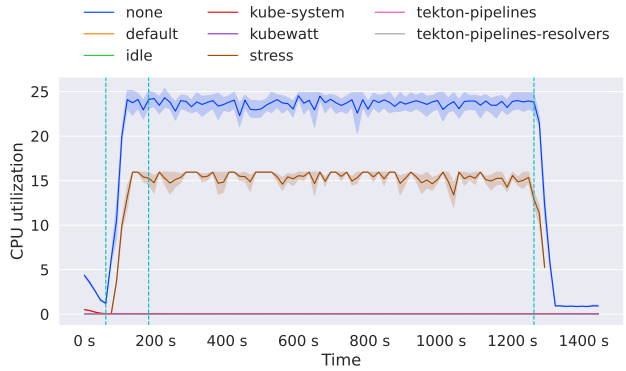
In this figure, we see results exactly as expected. In Figure 7.11a the total system power as reported by iDRAC is reported, for reference. In Figure 7.11b the power per namespace according to KubeWatt is shown. Note that 'static' is not a namespace, but is the namespace-less static power. All other namespaces in this figure are dynamic power. Some namespaces here are missing compared to Figure 5.5b; this is because all containers in those namespaces are part of the control plane and are not reported by KubeWatt. Note also that



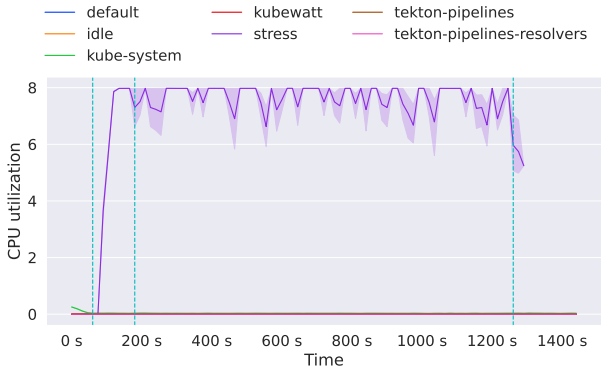
(a) The total system power usage



(b) The total power per namespace



(c) The CPU-utilization per namespace; all cAdvisor values



(d) The CPU-utilization per namespace, containers only

Figure 7.11: The result of deleting inactive pods. The blue markers indicate the times at which (1) the stressing load was started; (2) the inactive pods were deleted; (3) the stressing load was stopped, respectively. The tests were repeated four times.

the idle containers are not reported. This is the case because they are not using power: they are not running. Figures 7.11c and 7.11d show the CPU utilization for each namespace. We observe the same measurements as in Figures 7.11c and 7.11d, indicating the test was performed equally.

The main result is depicted in Figure 7.11b. We see that the power usage of the stressor pod goes up with CPU utilization, stays consistent throughout the test and goes down when the stressor is stopped. Conclusively, we see from these figures that KubeWatt can accurately report the power usage of the stress pod, even when a large number of idle pods are created and deleted during the test.

7.3. Discussion

With the results and amendments to KubeWatt that have been discussed, we can now turn again to the questions posed at the beginning of this section.

KWRQ1: How accurate are KubeWatt's total node measurements compared to a ground truth?

In Sections 7.2.1 and 7.2.2 we have seen that KubeWatt is able to use an external source of power to gather accurate readings of node power. For the initialization modes, this works well.

KubeWatt has one limitation in regard to total node power as it pertains to the estimator mode. The issue occurs when KubeWatt does report dynamic power usage, but has no containers with CPU utilization. Since the total utilization is zero, KubeWatt cannot attribute the dynamic power, which is therefore not reported. This issue occurs when a container is newly created and immediately generates high load, as we have seen in Section 7.2.3. In other situations, the total node power measurement is sufficient for the estimator mode.

KWRQ2:How (well) does KubeWatt attribute power usage to containers on the node?

We have considered container power attribution in Section 7.2.3. In general, we see that KubeWatt is able to accurately portray the power used by a container based on CPU load, in line with what is expected based on CPU utilization, throughput values and the observed power curve in Section 7.2.2 for the SMT-enabled case.

KubeWatt does incorrectly attribute power when containers are newly created and immediately generating load when other containers already exist. Since the Kubernetes metrics API does not immediately report this container and has a delay in metrics in general, its power is misattributed to other containers. However, this only causes a short peak in container power usage values and is resolved as soon as the new container is known by cAdvisor. The delay in metrics of the Kubernetes metrics API that does not align well with the power metrics may lead to more noisy or incorrect power attribution when containers are very turbulent in their power usage. Since data is collected every 15 s, KubeWatt only works well for clusters with sufficiently stable CPU-utilization among containers.

A limitation of KubeWatt's power attribution is that it can only attribute power based on CPU-utilization. In a general purpose computer, this is likely not much of a problem since the CPU is the biggest user of power; however, when we consider a system that uses accelerators such as a GPU or FPGA then power for containers using these will be misrepresented. KubeWatt similarly does not measure RAM power usage and utilization; however, we have seen with RAPL that RAM itself is likely of little influence to total power usage.

KWRQ3:How accurately can KubeWatt's base initialization mode report the static power value?

The base initialization mode is able to very accurately read the static power value, and does so very quickly, in five minutes. In our test we have seen the base initialization mode get very consistent measurements over time, with the largest internal difference being 0.4 W out of a 199.1 W mean over six KubeWatt runs.

KWRQ4:How accurately can KubeWatt's bootstrap initialization mode estimate the static power value?

We have seen that the bootstrap initialization mode can quite accurately estimate the static power value, given that we account for simultaneous multithreading. In this case, KubeWatt can estimate the static power value within 0.4–1.3 % of the expected value with perfect data, and can estimate the static power value within 5 % of the expected value with data only above 40 % CPU utilization.

While not tested, the SMT-enabled version of KubeWatt's bootstrap initialization mode would need more data for similar results. As it cuts away the top 50 % of data to obtain data similar to the non-SMT case, a lower bound cut of 40 % CPU utilization would in fact be removing 80 % of data. As such, when SMT is enabled a lower bound of 20 % would yield a static power value within 5 % of the expected value.

8. Conclusion

So far we have seen an experimental evaluation of Kepler in Section 5 and the introduction and experimental evaluation of our own tool, KubeWatt, in Sections 6 and 7. In this section, we reflect on the questions posed in the introduction and provide answers to them based on the previous two parts. As all Kepler and KubeWatt specific research questions have already been discussed in Sections 5.3 and 7.3, respectively, we only reflect on our main research goal here. Furthermore, we discuss some threats to validity that hold for both the Kepler and KubeWatt part, and discuss potential avenues for related work.

In the introduction we asked the following question:

> How can we accurately measure or estimate the power usage of a Kubernetes containers based on external measurements?

To answer this question we have evaluated the state-of-the-art tool Kepler, where we have found several limitations: Kepler improperly attributes static power to non-running containers, and Kepler cannot properly attribute dynamic power to the responsible container in some cases. In order to address some of these limitations we have also built a proof-of-concept tool named

KubeWatt. During this project we have found several important aspects to consider and some inherent limitations when building such an energy-measuring tool. For instance, the total power of a system cannot be taken at face value but must be divided into static and dynamic parts for a fair container attribution. Moreover, a tool such as KubeWatt or Kepler that uses external sources for metrics must consider that these metrics may not always be up-to-date. In the case of KubeWatt we saw that this had an effect on power attribution when creating new containers. Kepler will suffer from the same issue to a lesser extent. Since Kepler uses eBPF to obtain resource utilization directly from the Linux kernel, it has more control over how these metrics are obtained. Additionally, Kepler uses the regular pods API to obtain the list of pods. Both of these have lower latency than the Kubernetes metrics pipeline.

We have evaluated that KubeWatt can accurately divide total power into static and dynamic power based on one of two initialization modes. We have furthermore evaluated its capacity to attribute power usage to running containers and have seen that it can produce accurate metrics for containers. Lastly, we have shown that KubeWatt performs better than Kepler in specific situations where we have seen Kepler's limitations.

One of the main limitations in KubeWatt is that it considers CPU-utilization to be the sole source of power usage. This may hold (approximately) for general-purpose computers, but does not hold as soon as an accelerator such as a GPU or FPGA is introduced into the system.

8.1. Threats to Validity

Some aspects of our research have not been perfect. It is therefore important to consider areas of improvement in context of the research that has been performed.

Our first Kepler-specific research question, KPRQ1, asks how accurate Kepler can be in terms of node power measurements. While we investigated RAPL and Redfish power sources, we did not investigate the other power sources that Kepler can read information from; these being NVML for GPU power and ACPI or IPMI for platform power [16]. Since iDRAC was able to provide very accurate readings, we have been able to accurate survey Kepler's performance nonetheless, as far as container metrics are concerned.

The main server that was used for running tests was not fully functional. Specifically, one of the twelve memory sticks raised bit errors occasionally. The iDRAC monitoring reported that "the system memory has faced an uncorrectable multi-bit memory errors in the non-execution path of a memory device". While test results were discarded when an error occurred during the test, the overhead of performing error correction on a defective memory device may cause more noisy CPU load and/or power-usage during tests and thus influence results and test repeatability.

The system under test comprised a single server. While this is fine for testing purposes, results would have been more significant by combining a number of servers and distributing multiple workloads to more closely resemble a real-world cloud environment. This would also allow for testing, for instance, the difference in Kepler's/KubeWatt's behavior on master and worker nodes. The use of a single server was decided simply due to availability.

Furthermore, the server in question was not deployed in an ideal scenario. A real-world cloud environment would be a datacenter with proper cooling and a stable fan inlet temperature. Our server was not deployed in a real datacenter due to the need for management port (iDRAC) access which the University of Groningen would not allow in their datacenters. The room where the server was installed had a temperature which fluctuated between 25 °C and 28 °C during testing which may cause fluctuation in total power-usage and influence repeatability of tests.

During testing, we only ran artificial workloads using stress-ng. While these stress the CPU and memory, they do so in a very specific way which may not always be equal to a real-world workload. Nonetheless, both tools that were considered should be able to accurately account for their power usage, and the use of such controllable stressors is important for consistent and repeatable tests with results that are low noise and easy to interpret.

8.2. Future Work

Several parts and findings within this research project call for future research. First, KubeWatt is currently implemented as a proof-of-concept tool that is not feature-

complete for many scenarios. It supports only a single type of power collector, and it cannot attribute power properly for systems that, for instance, also use GPUs. KubeWatt should be extended to include such features. Moreover, KubeWatt has not been tested on realistic or even real-world cluster workloads. Deploying KubeWatt alongside an actual workload and by doing realistic tests could provide useful insights in where KubeWatt can improve.

During the KubeWatt evaluation we found interesting findings with respect to SMT, which had not before been observed in this manner. Future research should reproduce these findings using a similar setup and should investigate why we observe the described 'knee' so clearly. The authoritative paper on the matter, the SPEC power benchmark [4] also has SMT enabled and does not observe a similar result. It may be a result of the way that Kubernetes/cAdvisor measure CPU utilization, or it may be a result of the specific architecture of the Intel Xeon Gold 6226R that `SUT` uses. It would be very useful to learn more about this behavior.

9. Acknowledgements

I want to thank the following people for their contribution to this thesis:

- **Prof. Vasilios Andrikopoulos** and **dr. Brian Setz** for their continued guidance as supervisors and for providing feedback and advice throughout this project.
- **Wim Nap** of the University of Groningen's High Performance Computing department for allowing me to borrow a server for the duration of this project, and **dr. Brian Setz** for his help installing the server.
- **Rosalien Kinds** for her help with designing the KubeWatt logo.

References

[1] Mark Pesce. "Cloud Computing's Coming Energy Crisis - IEEE Spectrum." (), [Online]. Available: https://spectrum.ieee.org/cloud-computings-coming-energy-crisis (visited on 10/30/2024).

[2] Paul Cormier. "The State of Enterprise Open Source: A Red Hat report." (), [Online]. Available: https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022 (visited on 10/30/2024).

[3] Sjouke de Vries, "Cloud Applications Cost profiling using Application Performance Monitoring," University of Groningen, Jul. 16, 2022, 59 pp.

[4] J. v. Kistowski, H. Block, J. Beckett, K.-D. Lange, J. A. Arnold, and S. Kounev, "Analysis of the Influences on Server Power Consumption and Energy Efficiency for CPU-Intensive Workloads," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15, New York, NY, USA: Association for Computing Machinery, Jan. 31, 2015, pp. 223–234, ISBN: 978-1-4503-3248-4. DOI: 10.1145/2668930.2688057. [Online]. Available: https://dl.acm.org/doi/10.1145/2668930.2688057 (visited on 04/11/2024).

[5] B. Everman, M. Gao, and Z. Zong, "Evaluating and reducing cloud waste and cost—A data-driven case study from Azure workloads," *Sustainable Computing: Informatics and Systems*, vol. 35, p. 100 708, Sep. 1, 2022, ISSN: 2210-5379. DOI: 10.1016/j.suscom.2022.100708. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537922000476 (visited on 04/11/2024).

[6] R. Douhara, Y.-F. Hsu, T. Yoshihisa, K. Matsuda, and M. Matsuoka, "Kubernetes-based Workload Allocation Optimizer for Minimizing Power Consumption of Computing System with Neural Network," in *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2020, pp. 1269–1275. DOI: 10.1109/CSCI51800.2020.00238. [Online]. Available: https://ieeexplore.ieee.org/document/9458062 (visited on 05/24/2024).

[7] S. Ghafouri, S. Abdipoor, and J. Doyle, "Smart-Kube: Energy-Aware and Fair Kubernetes Job Scheduler Using Deep Reinforcement Learning," in *2023 IEEE 8th International Conference on Smart Cloud (SmartCloud)*, Sep. 2023, pp. 154–163. DOI: 10.1109/SmartCloud58862.2023.00035. [Online]. Available: https://ieeexplore.ieee.org/document/10349157 (visited on 05/24/2024).

[8] G. Fieni, R. Rouvoy, and L. Seinturier. "SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers." arXiv: 2001.02505 [cs]. (Jan. 2, 2020), [Online]. Available: http://arxiv.org/abs/2001.02505 (visited on 06/01/2024), pre-published.

[9] G. Fieni, R. Rouvoy, and L. Seinturier, *SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers*, 2020. DOI: 10.1109/CCGrid49817.2020.00-45. [Online]. Available: https://github.com/powerapi-ng/smartwatts-formula (visited on 06/23/2024).

[10] M. Amaral, H. Chen, T. Chiba, *et al.*, "Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, Jul. 2023, pp. 69–71. DOI: 10.1109/CLOUD60044.2023.00017. [Online]. Available: https://ieeexplore.ieee.org/document/10254956 (visited on 05/23/2024).

[11] "Project Metrics," CNCF. (), [Online]. Available: https://www.cncf.io/project-metrics/ (visited on 11/08/2024).

[12] *Redfish Data Model Specification*, version 2024.3. [Online]. Available: https://www.dmtf.org/sites/default/files/standards/documents/DSP0268_2024.3.pdf.

[13] "What is eBPF? An Introduction and Deep Dive into the eBPF Technology." (), [Online]. Available: https://ebpf.io/what-is-ebpf/ (visited on 10/28/2024).

[14] M. Amaral, S. Choochotkaew, Eun Kyung Lee, H. Chen, and T. Eilam. "Exploring Kepler's potentials: Unveiling cloud application power consumption," CNCF. (), [Online]. Available: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/ (visited on 04/15/2024).

[15] "ICT Sector Guidance built on the GHG Protocol Product Life Cycle Accounting and Reporting Standard," GeSI facilitates real world solutions to real world issues within the ICT industry and the greater sustainability community. (), [Online]. Available: https://www.gesi.org/research/ict-sector-guidance-built-on-the-ghg-protocol-product-life-cycle-accounting-and-reporting-standard (visited on 08/26/2024).

[16] "Kepler Deep Dive - Kepler." (), [Online]. Available: https://sustainable-computing.io/usage/deep_dive/#how-is-the-power-consumption-attribution-done (visited on 07/07/2024).

[17] "Kepler-doc/docs/design/metrics.md at main · sustainable-computing-io/kepler-doc," GitHub. (), [Online]. Available: https://github.com/sustainable-computing-io/kepler-doc/blob/main/docs/design/metrics.md (visited on 06/24/2024).

[18] V. Gudepu, R. R. Tella, C. Centofanti, J. Santos, A. Marotta, and K. Kondepu, "Demonstrating the Energy Consumption of Radio Access Networks in Container Clouds," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, May 2024, pp. 1–3. DOI: 10.1109/NOMS59830.2024.10575134. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10575134 (visited on 08/05/2024).

[19] D. Soldani, P. Nahi, H. Bour, *et al.*, "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3281480. [Online]. Available: https://ieeexplore.ieee.org/document/10138542 (visited on 05/24/2024).

[20] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128624002032 (visited on 08/05/2024).

[21] L. Andringa, "Estimating energy consumption of Cloud-Native applications," MSc Thesis, University of Groningen, Groningen, Jul. 4, 2024, 59 pp. [Online]. Available: https://fse.studenttheses.ub.rug.nl/33583/.

[22] OpenTelemetry. "Observability primer," OpenTelemetry. (), [Online]. Available: https://opentelemetry.io/docs/concepts/observability-primer/ (visited on 11/04/2024).

[23] T. Pol, "Carbon Footprint Monitoring up to Container-Level in Virtualized Environments: A Hardware and Hypervisor-Free Approach," University of Groningen, Apr. 11, 2024, 63 pp.

[24] Kuljeet Kaur, Sahil Garg, Geogres Kaddoum, Syed Hassan Ahmed, and Mohammed Atiquzzaman. "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem | IEEE Journals & Magazine | IEEE Xplore." (May 12, 2022), [Online]. Available: https://ieeexplore-ieee-org.proxy-ub.rug.nl/document/8825476 (visited on 05/24/2024).

[25] "Resource Management for Pods and Containers." (), [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (visited on 06/23/2024).

[26] B. Lemoine. "Energy Efficiency of N:1 Protection Setups with Kubernetes Horizontal Pod Autoscaler | IEEE Conference Publication | IEEE Xplore." (2022), [Online]. Available: https://ieeexplore-ieee-org.proxy-ub.rug.nl/document/9758129 (visited on 05/24/2024).

[27] I. Fé, T. A. Nguyen, Andréc. B. Soares, *et al.*, "Model-Driven Dependability and Power Consumption Quantification of Kubernetes-Based Cloud-Fog Continuum," *IEEE Access*, vol. 11, pp. 140 826–140 852, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3340195. [Online]. Available: https://ieeexplore.ieee.org/document/10347188?denied= (visited on 06/01/2024).

[28] B. R. Stojkoska and K. Trivodaliev, "Enabling internet of things for smart homes through fog computing," in *2017 25th Telecommunication Forum (FOR)*, Nov. 2017, pp. 1–4. DOI: 10.1109/FOR.2017.8249316. [Online]. Available: https://ieeexplore.ieee.org/document/8249316 (visited on 06/01/2024).

[29] N. Huang, A. Li, S. Zhang, and Z. Zong, "Reducing Cloud Expenditures and Carbon Emissions via Virtual Machine Migration and Downsizing," in *2023 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, Nov. 2023, pp. 74–81. DOI: 10.1109/IPCCC59175.2023.10253871. [Online]. Available: https://ieeexplore.ieee.org/document/10253871 (visited on 06/09/2024).

[30] "SPECpower_ssj® 2008." (), [Online]. Available: https://www.spec.org/power_ssj2008/ (visited on 06/09/2024).

[31] "Amazon Compute Service Level Agreement," Amazon Web Services, Inc. (), [Online]. Available: https://aws.amazon.com/compute/sla/ (visited on 06/09/2024).

[32] "Google Compute Engine Service Level Agreement (SLA)," Google Cloud. (), [Online]. Available: https://cloud.google.com/compute/sla (visited on 06/09/2024).

[33] "Microsoft - Service Level Agreements (SLA) for Online Services." (), [Online]. Available: https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services?lang=1 (visited on 06/09/2024).

[34] "Deploy using Helm Chart - Kepler." (), [Online]. Available: https://sustainable-computing.io/installation/kepler-helm/ (visited on 06/23/2024).

[35] A. C. Riekstin, B. B. Rodrigues, K. K. Nguyen, *et al.*, "A Survey on Metrics and Measurement Tools for Sustainable Distributed Cloud Networks," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1244–1270, 2018, ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2784803. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8226747 (visited on 06/08/2024).

[36] V. D. Reddy, B. Setz, G. S. V. Rao, G. Gangadharan, and M. Aiello, "Metrics for sustainable data centers," *IEEE Transactions on Sustainable Computing*, vol. 2, no. 03, pp. 290–303, Jul. 2017, ISSN: 2377-3782. DOI: 10.1109/TSUSC.2017.2701883. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TSUSC.2017.2701883.

[37] J. Kosińska, B. Baliś, M. Konieczny, M. Malawski, and S. Zieliński, "Toward the Observability of Cloud-Native Applications: The Overview of the State-of-the-Art," *IEEE Access*, vol. 11, pp. 73 036–73 052, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3281860. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10141603 (visited on 06/09/2024).

[38] *Rancher/rke*, Rancher, Jun. 21, 2024. [Online]. Available: https://github.com/rancher/rke (visited on 06/23/2024).

[39] "Prometheus," CNCF. (Jun. 3, 2024), [Online]. Available: https://www.cncf.io/projects/prometheus/ (visited on 06/03/2024).

[40] M. Hansen, *Mrlhansen/idrac_exporter*, Jun. 28, 2024. [Online]. Available: https://github.com/mrlhansen/idrac_exporter (visited on 06/29/2024).

[41] "Plugwise-standalone · main · Lars Andringa / Master Thesis · GitLab," GitLab. (May 13, 2024), [Online]. Available: https://gitlab.com/l.s.andringa1/master-thesis/-/tree/main/plugwise-standalone?ref_type=heads (visited on 06/29/2024).

[42] "Power Capping Framework — The Linux Kernel documentation." (), [Online]. Available: https://www.kernel.org/doc/html/latest/power/powercap/powercap.html (visited on 07/06/2024).

[43] "Kepler reports unrealistic measurements for short period · Issue #1344 · sustainable-computing-io/kepler," GitHub. (Apr. 11, 2024), [Online]. Available: https://github.com/sustainable-computing-io/kepler/issues/1344 (visited on 06/23/2024).

[44] *Sustainable-computing-io/kepler-model-server*, Sustainable Computing, Aug. 3, 2024. [Online]. Available: https://github.com/sustainable-computing-io/kepler-model-server (visited on 08/05/2024).

[45] C. I. King, *ColinIanKing/stress-ng*, Aug. 24, 2024. [Online]. Available: https://github.com/ColinIanKing/stress-ng (visited on 08/25/2024).

[46] "Training Pipeline - Energy Source - Kepler." (), [Online]. Available: https://sustainable-computing.io/kepler_model_server/pipeline/#energy-source (visited on 09/20/2024).

[47] "Kepler-model-server/src/kepler_model/util/train_types.py at 5e69fe9547b3596ba59cd4bdd8d855e1b1cdabdd · sustainable-computing-io/kepler-model-server." (), [Online]. Available: https://github.com/sustainable-computing-io/kepler-model-server/blob/5e69fe9547b3596ba59cd4bdd8d855e1b1cdabdd/src/kepler_model/util/train_types.py#L23 (visited on 09/20/2024).

[48] "Kubernetes Metrics (v1beta1)," Kubernetes. (), [Online]. Available: https://kubernetes.io/docs/reference/external-api/metrics.v1beta1/ (visited on 10/14/2024).

[49] *Kubernetes-client/java*, Kubernetes Clients, Oct. 13, 2024. [Online]. Available: https://github.com/kubernetes-client/java (visited on 10/14/2024).

[50] "What Is Hyper-Threading?" Intel. (), [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html (visited on 10/21/2024).

A. Metrics

Table 2 shows all metrics and their explanation as they appear in Prometheus.

Table 2:

Source	**Metric**	Explanation
Plugwise	`power_consumption`	Measured power by the wallplug
iDRAC	`power_control_avg_consumed_watts`	Average consumption of power control system
	`power_control_capacity_watts`	Capacity of power control system
	`power_control_consumed_watts`	Consumption of power control system
	`power_control_interval_in_minutes`	Interval for measurements of power control system
	`power_control_max_consumed_watts`	Maximum consumption of power control system
	`power_control_min_consumed_watts`	Minimum consumption of power control system
	`power_supply_capacity_watts`	Power supply capacity
	`power_supply_efficiency_percent`	Power supply efficiency
	`power_supply_health`	Power supply health status
	`power_supply_input_voltage`	Power supply input voltage
	`power_supply_input_watts`	Power supply input
	`power_supply_output_watts`	Power supply output
	`sensors_fan_health`	Health status for fans
	`sensors_fan_speed`	Sensors reporting fan speed measurements
	`sensors_temperature`	Sensors reporting temperature measurements
	`system_bios_info`	Information about the BIOS
	`system_cpu_count`	Total number of CPUs in the system
	`system_health`	Health status of the system
	`system_indicator_led_on`	Indicator LED state of the system
	`system_machine_info`	Information about the machine
	`system_memory_size_bytes`	Total memory size of the system in bytes
	`system_power_on`	Power state of the system
RAPL	`powercap_energy_uj_total`	Total energy used by package and DRAM components

Continued on next page

Table 2: (Continued)

cAdvisor	container_blkio_device_usage_total	Blkio Device bytes usage
	container_cpu_load_average_10s	Value of container cpu load average over the last 10 seconds
	container_cpu_system_seconds_total	Cumulative system cpu time consumed in seconds
	container_cpu_usage_seconds_total	Cumulative cpu time consumed in seconds
	container_cpu_user_seconds_total	Cumulative user cpu time consumed in seconds
	...	*Filesystem and network metrics omitted for brevity*
	container_memory_cache	Number of bytes of page cache memory
	container_memory_failcnt	Number of memory usage hits limits
	container_memory_failures_total	Cumulative count of memory allocation failures
	container_memory_mapped_file	Size of memory mapped files in bytes
	container_memory_max_usage_bytes	Maximum memory usage recorded in bytes
	container_memory_rss	Size of RSS in bytes
	container_memory_swap	Container swap usage in bytes
	container_memory_usage_bytes	Current memory usage in bytes, including all memory regardless of when it was accessed
	container_memory_working_set_bytes	Current working set in bytes
	container_oom_events_total	Count of out of memory events observed for the container
	container_processes	Number of processes running inside the container
	container_spec_cpu_period	CPU period of the container
	container_spec_cpu_shares	CPU share of the container
	container_spec_memory_limit_bytes	Memory limit for the container
	container_spec_memory_reservation_limit_bytes	Memory reservation limit for the container
	container_spec_memory_swap_limit_bytes	Memory swap limit for the container
	container_start_time_seconds	Start time of the container since unix epoch in seconds

Continued on next page

Table 2: (Continued)

	container_threads	Number of threads running inside the container
	container_threads_max	Maximum number of threads allowed inside the container, infinity if value is zero
	container_ulimits_soft	Soft ulimit values for the container root process. Unlimited if -1, except priority and nice
	container_cpu_cfs_periods_total	Number of elapsed enforcement period intervals
	container_cpu_cfs_throttled_periods_total	Number of throttled period intervals
	container_cpu_cfs_throttled_seconds_total	Total time duration the container has been throttled
	container_spec_cpu_quota	CPU quota of the container
	machine_cpu_cores	Number of logical CPU cores
	machine_cpu_physical_cores	Number of physical CPU cores
	machine_cpu_sockets	Number of CPU sockets
	machine_dimm_capacity_bytes	Total RAM DIMM capacity
	machine_dimm_count	Number of RAM DIMM
	machine_memory_bytes	Amount of memory installed on the machine
	machine_nvm_avg_power_budget_watts	NVM power budget
	machine_nvm_capacity	NVM capacity value labeled by NVM mode
Kepler	container_bpf_block_irq_total	Aggregated block irq value obtained from BPF
	container_bpf_cpu_time_us_total	Aggregated CPU time obtained from BPF
	container_bpf_net_rx_irq_total	Aggregated network rx irq value obtained from BPF
	container_bpf_net_tx_irq_total	Aggregated network tx irq value obtained from BPF
	container_bpf_page_cache_hit_total	Aggregated Page Cache Hit obtained from BPF
	container_cache_miss_total	Aggregated cache miss value
	container_core_joules_total	Aggregated RAPL value in core in joules
	container_cpu_cycles_total	Aggregated CPU cycle value
	container_cpu_instructions_total	Aggregated CPU instruction value

Continued on next page

Table 2: (Continued)

	container_dram_joules_total	Aggregated RAPL value in dram in joules
	container_joules_total	Aggregated RAPL Package + Uncore + DRAM + GPU + other host components (platform - package - dram) in joules
	container_other_joules_total	Aggregated value in other host components (platform - package - dram) in joules
	container_package_joules_total	Aggregated RAPL value in package (socket) in joules
	container_uncore_joules_total	Aggregated RAPL value in uncore in joules
	node_core_joules_total	Aggregated RAPL value in core in joules
	node_cpu_scaling_frequency_hertz	Current average cpu frequency in hertz
	node_dram_joules_total	Aggregated RAPL value in dram in joules
	node_other_joules_total	Aggregated RAPL value in other components (platform - package - dram) in joules
	node_package_joules_total	Aggregated RAPL value in package (socket) in joules
	node_platform_joules_total	Aggregated platform (entire node) in joules
	node_uncore_joules_total	Aggregated RAPL value in uncore in joules
KubeWatt	kubewatt_container_power_watts	Power attributed per container. Labelled by namespace, node, container name, pod name and type.