Vrije Universiteit Amsterdam

Master Thesis

Real-time Scaphandre Energy Metrics Pipeline Integrated with Escheduler

Author: Tim van Kemenade (2753428)

1st reader: Daniele Bonetta
2nd reader: Tiziano De Matteis
daily supervisor: Matthijs Jansen

A thesis submitted in fulfillment of the requirements for the VU Master of Science degree in Computer Science

August 19, 2024

Abstract

Energy usage of digital infrastructure is high and only expected to increase in the Netherlands as digital infrastructure expands. Data centers contribute a lot to this energy usage and will contribute even more as more applications will become cloud-native. Placing the applications on the cloud based on their energy usage could lead to a drastic reduction in energy usage. In this thesis we will answer the main question: **How to integrate real-time energy metrics with a Kubernetes scheduler running in a VM?**

Tools like Scaphandre exist to measure real-time energy usage on a per-pod level. However, Scaphandre has not been integrated with schedulers. In this work we integrate Scaphandre with a Kubernetes scheduler, Escheduler, to schedule based on real energy usage. We do this by creating a real-time energy metric pipeline that exposes Scaphandre metrics from the host in the guest VMs running a Kubernetes cluster. Escheduler uses the energy metric pipeline to gather metrics of pods and nodes to make a scheduling decision.

We find that the real-time energy metric pipeline has little overhead and makes new insights into pod level energy metrics available. Scaphandre does suffer when a lot of nodes exists on a single host and has trouble representing peaks in usage in the metrics, but overall is reasonably accurate.

We compared Escheduler to the default Kubernetes scheduler, kube-scheduler, and found that it makes the same scheduling decisions to balance the load, but does so only using the energy metrics.

We made all experiments reproducible by providing automatic infrastructure deployment and setup as well as provide a load to compare them under with measurements of energy and CPU usage built-in.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Acronyms

RAPL Running Average Power Limit. 2, 3, 10, 14, 15, 18, 24, 29, 40

VM Virtual Machine. iii, 2–7, 9–12, 14, 15, 17–19, 21, 24–30, 32–34, 37, 39, 40

Glossary

Continuum is a deployment and benchmarking framework for the edge-cloud compute continuum (1). Used for VM creation and benchmarking setup. 8, 9, 15, 17, 18, 21, 24, 33, 37, 40, 49–51

DeathStarBench is a open-source benchmark suite for cloud microservices (2) of which we will focus on the social network end-to-end service. 9, 10, 20, 23–27, 31, 33, 40, 49

Grafana is the open source analytics & monitoring solution to visualize Prometheus metrics. 9, 15, 18

Kubernetes also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. iii, 3–9, 11, 12, 14, 15, 18–20, 23, 24, 26–31, 33, 34, 37, 39, 40, 49

KVM (Kernel-based Virtual Machine) is an open source virtualization technology built into Linux that allows the kernel to function as a hypervisor. 9

libvirt is an open-source API, daemon and management tool for managing platform virtualization. Used to manage QEMU/KVM. 9

Prometheus is an open-source monitoring system. 9, 15, 18–20, 24, 27, 33, 35, 39–41, 49

QEMU is an open-source full system emulation. iii, 9, 15, 18, 19, 26–29, 31, 37, 51

Scaphandre is a metrology agent dedicated to electric power and energy consumption metrics (3). Used to propagate energy metrics to a VM and Prometheus. i, iii, 3, 7, 9–11, 14, 15, 17–19, 21, 23–29, 31–34, 37, 39, 40, 49–51

virtiofsd is a shared file system that lets virtual machines access a directory tree on the host (4). 9–11, 15, 17–19, 24, 27, 37, 39, 49

1

Introduction

The growth of cloud data centers in the Netherlands has been high with a record of new supply coming online in 2021 (5). These data centers rely largely on green energy with a current estimated energy share of 90% which is estimated to grow to >99% by 2027 (5). The energy footprint of the digital infrastructure in the Netherlands is estimated at 0.65% (18 Petajoule) of the Dutch energy usage (6). The share of energy usage from the digital infrastructure is only expected to grow due to increasing digitalization.

The Dutch Manifesto on Future Computer Systems and Networking Research in the Netherlands also emphasizes the importance of the data centers (7). These data centers and related ICT infrastructure enable 3.3 million jobs and over 60% of the Dutch GDP. Cloud adaptation already exceeds 90% for economic organizations and 65% for government and public education organizations. This reliance of other sectors on the digital sector in the Dutch economy is also by the government making a push towards becoming carbon neutral even more reliant on a transition from the digital sector to the use of more green energy and improving the energy efficiency of existing (and future) services (8).

Continued digitalization is expected to increase the number of jobs and importance to the economy with the Manifesto highlighting the need for dynamic workloads and resource management for the new societal needs (7). With dynamic workloads and other resource management goals also comes the need for more detailed insights. More detailed insights are covered by the research topic on the rethinking of telemetry, monitoring, and real-time observation (7). Measuring energy in real-time would contribute to both of these goals and align with the societal pressure to become greener and increase energy efficiency as the amount of connected devices is expected to grow with new applications like the edge.

The observability of real-time energy metrics can be explored in multiple ways to achieve higher energy efficiency. Monitoring is a straightforward approach to create greater aware-

ness of the energy footprint that running an application has (9). Monitoring of energy usage is already done on larger scales such as individual buildings, or even servers having accurate total usage, however, the total energy usage doesn't correlate to the energy usage of an individual application. With increased granularity, we can paint a better picture of what is using energy and potentially wasting energy. Adjusting based on the energy metrics can be done by individual applications, but we should also be able to better place the applications to save energy. The energy usage of an application can differ depending on the placement as nodes can be more energy efficient, a more distributed load can result in less idle energy waste (10), and communication between nodes to run an application incurs energy cost which is even amplified by data transfer in preparation to running an application. The scheduler determines the placement of an application in the cloud. The scheduler is responsible for managing the overall cloud's resources and scheduling decisions can have a big impact on overall energy consumption (11).

In a perfect world, the scheduler has complete information about the state of the cluster, limits of any node in the cluster, and accurate information on how a job behaves in terms of resource consumption and completion time. However, the availability of energy metrics is hard due to the encapsulated nature of a virtualized environment (12). Measuring the cost and latency associated with real-time energy usage measurements would allow it to be more adaptable. However, including the real-time energy usage in the scheduler could prove beneficial in reducing energy usage and a logical choice if energy efficiency is set as a goal. Scheduling based on energy consumption is usually done via an energy model. An energy model can be trained on real energy data (13) and/or estimate energy usage based on CPU statistics collected at runtime. The energy models however are only an approximation of the energy consumption (14). The real energy consumption of a machine can be measured using an external power meter. The power meter is however hard to integrate with such a system which makes it very use case specific and not broadly applicable. We need an approach that can measure energy in real time with acceptable accuracy, which can be achieved using power measurements integrated into the CPU. One such CPU energy measurement is called Running Average Power Limit (RAPL). RAPL reports CPU (including integrated graphics), memory, and PCI bus energy usage.

RAPL is integrated into the CPU. However, cloud instances are usually encapsulated in a VM to achieve resource separation (15). A VM might still be directly pinned to a core, but not to its associated energy metrics. Model-based approaches do not suffer from limited availability as they don't require read access to data on the host, but these approaches lack accuracy. The host also doesn't know what exactly happens in the VM making it harder to

make decisions about guest VMs from the host (16). However, sharing the energy metrics at the host to the guest VMs would allow for per-process energy reporting and scheduling that is energy aware.

This paper will focus on providing real-time energy metrics from the host to the guest VMs in which a Kubernetes scheduler can then use the energy metrics to decide on job scheduling. In doing so, we establish a framework in which different schedulers can be tested for their energy usage, CPU usage, and end-to-end latency. The energy usage will be available per process running inside the VM resulting in higher granularity for scheduling decisions.

1.1 Context

In recent years energy consumption has gained more attention as the world tries to reduce its carbon footprint while simultaneously using more energy with digital infrastructure only expected to expand. Data centers are a huge driver of energy consumption (9) and this will remain an issue as more applications transition to a cloud-native approach, being hosted on data center infrastructure. The huge amount of applications hosted in a central network provides the opportunity to reduce energy consumption by performing smarter placement. A scheduler performs the placement of applications yet the scheduler is often unaware of the real energy consumption its decisions have. We want to enable real-time energy metrics on a Kubernetes cluster running on VMs in a data center (17). Data centers manage the clusters using a tool such as Kubernetes like the Google Cloud Provider (GCP) with the Google Kubernetes Engine (GKE). This thesis works towards a pipeline providing real-time energy metrics to a Kubernetes scheduler. Real-time energy metrics in the Kubernetes cluster allow more fine-grained information on energy usage, like computing energy usage for every running pod. From the Kubernetes cluster we can provide a base for further understanding and future research on the impact scheduling has on energy consumption.

A real-time energy metrics pipeline needs real energy statistics to attribute to different processes. Recent advancements that enable the measurement of real-time energy metrics are the introduction of Scaphandre, a RAPL based energy monitoring tool. Scaphandre can measure real-time energy usage on a per-process basis. Exposing these metrics to a Kubernetes cluster would allow us to attribute energy usage to individual pods. The metrics also need to be transferred from the host to the guest VMs and need to be recomputed after to assign the energy metrics of the VM process to individual pods in the Kubernetes cluster.

With the real-time energy metrics pipeline in place, we can use energy usage to schedule pods in Kubernetes. Such a scheduler would have to access the energy metrics, map the energy metrics to existing pods, store pod and node energy metrics, and schedule said pods.

The real-time energy-based scheduler allows for scheduling control based on real-time energy usage. But how impactful it is needs to be measured. Current and future schedulers need to be compared in terms of energy usage. To facilitate comparing schedulers this thesis focuses on reproducible results. The results should be observable both during measuring and when not actively monitoring to allow greater insight into the behaviour of different schedulers in terms of energy usage.

Making the energy consumption inside a VM of different tasks observable would allow a greater sense of responsibility as well as insight to reduce the energy consumption as energy consumption patterns can be better recognized. To achieve this we need to know the actual energy consumption and act according to these energy metrics. The energy metrics need to be available per process in a VM to know how much energy a specific task is used in a data center that uses VMs to encapsulate resources.

1.2 Problem Statement

Minimizing energy consumption is a trending goal for schedulers in the cloud. The energy consumption is usually explored in a containerized setup on bare-metal machines (18), even though *cloud providers partition bare-metal machines using VMs which then run the containerized environment.* The containerized environment living in a VM brings problems with it as it prevents direct access to physical hardware to the extent possible on a bare-metal machine. Energy metrics are hard to access in a VM environment. Propagating these metrics to the VM is important for multiple reasons as seen in the Problem Statement (PS):

PS1 Individual real-time energy usage of Kubernetes applications is obscure: There are no direct indicators of energy used by an application running in Kubernetes. To calculate the energy used by an individual pod one must know the total CPU time used by the system, the CPU time used by the pod, and the total energy usage of the system. The data obscurity makes it hard for developers and corporations to observe the energy usage of an application running in Kubernetes. This is even further complicated by the VM encapsulation as it obscures the necessary information

further. Addressing the problem would provide accountability for inefficient energy-consuming patterns in applications. As CPU, RAM, and storage resources become cheaper it could get even more beneficial to optimize for energy usage.

PS2 Accuracy and frequency of real-time energy metrics is unknown: The tools exist but they are untested. If we want to rely on energy usage in the future we have to make sure that the accuracy and frequency are known. We want to solve **PS1** which would expose energy metrics. But we also need to verify the accuracy and frequency of reporting. Doing this would allow for better reasoning about the metrics and could prove beneficial for future implementations of schedulers.

PS3 Kubernetes scheduling decisions do not take real-time energy usage into account: Current energy-aware solutions rely on energy models to estimate the energy used. The energy model has incomplete information and its performance is hard to verify as real-time energy metrics are not readily available. Making scheduling decisions based on the real-time energy metrics would allow for local optimization for tasks that need to be scheduled right away. Taking real-time energy usage into account would also allow short-term energy usage predictions to better balance the energy capacity. Combining deferred scheduling with real-time energy usage-aware scheduling enables optimal energy-saving pod placement.

PS4 Energy based comparison of schedulers is hard to reproduce: As schedulers advance it becomes more important to compare the schedulers in a reproducible manner to draw conclusions as to what scheduler performs better under which load. This is hard due to differing experimental setup. A single setup that can be performed automatically including energy measurements and the pipeline setup with a configurable load and scheduler setup would allow easier comparison between different schedulers and enable more insights into energy usage patterns of schedulers.

1.3 Research Questions

The main question (MQ) we try to answer is: **How to integrate real-time energy metrics with a Kubernetes scheduler running in a VM?** We answer the MQ using 4 Research Questions (RQ) and explain how they are connected to the problem statement:

RQ1 How to measure energy usage inside a VM, per pod running inside the Kubernetes node? (**PS1**)

RQ1.1 What are the approaches to measuring energy usage?

RQ1.2 What is the overhead of measuring energy usage?

Energy readings need to be processed to get the energy metrics on the host and these need to be transferred to the guest VMs to then divide the VMs energy footprint over the pods using the energy inside the Kubernetes node. We need to know how how much overhead is incurred for measuring this energy usage and propagating it to the VM.

RQ2 What is the accuracy and frequency of the real-time energy metrics? (**PS2**)

RQ2.1 How accurate are the energy metrics?

RQ2.2 How often is the real-time energy metrics reported?

We need to know how accurate these real-time energy readings are and how frequently the metrics are reported. The accuracy and frequency being known would allow us to better reason about the real-time energy metrics.

RQ3 How can we design a Kubernetes scheduler that optimizes energy usage? (**PS3**)

RQ3.1 What components are a Kubernetes scheduler composed of?

RQ3.2 What limitations does the VM encapsulation impose on the Kubernetes scheduler?

RQ3.3 How to make energy usage available to the scheduler?

Making our own Kubernetes scheduler that makes decision based on real-time energy metrics requires us to dissect a Kubernetes scheduler and design our own scheduler. One step is knowing which components we need for a Kubernetes scheduler to know where we can place the energy metrics processing to include it in our scheduling decision process. The custom scheduler in this thesis runs in an encapsulated VM environment for which we need to know the differences between running on bare-metal to generalize the findings. We also need to now in what form the Kubernetes scheduler should receive the energy metrics to use current (and past) energy metrics to estimate the energy use of tasks that have yet to be scheduled.

RQ4 What is the energy-performance trade-off between different Kubernetes schedulers? (**PS4**)

RQ4.1 Which Kubernetes schedulers are there?

RQ4.2 How to compare different Kubernetes schedulers against each other?

RQ4.3 How to evaluate the energy-performance trade-off?

There are many Kubernetes schedulers implemented. It is trivial to test the default Kubernetes scheduler, but we also need to make a choice which we can reason about for choosing the other Kubernetes scheduler. This requires a look into what Kubernetes scheduler implementations exist. We also need to know how to get energy usage statistics on a Kubernetes scheduler and how a potential Kubernetes scheduler could use energy metrics to compare it to a custom Kubernetes scheduler that uses the real-time energy metrics. Last we need to design experiments to give a fair comparison of the energy-performance trade-off of the different Kubernetes schedulers. And we would like to do all of this in a reproducible manner to enhance future testability of Kubernetes schedulers.

1.4 Research Methodology

We make use of the following methods (M) in this thesis:

M1 Quantitative research (19, 20) via benchmarks that single out the impact of individual components to highlight potential shortcomings to answer **RQ4.2**;

M2 Design, abstraction, prototyping (21, 22, 23) to answer **RQ3.2**, **RQ3.3**;

M3 Experimental research (24, 25, 26) via designing appropriate workload-level benchmarks, and a performance comparison of different schedulers to answer **RQ4.2** and **RQ4.3**;

M4 Open-science (27, 28, 29, 30) via publicly available tools and reproducible experiments to answer **RQ1.1**, **RQ3.1**.

1.5 Thesis Contributions

The requirements lead to the thesis Contributions (C):

C1 We measure real-time energy usage and expose it in every VM for every Kubernetes pod using our energy metric pipeline (**RQ1**).

C2 We quantified the accuracy of Scaphandre and singled out the overhead of individual components (**RQ1**) while enabling the direct comparison of schedulers (**RQ2**).

C3 We implemented Escheduler, a Kubernetes scheduler optimized for energy consumption that uses real-time energy metrics (**RQ3**).

C4 We made the comparison of schedulers reproducible via automated infrastructure deployment and benchmarking of individual components (**RQ1**) and schedulers (**RQ4**).

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

To understand more about plagiarism policy at VU Amsterdam, see `https://vu.nl/en/about-vu/more-about/academic-integrity`.

1.7 Thesis Structure

1. (CURRENT SECTION) Introduction with contributions, methodology, and thesis structure on energy usage in Continuum of various Kubernetes scheduling algorithms.
2. Background.
3. Design details on power measurement setup, how it propagates to Kubernetes, and requirements of the scheduler, Escheduler.
4. Implementation of the infrastructure and Escheduler with a decision model based on expected energy usage.
5. Evaluation of different Kubernetes schedulers and supporting baseline measurements.
6. Related Work
7. Lessons Learned
8. Conclusion explicit answer to research questions

2

Background

This thesis will built on top of previous work. We specifically use Continuum for the setup of a Kubernetes cluster inside multiple VMs, Scaphandre for the energy measurements (alternatives are also mentioned), virtiofsd for file sharing between the host and guest VMs, and DeathStarBench with wrk2 to perform the load of the benchmark. We will give some background on these tools in this section.

2.1 Continuum

This thesis makes extensive use of Continuum (1) to set up the cloud infrastructure. Everything that requires a restart of the VM is done by Continuum. This thesis uses a slightly adjusted version that mounts the shared energy metrics in the right location and uses memory backing. The VMs started by Continuum are QEMU KVM VMs managed by Libvirt. Other things handled by Continuum is the VM domain file including configurable settings, networking, predefined VM names, setup of Kubernetes with metrics server, setup of Prometheus, setup up of Grafana, and access inside the VM with the host's Docker registry. The use of a local registry is recommended to prevent hitting the pull limit of images. The OS used by a Continuum VM is Ubuntu 20.04 and Continuum has also been tested on Ubuntu 22.04.

2.2 Power usage measurement tools

This thesis wants to make use of Scaphandre (3) due to being a well-documented tool. Alternatives include PowerAPI (31) and Kepler (32). A comparison of Scaphandre, PowerAPI, and Kepler is not made in this thesis as an extensive comparison study already

exists (33). Scaphandre computes energy metrics based on Running Average Power Limit (RAPL). RAPL is not available by default due to security concerns (34) requiring the need for more isolation by preprocessing on the host.

Another option would have been an external watt-hour meter but this was not considered due to it being less accessible.

2.3 File host/guest sharing

Real-time energy metrics are measured on the host and need to be transferred to the guest VM. We make extensive use of virtiofsd (4) to mount the shared metric file into the guest VM.

2.4 Benchmark(ing)

We use DeathStarBench (35) which can be found on GitHub (2) to test the schedulers. We specifically use the social network from DeathStarBench and generate a HTTP load on it using the adjusted version of wrk2 also on DeathStarBench. A further explanation as to how DeathStarBench is used can be found in the implementation section.

3

Design of Escheduler: A Power-based Kubernetes Scheduler

In this section we answer **RQ1** and **RQ3** about setting up the energy metric pipeline and using this in our scheduler Escheduler. We will set up a pipeline to feed real-time energy metrics to the scheduler. The scheduler, Escheduler, will utilize the real-time energy metrics to place pods to save energy over the lifetime of the Kubernetes cluster. To identify the different components, we must first understand the system we are working with. The system consists of multiple VM on a single host running a Kubernetes cluster. The system is further discussed in Section 3.1. Measuring the real-time energy usage is done on the host using Scaphandre and put into the corresponding VM using virtiofsd and a custom exporter as discussed in Section 3.3. The exposed energy metrics enable us to make a scheduler Escheduler that can place pods based on the available readings. Our design of Escheduler will be discussed in Section 3.2. The last problem we are left with is that Escheduler only uses real-time data, and on startup doesn't have any data yet when it gets asked to schedule pods. The lack of startup data will be discussed in Section 3.4.

3.1 System of Interest

The system this paper focuses on will be a single host running VMs with each VM being a separate node in a Kubernetes cluster. The Kubernetes cluster will have one controller node and multiple worker nodes.

Each VM has isolated hardware resources allocated from the host. Each VM should also be capable to run in any Kubernetes cluster that has the prerequisites to have real-time energy readings. The need for real-time energy readings in all nodes results in needing

real-time energy readings for each individual VM and the VM also being the owner of the real-time energy readings. We need to expose the metrics gathered on the host to each individual VM and expose the metrics in each Kubernetes node.

The Kubernetes cluster must act on the real-time energy readings in the entire system. We need to share real-time energy readings to achieve global information about energy usage. The energy usage needs to be known to the scheduler. The Kubernetes cluster has a single controller responsible for running our scheduler Escheduler. So all the energy usage information must be known to the controller node for Escheduler to make a decision, but not every node needs to have all this information.

3.2 Escheduler

We first need to establish design requirements for our scheduler Escheduler. We will list both Functional Requirements (FR) and Non-Functional Requirements (NFR).

FR1 Escheduler must be integrated with real-time energy metrics from Scaphandre.

The goal of Escheduler is to optimize based on real-time energy metrics. Scaphandre is the tool used to measure the real-time energy based on RAPL readings. Escheduler has access to the energy usage of pods internally and must use the energy usage information.

FR2 Escheduler past energy metrics for scheduling new pods.

Escheduler must use previously gathered metrics to estimate the energy requirements of a pod. Escheduler should periodically collect energy metrics from the pods and store the energy metrics for later use.

FR3 A pod must be scheduled if a node can accommodate the pod.

Pods shouldn't be deferred to a later time to lower energy usage. Any pod that gets scheduled has the requirement to be scheduled right away. We design a scheduler that decides on placement in the current state and force Escheduler to make that decision if placement on any node is possible.

NFR1 Run on a Kubernetes cluster running on VMs.

The system of interest as seen in Section 3.1 is a Kubernetes cluster running on VMs. We must enable Escheduler to get access to sufficient data in the virtualized system.

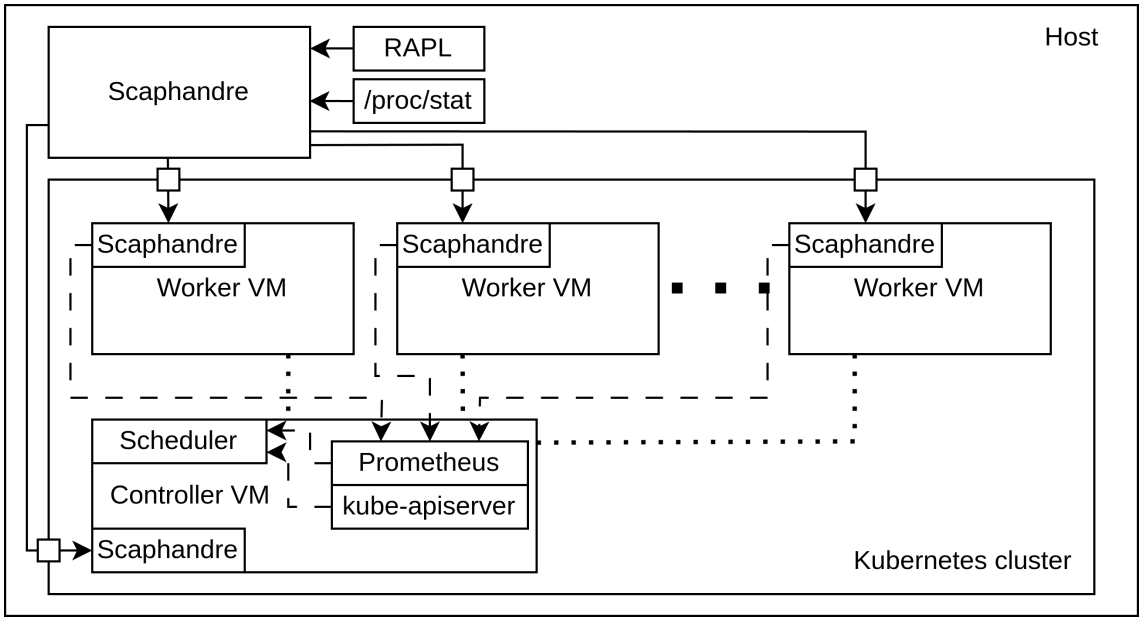

Figure 3.1: Dataflow within the host and Kubernetes cluster.

NFR2 Pod energy metrics must be observable from the outside via the Prometheus API.

We must provide a pod's energy usage from outside the Kubernetes cluster. Observability of the energy use of pods provides data on how to improve the pod and placement of the pods. The observability helps in gathering the energy impact parts of the digital infrastructure have.

NFR3 Escheduler can only use metrics available internally.

The real-time energy metrics are calculated on the host. We must make the energy metrics available to the guest VMs by propagating the real-time energy metrics from the host to the guest and share it with the cluster. Escheduler should rely fully on internal metrics once the energy metrics are contained within the cluster.

These goals are established to get real-time power usage metrics in the Kubernetes scheduler while the impact can be observed for added incentives and accountability. Another important aspect is the ability to react to new and recurring tasks. The same task can happen often due to restarts and scaling. Each of these instances is a chance to get a more accurate estimation of the power usage of the schedulable pod as we have previous evidence of its power usage patterns. Other pods also provide energy usage data that can be generalized to estimate a new pod's energy usage.

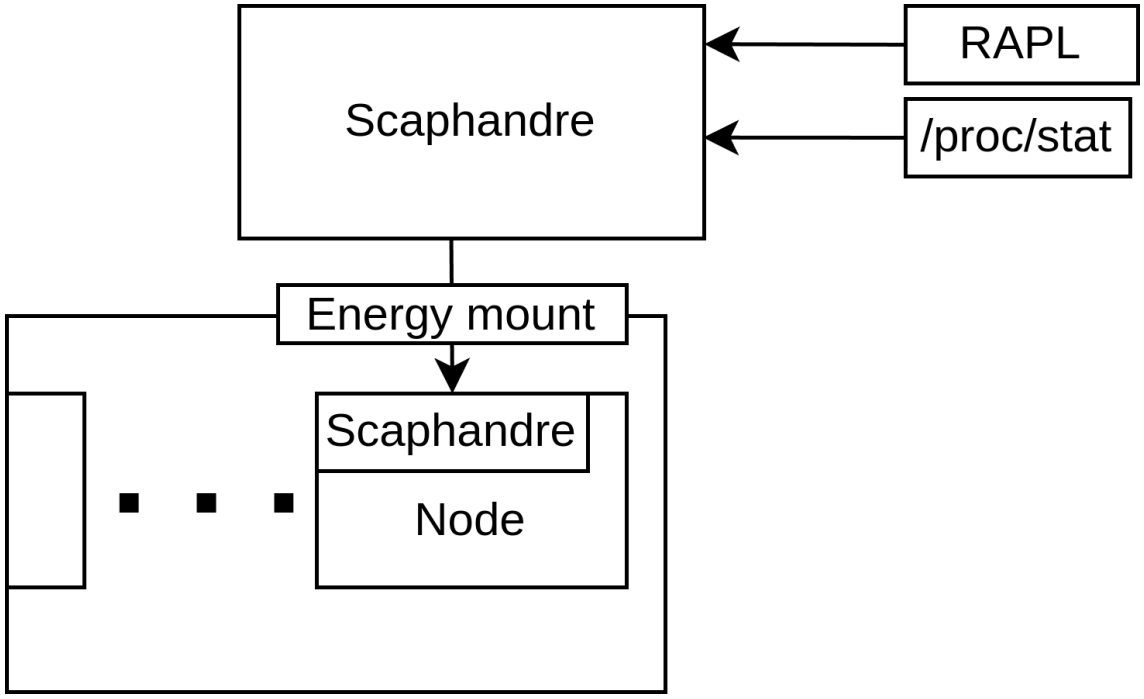

Figure 3.2: Dataflow of the energy metrics pipeline from the host to the guest VM.

3.3 Exposing Energy Metrics

We want to make real-time energy readings available to Escheduler **FR1**. We need to achieve the metric integration while running on a Kubernetes cluster running on VMs **NFR1**. We also need to observe the metrics from the outside **NFR2**. To achieve the aforementioned requirements we need to compute energy metrics on the host and propagate the energy metrics to the VM and into the Kubernetes cluster. We first need to get real-time energy readings. The source of the energy readings will be Running Average Power Limit (RAPL), a hardware feature acting as an energy consumption monitor across different domains of the CPU chip, attached DRAM, and on-chip GPU that has been available since the Intel Haswell architecture. RAPL is more readily available than a physical power meter as it is directly built into modern systems.

There are existing libraries to process the RAPL metrics with mechanisms to transfer the energy readings to a VM. Scaphandre, PowerAPI, and Kepler all work directly on RAPL readings and have the capability to expose the readings to a VM. We used Scaphandre instead of PowerAPI as it had more documentation to work on top for the use case of a Kubernetes cluster running on VMs. When we started the thesis Kepler used a software-based model to map these readings to a VM, a software-based model has potentially lower

overhead and similar accuracy due to the uncertainty in scheduling but was out of scope for this thesis. Scaphandre has support for QEMU and Prometheus built in and works using a shared file system recommending the use of virtiofsd.

Exposing the energy metrics requires many components to work together that would not be needed in common setups. The flow of data as seen in Figure 3.1 starts with the Running Average Power Limit (RAPL) metrics and /proc/stat information. RAPL and /proc/stat get processed by Scaphandre on the host, where the total energy used from RAPL gets divided over all processes based on /proc/stat. Scaphandre stores energy metrics per QEMU VM in a separate folder. Every energy metric folder is mounted to the corresponding QEMU VM using virtiofsd. The mounted energy metric file is read by another instance of Scaphandre that is running inside the Kubernetes cluster. All VMs send the metrics Scaphandre calculated per pod to Prometheus. Escheduler can access the energy metrics from the Prometheus API.

With the energy metrics available in the cluster we only need to make the per-pod metrics available outside of the cluster. We achieve the observability by exposing the Prometheus API to the outside and include Grafana for better visualization which is also loaded by Continuum when observability is turned on. An overview of the metric propagation can be seen in Figure 3.2.

3.4 Scheduling flow

Escheduler must use past energy metrics to schedule new pods **FR2**. There are two options when gathering past data with our use of Prometheus. Option 1 is using time range queries to request information on past events and option 2 is periodically retrieving current pod energy metrics. Option 1 has fewer requests to Prometheus but the range query is only useful for a range where pods were present on the cluster, which would be sufficient on a busy cluster but if the time range does not contain data you need to either look further back or schedule without information. Option 2 although it sends more requests has greater control over what is stored.

We schedule the pod if more than 1 node is available **FR3**. We schedule on the pod with the most energy capacity left if there are multiple nodes available.

With only using metrics available internally **NFR3** we suffer from a lack of data on startup as there are little metrics to use. One major caveat with scheduling in Kubernetes is that once we place a pod we cannot replace that pod unless preemption is enabled. We have to wait for either the pod lifecycle to end and reschedule or wait for additional scaling

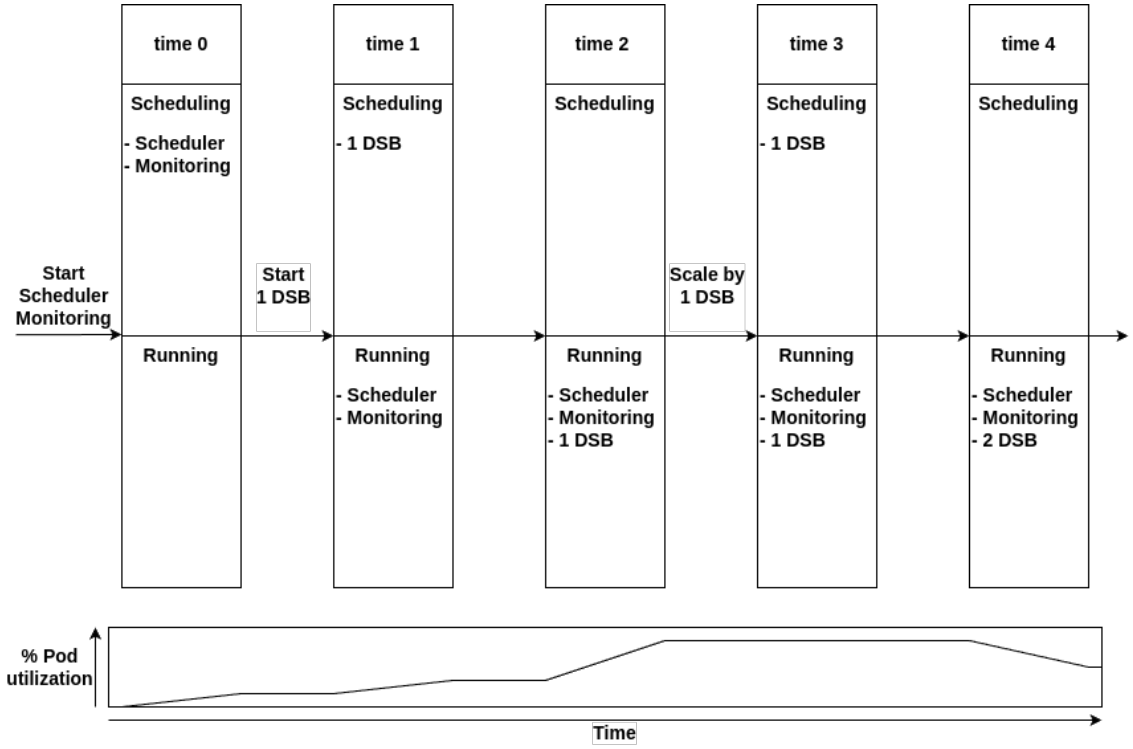

Figure 3.3: Cluster scheduling lifecycle.

that starts up new pods. The small amount of past information becomes problematic when the cluster is just initialized as we have no previous real-time energy data to place pods in an optimal way to reduce power usage. So as seen in Figure 3.3 the scheduler doesn't have past energy data to make a scheduling decision when the deployment DSB gets queued to be scheduled. Only on later scheduling events does Escheduler have past energy metrics available to reason make a measured estimate of the pod's energy use.

4

Implementation

Our goal as per the Research Questions (1.3) consists of 3 parts: measuring energy **RQ1**, introducing a Kubernetes scheduler that optimizes based on energy usage **RQ3**, and evaluating the scheduler based on energy **RQ4**. The Research Questions (1.3) result into 5 implementation parts as discussed in the Design section (3).

1. Setup the VMs
2. Use Scaphandre and expose metrics to the VMs Prometheus
3. Run DeathStarBench and configure to stress scheduler performance
4. Implement the scheduler: Escheduler
5. Compare kube-scheduler to Escheduler

4.1 VM Setup

We first have to prepare the infrastructure that we want to run (**NFR1**). We need to define what Scaphandre will output on the host and propagate the output to the VM. Scaphandre recommends the VM to use memoryBacking and requires a form of file sharing between the host and VM for which Scaphandre recommends virtiofsd.

The framework to start every VM is Continuum. We have to modify Continuum to include the memoryBacking and virtiofsd. We extend the domain generation of Continuum to include memoryBacking and virtiofsd. virtiofsd needs to connect the output directory (/var/lib/libvirt/scaphandre/) of Scaphandre to the mount point in the VM. Continuum needs to know the state of the current VMs which it does via inventory_vms. We include the virtiofsd setting in inventory_vms to ensure proper regeneration pf the VM. We need

to be able to store the metrics to later consume them in the scheduler Escheduler. We use Prometheus to store the metrics which we enable in Continuum with the observability setting. Continuum also includes Prometheus and Grafana with observability on meeting the need for observability outside of the cluster (**NFR2**). We expose Prometheus and Grafana using port-forwarding.

4.2 Scaphandre Integration

We chose Scaphandre as the power usage measurement tool, to provide real-time energy metrics and need to integrate Scaphandre with Escheduler (**FR1**). By using Prometheus as a time series database we can integrate Escheduler with Scaphandre if the Scaphandre metrics can be exposed to Escheduler via Prometheus. The pipeline consists of a host Scaphandre instance that processes RAPL and updates the energy usage for every QEMU VM, a mount using virtiofsd, and a guest Scaphandre instance running in Kubernetes to compute the energy usage per pod and share it via Prometheus. The propagation from host to the guest VM into the Kubernetes cluster to be exposed via Prometheus completes the pipeline (**C1**).

4.2.1 Host Scaphandre Instance

We need to start the pipeline at the host starting with the RAPL energy readings being processed by Scaphandre. Scaphandre has built-in support for QEMU which can be unlocked by using the QEMU feature flag during building (cargo build –release –features qemu). The QEMU mode uses /proc/stat to attribute the energy usage of a VM and stores the attributed energy usage in /var/lib/libvirt/scaphandre/{domain_name} as a counter from the start of measurement in Joules.

The current stable version of Scaphandre is v1.0.0 which we use in this Thesis, but the initial version used was v0.5.0 did not have stable metrics propagation and suffered from a bug during type conversion causing the metrics to go to 0. We initially adjusted Scaphandre v0.5.0 to resolve the bug and have stable propagation, but these issues were later solved in version v1.0.0 which became available in February.

4.2.2 Metrics Available in VM via virtiofsd

We stored the energy counter in /var/lib/libvirt/scaphandre/{domain_name}, which is the source directory we specified earlier in the VM configuration as well as the default location used by Scaphandre. The shared Scaphandre files are shared under the mount

point which we have to mount in /var/scaphandre. The metric location on the VM is the exact path that will later be consumed by the Kubernetes Scaphandre instance. The virtiofsd mounting is done for every node.

4.2.3 Guest Scaphandre in Kubernetes

We use another Scaphandre instance, this time running inside Kubernetes on every node. The Scaphandre instance for every guest computes the local energy usage per pod. We run the Kubernetes Scaphandre instance with the QEMU flag as well together with the vm flag to ensure is aware of the location and type of data it will consume. The installation is done using helm with the serviceMonitor enabled inside the monitoring namespace. The service monitor is important to register the metrics with Prometheus and we run the service monitor in the same namespace as Prometheus to ensure it gets registered. We meet **FR1** as Prometheus allows us to observe the metrics from the outside.

4.3 Implement Escheduler

We implemented Escheduler inside Go using kelseyhightower/scheduler's implementation as a base. We extended the default kube-apiserver requests to use HTTPS using the default certificate and token as HTTPS is required in our cluster. We use kube-apiserver to get metadata on pods and nodes. We can use the metadata to get a list of pods that need to be scheduled. We built energy metrics collection into the Escheduler via a request to the Prometheus API. The energy metrics are used to estimate the pod's usage based on previous metrics gathered on the pod as well as using the node energy usage to balance the energy consumption across the cluster. Scaphandre only gives us the pod name and instance IP. We still have to map the energy usage to the correct node to compute the node energy usage. We use kube-apiserver pod metadata to achieve this by looking at which node a pod is scheduled by node name. Every instance IP is mapped to a node name with the mapping. The map is then also used as a lookup to find available nodes. We gather metrics every second to ensure that we have recent information on the node. We look at pod metrics if the node energy changed and for these pod metrics we average the energy usage for all currently running pods (combining pods with same name) over the last measurements using 50% weight for the new measurement and 50% weight for the old measurement. This information is gathered at regular interval so that the weight of the measurement is similar. We also gather the energy usage of nodes at a scheduling event. Once a node is selected to be scheduled on we assign a temporary energy value to it.

The temporary energy value gets assigned to the node on which the pod is scheduled and removed once the pod is running. To remove this overhead we look at pods energy usage for which we assigned the additional buffer on every scheduling event. A pod is scheduled on the node with the most energy capacity left as we defined and is called Expected Energy Capacity (ExEC). We meet **FR2** as we schedule all tasks depending on the current energy usage of the node and previous energy usage information of pods. We always schedule a pod if a node is available satisfying FR3. No external metrics are used, we use kube-apiserver and Prometheus all running on the cluster as per NFR3. With this we finished Escheduler (**C3**).

Kubernetes provides a plugin functionality where only certain stages of the default Kubernetes scheduler are replaced. We did not use this approach as it limits the control flow of the application preventing us from gathering metrics at a regular interval and storing past data. We do want to acknowledge that the plugin functionality allows the use of the energy-based scheduling for a wider audience and that by only adding a plugin the rest of the scheduler can evolve around the energy-based filtering.

4.4 Compare kube-scheduler to Escheduler

We want to compare the default Kubernetes scheduler, kube-scheduler, to our scheduler, Escheduler. However, we also want to be able to reproduce the comparison for future schedulers. We need to choose a load to perform the benchmark with. We also need to ensure reproducibility by providing a framework to rerun the experiment with the full energy metric pipeline setup. This is done via an automatic deployment system as seen in Figure 4.1.

4.4.1 Benchmark using DeathStarBench

We use DeathStarBench (DSB) for the benchmark. We specifically focus on the social-network variant which is "A social network with unidirectional follow relationships, implemented with loosely-coupled microservices, communicating with each other via Thrift RPCs." (2). The load is generated using an adapted version of wrk2 also in the DeathStarBench repository. wrk2 generates an HTTP load on a given URL using a Lua script to generate the requests. We run social-network using helm with 3 replicas for every pod. Helm also specifies the scheduler that will be used. We then create scheduling events by killing 3 pods every minute.

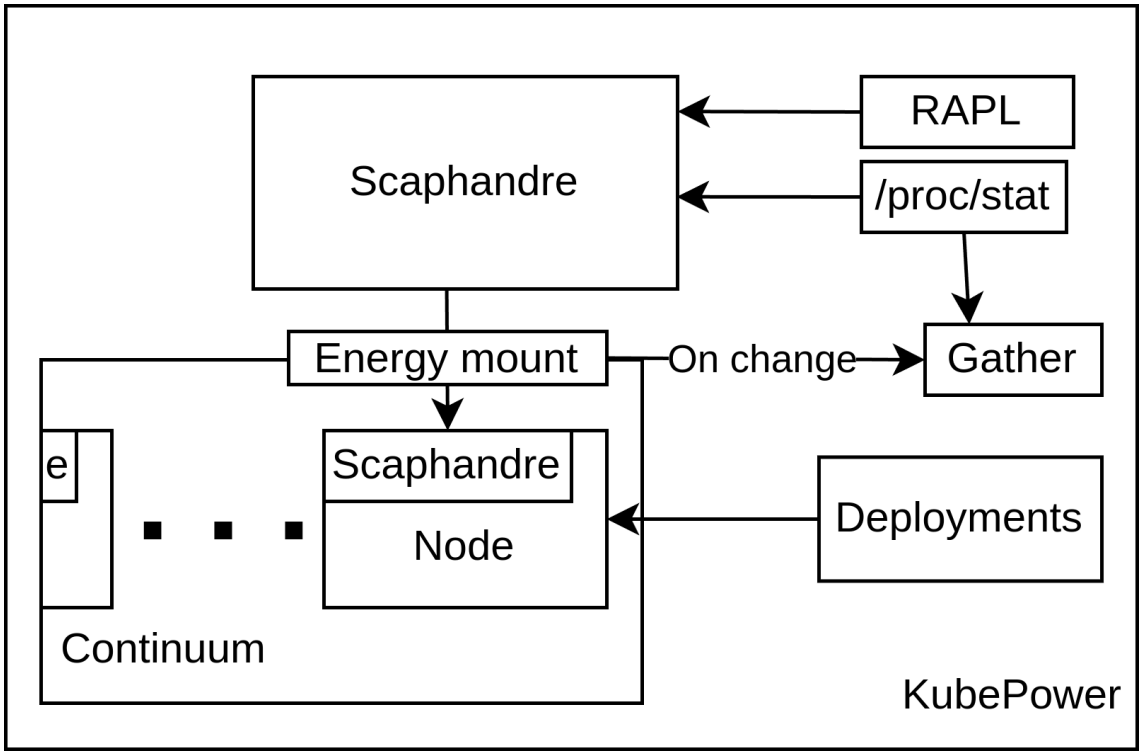

Figure 4.1: Automated deployment, setup, and execution of experiments.

4.4.2 Automated Benchmarking

Continuum cannot handle all setup to prepare for the benchmark. Continuum lacks flexibility and debug capabilities when it comes to starting host processes and configuring and starting guest VM processes. We implemented a script to perform the setup and measurements automatically based on the provided command line arguments. The automated benchmark tool starts Continuum, setups the remainder on the guest VM depending on the configuration, starts Scaphandre, performs the measurements for the benchmark, and gracefully shuts down all processes and VMs afterward. The measurements are taken every time a VM's shared energy file gets updated for that VM using /proc/stat for the CPU and the shared energy file for the energy usage. Both the energy file as well as /proc/stat contain a counter which we store the difference compared to the last measurement in a file. With this we automated the experiments and made it reproducible (**C4**).

5

Evaluation

In this section, we will present the design of the experiments and the results to answer **RQ1**, **RQ2**, and **RQ4** to compare schedulers in a reproducible manner based on energy usage. We will go over the used host and its potential impact on the result, establish multiple baselines to compare the overhead and impact of running our scheduler Escheduler, explain how the metrics are retrieved, and iscuss the aforementioned metrics were chosen over alternatives. We conclude this section with the results of the experiments and compare the default Kubernetes scheduler to our scheduler, Escheduler.

We investigate the relation between CPU usage and energy usage. We measure the processing time of Scaphandre under varying amounts of nodes. The evaluation of Escheduler and the comparison to the default Kubernetes scheduler is focused on energy usage over the duration of the benchmark. A brief summary of the findings:

1. We measured the energy overhead incurred by the different components from the pipeline as described in 4.2 including a baseline for DeathStarBench and Escheduler: We find that energy overhead for individual components is consistent, the kube baselines have a small increase in usage considering that the node contains 2 cores as opposed to 1. The baseline for DeathStarBench and Escheduler show that they are resource-intensive even during idling. DeathStarBench has a lot of nodes with communication taking up resources, while Escheduler uses a lot of resources gathering the metrics every minute.

2. We also looked at Scaphandre computed energy usage vs CPU usage using a CPU-only maximum load benchmark: We find that Scaphandre is reasonably consistent but suffers under high load as it can overrepresent minor deviations at max usage.

3. We measured the reporting latency of Scaphandre passing energy usage metrics for different node configurations: We find that the base processing time for Scaphandre is high and consistent at 7 seconds for a low amount of nodes and gets less consistent for higher node counts although reporting latency is lower.

4. We observed the relation between CPU usage and energy usage during the component and node overhead baselines: We find that for a low amount of nodes, insignificant variance is found between reporting times across nodes, more nodes result in higher reporting latency. However, the ratio between CPU usage and energy usage is approximately the same.

5. We measured the energy usage over the entire duration of a benchmark for the default Kubernetes scheduler and for Escheduler: We find that the default scheduler is better overall as it doesn't suffer from a lack of information and distributes workload in an energy-efficient way for a homogeneous system. Escheduler is close to the default Kubernetes scheduler's energy usage but requires regular data gathering resulting in more overall energy usage.

5.1 Experimental Setup

We first have to establish various baselines to single out the overhead of running individual components. We are working with multiple VMs each of which will act as a node in a Kubernetes cluster. This Kubernetes cluster will be running DeathStarBench's social network, a website benchmark suite, with RAPL energy usage metrics. We want to compare Escheduler to the Kubernetes default scheduler in terms of energy usage and end-to-end-The energy usage is computed at the host using RAPL readings processed by Scaphandre, propagated to the VM using virtiofsd, and exposed in Kubernetes via Prometheus which gets data from the guest instance of Scaphandre providing a service monitor.

This setup contains many components for which we want to single out the overhead and maximum throughput and is increasingly dependent on other components as described in Table 5.1. The aforementioned baselines will be started by Continuum using a setup with a single controller and single worker node both leveraging 5 cores. The controller also acts as a worker node but has the additional responsibility of scheduling and managing the cluster when Kubernetes is running. The performance benchmarks will be based on a setup with a 1 controller node and 8 worker nodes so that every node has 2 full cores (2 cores required to run Kubernetes node) to use as described in Table 5.2. We stop at 9

VMs to keep a core free for the host to run Scaphandre. An additional baseline experiment is run to understand the behavior of the impact of nodes on the full experimental setup (kube_sca_dsb_sched as described in Table 5.1) leveraging all individual components. The impact of the number nodes on the reporting times of Scaphandre is measured with different amounts of nodes to observe the scalability.

Baseline	QEMU	Kubernetes	Prometheus	virtiofsd	Scaphandre	CPU 100%	DSB	Scheduler
qemu	✓							
qemu_virtiofsd	✓			✓				
kube	✓	✓						
kube_prom	✓	✓	✓					
kube_sca	✓	✓	✓	✓	✓			
kube_dsb	✓	✓	✓	✓	✓		✓	
qemu_cpu100	✓			✓		✓		
kube_cpu100	✓	✓		✓		✓		
kube_sched	✓	✓	✓	✓	✓			✓
kube_dsb_sched	✓	✓	✓	✓	✓		✓	✓

Table 5.1: Cummulative baseline configurations. Highlighting what components are measured in each baseline.

	Host	**Guest**	**Additional info**
OS	Ubuntu 22.04.3 LTS	Ubuntu 20.04.6 LTS	
CPU	20 cores x 1 thread	1-19 cores	2 x Xeon Silver 4210R CPU @ 2.40GHz
RAM	256 GB	25 GB	4 x 64GB DDR4 @ 2933 MHz

Table 5.2: Hardware specifications for Host and guest(s).

5.1.1 Scheduler Benchmark

Both schedulers will be benchmarked using the same benchmark suite. The setups for both will be kept with the full energy metrics pipeline as described in 4.2 to still enable full observability on pod metrics. The schedulers will be benchmarked using DeathStarBench with a HTTP benchmarking tool called wrk2. The load generated by wrk2 is designed to trigger DeathStarBench scaling to increase the number of scheduling decisions being made. Escheduler has no energy usage information on startup and gathers this information while

running. Gathering energy usage information is done by measuring energy usage of running pods, so every scheduled pod adds additional energy usage information. We want to trigger this scaling in a controlled manner to see the influence of information on the scheduling decision in Escheduler. To achieve the scaling in a controlled manner we will use regular benchmark intervals of 3N seconds with M seconds to scale down in between the load. We can scale the number of pods used by DeathStarBench by using a Redis with a replicated setup as the replicated setup scales to accommodate a larger volume of reads, meaning we can scale it linearly with reads. The scaled pods in a replicated setup still share the same master which means we can also easily put load on the Redis instances by doing a write operation to the master Redis instance that needs to be propagated to the slaves using a compose action. Every benchmark interval will consist of an initial burst of compose posts (writes) for N seconds which will put load on the current pods, but not scale as it is a write operation to a single master, then scale by doing a burst of read operations for N seconds and then in the final N seconds do a mixture of compose and read to get a master that has to update the slaves often. So we will put load on the existing pods, trigger a scaling, and put a load on the the existing pods (including the new pods created for scaling). After the benchmark interval, we give a minute to scale down to restart the experiment. Repeating the experiment with a scale down period in between ensures that Escheduler still has a filled cache of previous energy usage information.

5.2 Experiment Results

- Component overhead 5.2.1: Step-wise measurement of the setup overhead, adding an increasing amount of components to single out the overhead of individual components as described in Table 5.1.
  - We will focus on QEMU and Kubernetes components. Isolating the components by having a minimal setup. Only the controller (in case of QEMU only one VM exists) will be measured.
  - We will test QEMU and Kubernetes via a 100% CPU load. It is worth noting that the minimal setup for Kubernetes requires 2 nodes (1 controller and 1 worker node) with each 2 cores, and we will only measure on the worker node.
- Node overhead 5.2.2: Testing the reporting latency of Scaphandre (using qemu baseline setup as described in Table 5.1) with an increasing amount of nodes.

- CPU versus Energy usage 5.2.3: We determine the correlation of CPU and energy usage using the measurements from the baselines. We calculate the ratio of differences between measurements (the difference in absolute values between measurements i - 1 and i) and compare the ratio of the CPU and energy measurements.
- Results kube-scheduler vs Escheduler 5.2.4: Results from the default Kubernetes scheduler kube-scheduler, and our scheduler, Escheduler, using the scheduler benchmark 5.1.1.

5.2.1 Component Overhead

The technology stack to enable Escheduler running in a Kubernetes cluster ran on VMs consists of several layers running on top of each other. We need a Scaphandre instance on the host, run the VMs, propagate the energy usage metrics via virtiofsd to the guest VM, run Kubernetes, enable Prometheus observation, expose the propagated energy usage in Prometheus using another instance of Scaphandre per VM (with each VM representing a Kubernetes node), run the scheduler Escheduler, and run DeathStarBench on top of the entire stack in Kubernetes for the benchmarks. We also need to know how Scaphandre and Kubernetes behave under load by consuming all compute resources available to the VM to see if a high load influences the measurements being taken. The measurements happened over 10 minutes and was repeated 6 times equaling an hour of data for every component.

The results can be seen in Figure 5.1 and Figure 5.2. We can conclude that QEMU, QEMU with virtiofsd, and Kubernetes have no significant overhead. Prometheus has very low cost but peaks more often due to metric reading. Scaphandre is more stable but this is within margins of error as the system had less interference by other VMs at the time. The cost of DeathStarBench and Escheduler is very high but the combined cost being lower than the individual cost of Escheduler is due to Escheduler being influenced by a high amount of VMs consuming resources. However, when no pods are consuming resources Escheduler still takes up the most resources as it regularly gathers information activating Prometheus and Kubernetes API server to do so. We can also see that the QEMU CPU load 100 varies by a considerable amount (almost 17%) which is due to inconsistencies with the virtualized CPU.

5.2.2 Node Overhead

Scaphandre measures the energy usage of the system and splits this per process. The process will be linked to a VM and the resulting energy usage of the VM is placed in a

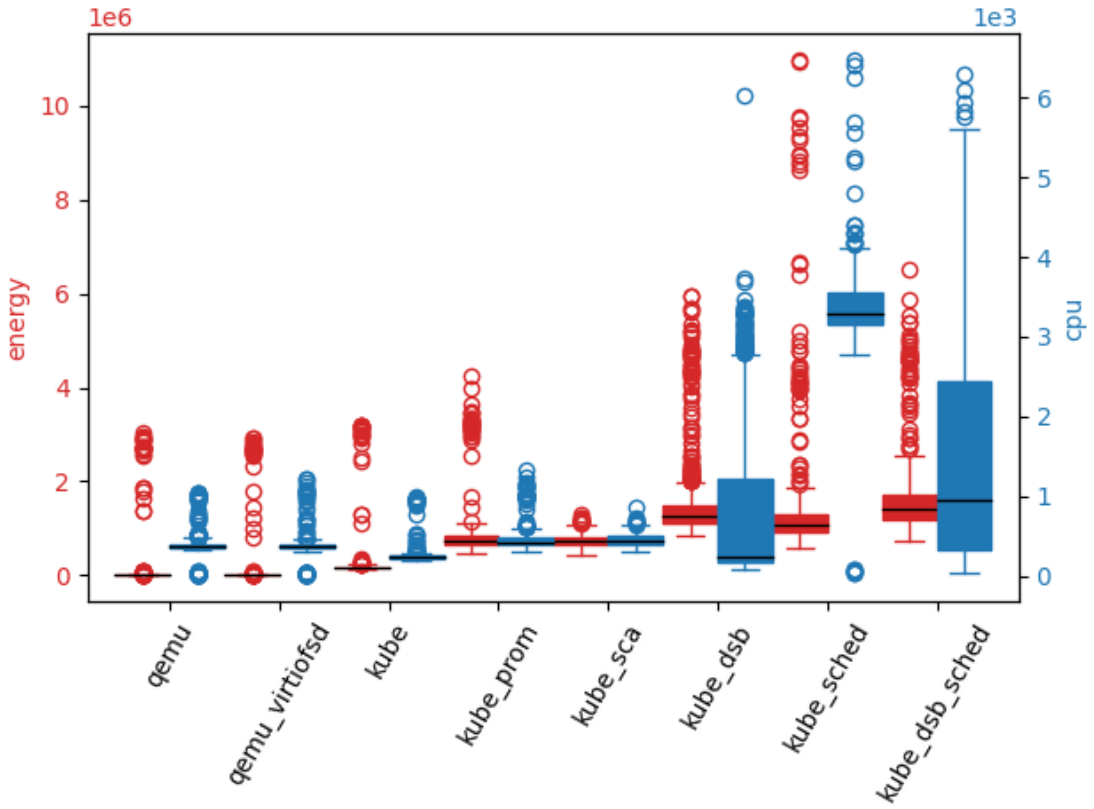

Figure 5.1: Individual component overhead of idle QEMU and Kubernetes components by separating induced overhead of different components running in tandem.

file shared between the host and the guest VM. The setup used is qemu as described in Table 5.1 as we only monitor the shared files. The reporting latency will be measured for 1 node to observe the base reporting latency and for 4, 8, 12, 16, and 20 (each node has 1 core) to show how reporting latency is influenced by node count. The measurement with 20 nodes has every core assigned to a VM, but as the CPU is not pinned and the VM will be mostly idle it won't impact Scaphandre. The measurements happened over 10 minutes for each baseline.

The results can be seen in Figure 5.3. We can conclude that the node overhead is significant. With a low amount of nodes we stay around a fairly consistent 7 seconds latency with reporting times being close to each other as well. But as we scale up we notice that the reporting times increase and the time at which energy metrics are given are scattered. The wider range of reporting times does result in lower overall reporting times as Scaphandre puts out metrics more often as it finishes computing at different intervals and immediately propagates the result.

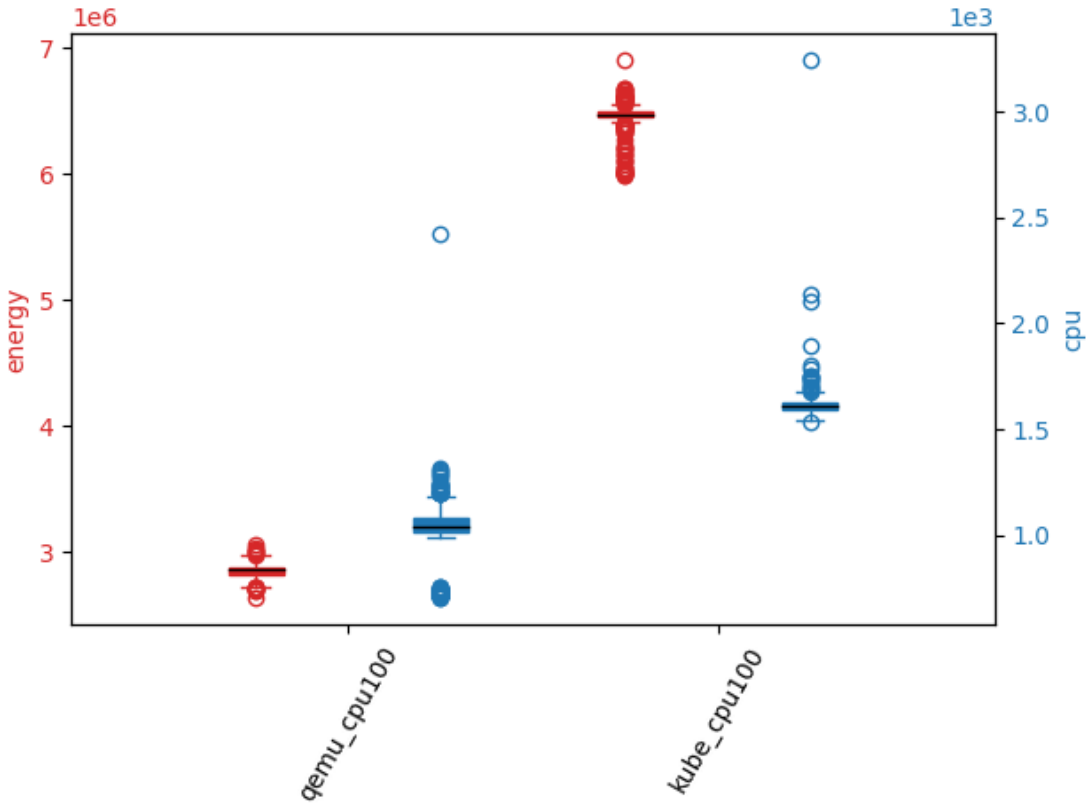

Figure 5.2: Reference of the energy and CPU capacity of QEMU and Kubernetes.

5.2.3 CPU versus Energy Usage

The major contribution of this thesis is that we created a scheduler Escheduler based on real-time energy metrics (**C2**), but we have to verify the accuracy of the energy metrics. An inaccurate energy usage being reported by Scaphandre could lead to worse scheduling decisions by Escheduler. The energy usage being measured is based on RAPL reporting. RAPL reports energy usage for CPU, on-chip GPU, cache, and DRAM. Our setup does not include the on-chip GPU (nor the external GPU). Furthermore, the cache and DRAM will be a minor part of the energy usage compared to the CPU. For these reasons, we focus on the CPU ticks from /proc/stat versus Energy usage comparison where we use an application that has insignificant memory and cache usage.

To summarize this thesis focuses on 2 metrics:

- CPU usage based on ticks consumed by the VM from the counters in /proc/stat.
- Energy usage from Scaphandre using the counters processed and put into the shared

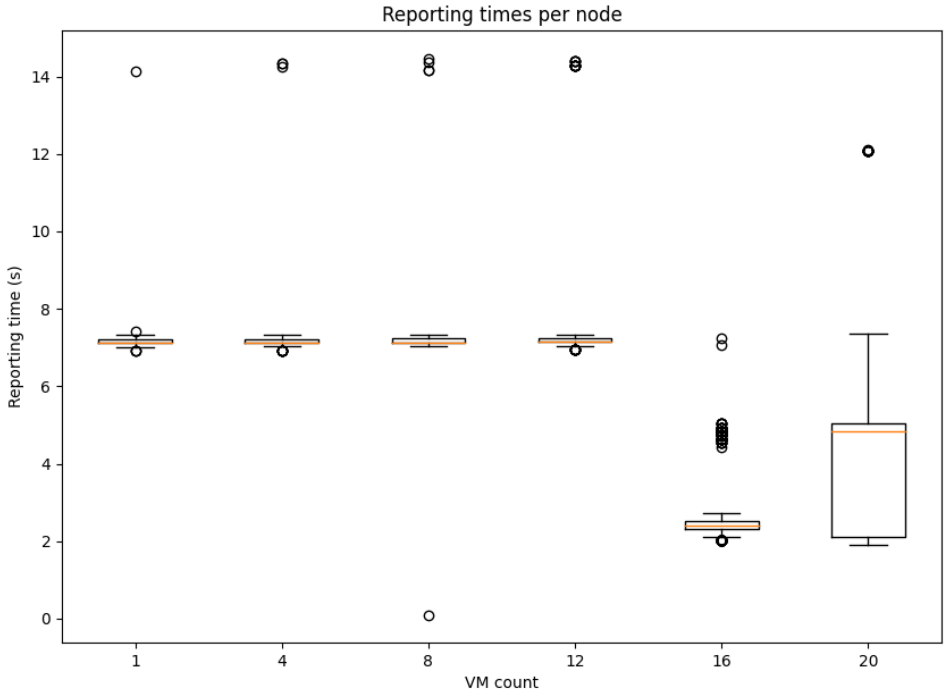

Figure 5.3: Influence of amount of nodes on reporting times when running the qemu setup as described in Table 5.1.

files for each VM.

The application to stress the CPU to 100% usage will let all cores continuously do a simple multiplication in a loop until the benchmark is over. This benchmark will be run on a bare VM with the qemu_cpu100 setup and on the Kubernetes nodes using the kube_cpu100 setup as described in Table 5.1. The measurements had a duration of 10 minutes and were repeated 6 times equaling an hour of data. All measurements are performed on the host.

So the CPU benchmarks with a load of 100% will look like:

- A bare VM using the qemu_cpu100 setup with 1 core running for 10 minutes with the experiment repeated 6 times.
- Two VMs running a Kubernetes cluster using the kube_cpu100 setup with each 2 cores with the worker node running the program to load test the node. The worker node is the node being measured for a duration of 10 minutes with the experiment repeated 6 times.

We observed the overhead of individual components in Section 5.2.1. However, the component overhead baselines do not show the full picture as they focus on the overhead brought by individual components in terms of both energy and CPU losing data about how closely correlated energy and CPU are. The results from the CPU 100% load are particularly interesting as they stress both the energy usage and CPU. Every CPU usage fluctuation should be reflected in the energy usage as well. We observe that QEMU as seen in Figure 5.4 has really stable energy usage and comes close to the CPU but does not match the CPU outliers. A perfect energy measurement would mimic the CPU usage as it is the largest energy consumer. The measurements are taken every time a new energy reading is published. We observe that Kubernetes as seen in Figure 5.5 fluctuates from time to time in both energy usage and CPU usage. The energy usage fluctuations however are often way higher than the spikes in CPU usage. The differences between the ratios are likely attributed to a Scaphandre allocation error. Scaphandre measuring total system energy and dividing the total system energy over all processes, but the process of measuring induces a slight delay where the energy metrics do not match reality. When one process doesn't get the right amount of energy usage assigned it is likely to affect other processes as well due to the computed share being skewed.

5.2.4 Results kube-scheduler vs Escheduler

We will now evaluate the default Kubernetes scheduler, kube-scheduler, against Escheduler. We run DeathStarBench with 3 replicas for every chart under a constant HTTP load taking using 8 threads, 64 connections, and 2056 requests per second using wrk2. While doing so we kill 3 nodes every minute which need to be rescheduled by the respective scheduler. We make sure to not kill nginx and media-frontend as these services require multiple minutes to start up. We ran this benchmark for 1 hour for both schedulers with the same system conditions.

The results can be seen in Figure 5.6. We can see that the energy usage and the overall CPU usage of Escheduler is higher. This is due to the frequent metrics being gathered which happens every second to gather past data as well as for every scheduling event. The CPU usage fluctuates less than for kube-scheduler for this same pattern as Escheduler is always performing some operation, while kube-scheduler only has to do so occasionally. Escheduler also takes longer to schedule as it has to wait for responses for different metrics on the nodes and pods before making a decision. The scheduling delay results in slower startup times as pods to be scheduled will have to wait longer in the queue. The scheduling decisions made between Escheduler and kube-scheduler are similar as they distribute load

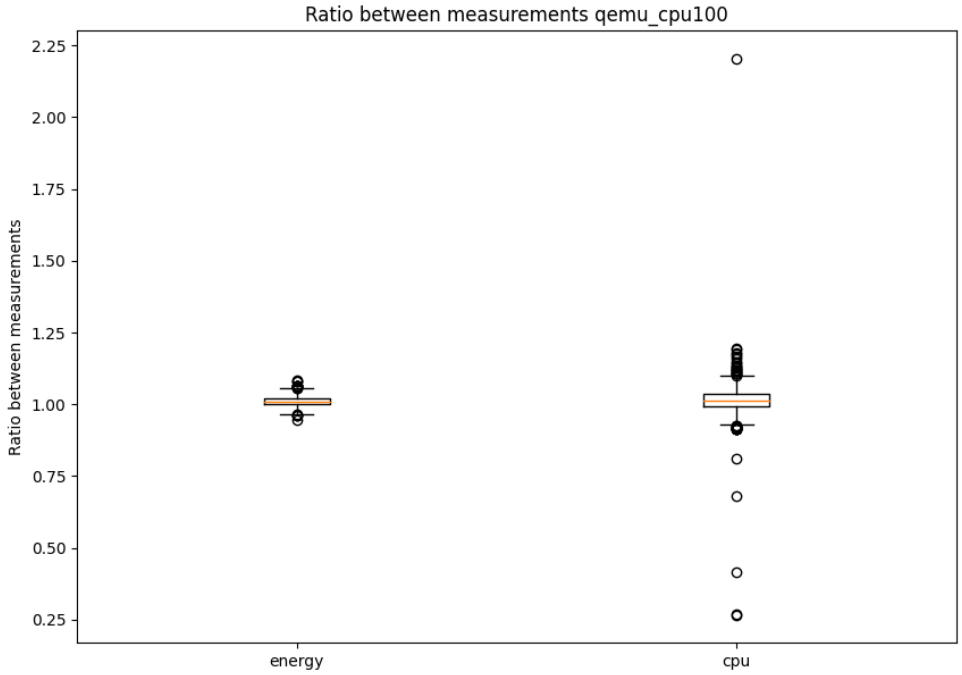

Figure 5.4: Correlation between CPU usage and energy usage for qemu_cpu100 as described in Table 5.1.

in a similar manner so the lower costs on the controller node is what made kube-scheduler the more energy efficient scheduler.

5.3 Threat to Validity

We measured the results when little occurred on the shared computing node, but not all baseline runs were run in isolation as other VMs from other users were not shutdown. This could not be prevented as the experiments had to be rerun in a late stage do to an issue during measuring. The impact should be insignificant on other experiments as the additional VMs did not use significant resources and we tested idle VMs, with the exception of the Escheduler baseline as this needed to be rerun when the node was busy with other experiments. Escheduler baseline is an idle test and shouldn't influence others but the measurements fluctuate more with a higher amount of VMs taking resources as Scaphandre computes the energy usage of all VMs.

Scaphandre fluctuates more in a virtualized environment compared to directly on the

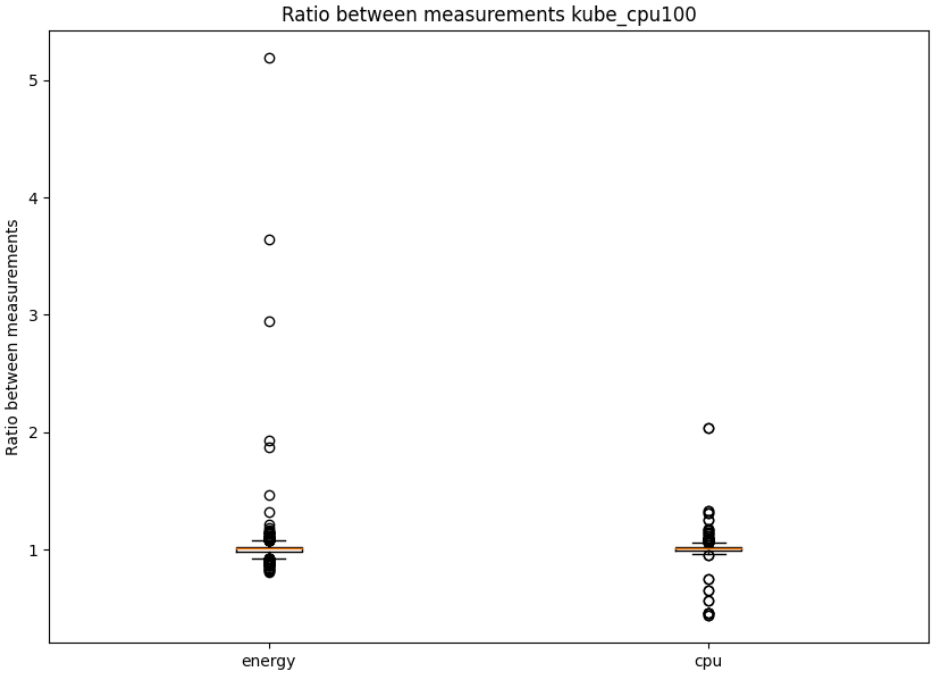

Figure 5.5: Correlation between CPU usage and energy usage for kube_cpu100 as described in Table 5.1.

host. We directly read any shared file changes so the transfer time to the guest VM is not measured, yet we still observed differences compared to /proc/stat which we used to measure CPU ticks between measurements. We avoided measuring from the exposed Prometheus energy metrics as it would use additional resources in the VM.

We did not test PowerAPI and cannot conclude whether Scaphandre or PowerAPI is better. We can only conclude that both provide similar functionality on paper. We also didn't try alternative file-sharing modes or different startup methods for Scaphandre inside the VM.

DeathStarBench has over 25 pods which we replicated 3 times to get to 75 pods. We killed these pods to get new scheduling events. Rescheduling pods is the optimal scenario for Escheduler as it allows for better reuse of past data. A system with more different tasks would give a greater advantage to the default Kubernetes scheduler, kube-scheduler.

We benchmarked on a homogeneous cluster of VMs. VMs with differing resources could show entirely different results. We used a homogeneous cluster to build on functionality Continuum provides in setting up a homogeneous cluster of VMs.

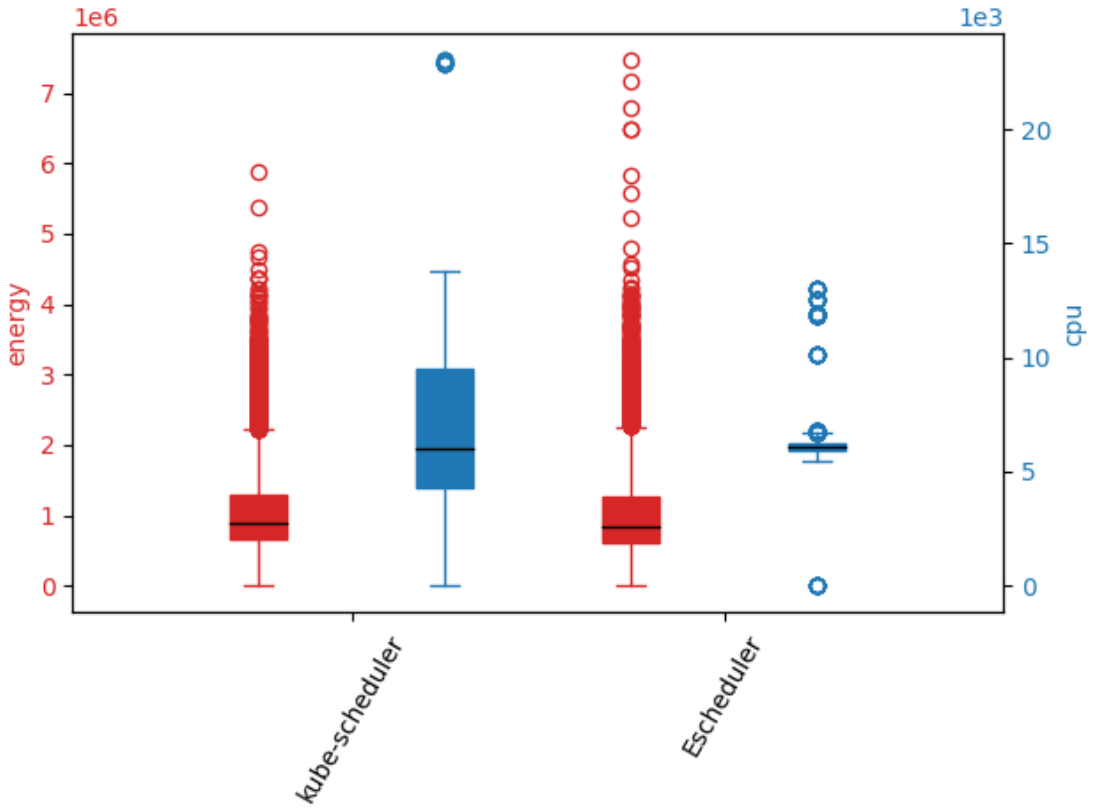

Figure 5.6: Energy and CPU consumption of kube-scheduler and Escheduler of the entire Kubernetes cluster.

5.4 Discussion on Evaluation

We see that running the energy metrics pipeline as described in 4.2 has a minimal impact on overall costs as it doesn't consume a lot of resources. The pipeline works best under a lower amount of nodes as there is less fluctuations in reporting times and accuracy. Escheduler uses a lot of resources to simply gather all necessary metrics and has to do so for every pod scheduling event as well making Escheduler expensive to run. Scaphandre does suffer under high load but is reasonably close in approximating the energy used. Escheduler had no advantage over kube-scheduler due to the amount of resources it took to consume the pipeline information. More resources are needed when more pods are in the cluster as that requires more fetching from different pods. More VMs also forces Scaphandre to gather metrics for more instances, degrading accuracy slightly and having a big influence on reporting times. Future implementations should focus on using past data retrieved at regular intervals more to prevent to much overhead on every scheduling event. The insight

into energy usage is however accurate and Prometheus allows developers and corporations to make decisions based on pod energy consumption.

6

Lessons Learned

During this thesis a lot of progress deemed unnecessary. Some of the progress being unnecessary was unavoidable like the node I was working on automatically updating to a Ubuntu 22.04 while I spend the past few months working on Ubuntu 20.04 compatible approaches as I missed a lot of functionality in QEMU, Virtiofsd, and more. We could have avoided the wasted progress by immediately discussing a potential upgrade of the node OS or keeping the node OS the same and reverting the automatic update. However, by spending more time on the scope and keeping track of how well the scope is followed would allow for less unrelated work being done.

Another problem was Continuum being unstable which we could have avoided by completing the infrastructure requirements at once rather than spreading it out. The Kubernetes package migration issue solved in this thesis would still have occurred but that would also take less time if the scope of Continuum's use was better defined at the start of the thesis.

Development speed was limited by the many VM startups which should have been reduced to spend less time waiting on results.

The development of the scheduler specifically proved more difficult due to the Kubernetes environment changing the authentication settings due to a minor update that was automatically applied. Authentication was factored in too late and should have been recognized earlier as a potential threat causing a delay. The scheduler also suffers from limited data on pod placement. Scaphandre gives pod statistics based on the pod name and instance IP, however, the node gets picked based on node name leading to assumptions built-in to the scheduler to retrieve this information using a heuristic.

7

Conclusion

We conclude that our scheduler Escheduler uses more energy to achieve similar scheduling decisions. Escheduler schedules purely based on the energy metrics but gathering the metrics is expensive. Scheduling is also slower as it takes time to gather the metrics to make a scheduling decision. Escheduler receives the real-time energy metrics from the energy metrics pipeline that utilizes Scaphandre on the host and Scaphandre on the guest VM to compute the energy usage per pod in the Kubernetes cluster. We can access the metrics via Prometheus.

We also showed how the component overhead of the metric pipeline is insignificant and that Scaphandre is accurate when the system is not under load but struggles with bigger fluctuations in the CPU as it often overestimates the energy usage.

We also see that Scaphandre works best with a low amount of virtualized VMs as the reporting times fluctuate more with more VMs. The difference in reporting time can in turn cause Escheduler to make assumptions that are wrong due to being based on incomplete data as the data is not up to date. The variance in reporting times was not taken into account in the scheduler as we do not keep track of the frequency of reporting and potential gaps in data. The average reporting latency goes down with more nodes.

All the experiments are reproducible including the measurements and the load generated. This can be useful to compare future schedulers as it provides easy access to the full metric pipeline.

A brief summary of the answers to the Research Questions 1.3:

RQ1 We measure energy usage inside a VM using our energy metric pipeline that uses RAPL to get real-time energy usage on the host, computes how much is caused by every guest VM using Scaphandre, sends it to the host using virtiofsd, uses another

Scaphandre instance on the guest VM to get the metrics inside the Kubernetes cluster per pod, and exposes the metrics to the outside using Prometheus. (**RQ1**)

RQ2 The energy usage is accurate under a stable load but suffers from inaccuracies during spikes. For a low amount of nodes (up to 12) the reporting latency is consistently at 7 seconds, but for more nodes the reporting latency fluctuates more as Scaphandre reports partial results more often. (**RQ2**)

RQ3 We implemented Escheduler which performs similar scheduling decisions as the default Kubernetes scheduler, kube-scheduler, using only energy metrics but uses more energy overall as the energy metric retrieval is expensive as it is done every second and for every scheduling decision. (**RQ3**)

RQ4 We automated the setup using an extended version of Continuum as well as a Python script to setup additional components after the startup of VMs and Kubernetes. The benchmarking and measurements are also automated with DeathStarBench's social-network with wrk2 HTTP load generator and a kill pod loop as the benchmark. (**RQ4**)

With the Research Questions answered we look back at the main question **MQ**: **How to integrate real-time energy metrics with a Kubernetes scheduler running in a VM?** The answer is that we transform RAPL readings into attributed energy per guest VM using a host instance of Scaphandre, transfer the attributed energy to the guest VM, process the per pod energy metrics using a guest instance of Scaphandre, expose the metrics to Escheduler with Prometheus, and schedule on the node with the largest remaining energy capacity using our ExEC scheduling algorithm.

7.1 Limitations and Future Work

Escheduler gathers metrics every second and for every scheduling decision. The latter causes a scheduling delay as Escheduler has to wait for the metrics. Implementing a time series query could remedy this shortcoming but was not possible in this thesis due to time constraints.

Escheduler was built using metrics split over multiple VMs, which get requested using Prometheus. Some metrics might be able to be shared using a less expensive method reducing the overall costs of metrics gathering in the scheduler.

Experiments couldn't always be run in isolation on the computer as due to time constraints there was a lot of contention on the computer potentially influencing the results.

The energy metric pipeline is promising as it provides energy metrics with little overhead. The problem this thesis suffered from is that the requests to Prometheus take up a large portion of the time, but better reuse of metrics could remedy the amount of requests required to achieve a scheduling decision.

References

[1] Matthijs Jansen. **Continuum**, 2022. ix, 9, 50

[2] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. **DeathStarBench**, 2019. ix, 10, 20

[3] Benoit Petit. **scaphandre**, 2023. ix, 9, 51

[4] virtio fs. **virtiofsd**, 2020. x, 10

[5] Dutch Data Center Association. **State of Dutch Data Centers 2022**, 2022. 1

[6] Ministerie van Economische Zaken en Klimaat Directie Digitale Economie. **De staat van de digitale infrastructuur**, Jan 2024. 1

[7] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, 2022. 1

[8] Tommy van der Vorst, Marenne Massop, and Adriaan Smeitink. **Dialogic - De digitale voetafdruk. Emissies van de digitale sector in (toekomst)perspectief**, Sep 2023. 1

[9] Martijn Koot and Fons Wijnhoven. **Usage impact on data center electricity needs: A system dynamic forecasting model**. *Applied Energy*, **291**:116798, 2021. 2, 3

[10] Ergun Biçici. **A Cloud Monitor to Reduce Energy Consumption With Constrained Optimization of Server Loads**. *IEEE Access*, **12**:25265–25277, 2024. 2

[11] Youssef Saadi, Soufiane Jounaidi, Said El Kafhali, and Hicham Zougagh. **Reducing energy footprint in cloud computing: a study on the impact of clustering techniques and scheduling algorithms for scientific workflows**. *Computing*, **105**(10):2231–2261, 2023. 2

[12] Jinjiang Wang, Hangyu Gu, Junyang Yu, Yixin Song, Xin He, and Yalin Song. **Research on virtual machine consolidation strategy based on combined prediction and energy-aware in cloud computing platform**. *J. Cloud Comput.*, **11**:50, 2022. 2

[13] Jingbo Li, Xingjun Zhang, Zheng Wei, Jia Wei, and Zeyu Ji. **Energy-aware task scheduling optimization with deep reinforcement learning for large-scale heterogeneous systems**. *CCF Trans. High Perform. Comput.*, **3**(4):383–392, 2021. 2

[14] Mohammad Aldossary. **A Review of Energy-related Cost Issues and Prediction Models in Cloud Computing Environments**. *Comput. Syst. Sci. Eng.*, **36**(2):353–368, 2021. 2

[15] Mohammad Masdari, Sayyid Shahab Nabavi, and Vafa Ahmadi. **An overview of virtual machine placement schemes in cloud computing**. *J. Netw. Comput. Appl.*, **66**:106–127, 2016. 2

[16] Bin Shi, Lei Cui, Bo Li, Xudong Liu, Zhiyu Hao, and Haiying Shen. **ShadowMonitor: An Effective In-VM Monitoring Framework with Hardware-Enforced Isolation**. In Michael D. Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, **11050** of *Lecture Notes in Computer Science*, pages 670–690. Springer, 2018. 3

[17] Hossein Aqasizade, Ehsan Ataie, and Mostafa Bastam. **Kubernetes in Action: Exploring the Performance of Kubernetes Distributions in the Cloud**. *CoRR*, **abs/2403.01429**, 2024. 3

[18] Attique Ur Rehman, Songfeng Lu, Mubashir Ali, Florentin Smarandache, Sultan S. Alshamrani, Abdullah Alshehri, and Farrukh Arslan. **EEVMC: An Energy Efficient Virtual Machine Consolidation Approach for Cloud Data Centers**. *IEEE Access*, **12**:105234–105245, 2024. 4

[19] Naim Kheir. *Systems Modeling and Computer Simulation*, **2**. Routledge, New York, NY, USA, 1996. 7

[20] Yair Levy and Timothy J Ellis. **A systems approach to conduct an effective literature review in support of information systems research**. *Informing Science*, **9**:181–212, 2006. 7

[21] A. Iosup, L. Versluis, A. Trivedi, E. van Eyk, L. Toader, V. van Beek, G. Frascaria, A. Musaafir, and S. Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776, Los Alamitos, CA, USA, jul 2019. IEEE Computer Society. 7

[22] Richard R Hamming. *Art of doing science and engineering: Learning to learn*, **1**. CRC Press, London, LDN, UK, 1997. 7

[23] Marcus A. Rothenberger Ken Peffers, Tuure Tuunanen and Samir Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *Journal of Management Information Systems*, **24**(3):45–77, 2007. 7

[24] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*, **1**. John Wiley & Sons Inc., New York, NY, USA, 1991. 7

[25] Gernot Heiser. **Systems Benchmarking Crimes**, 2018. 7

[26] John Ousterhout. **Always Measure One Level Deeper**. *Commun. ACM*, **61**(7):74—-83, Jul 2018. 7

[27] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer,] Fotis Psomopoulos, Tony Ross-Hellauer, René Schneider, Jon Tennant, Ellen Verbakel, Helene Brinken, and Lambert Heller. *Open Science Training Handbook*. FOSTER, 2018. 7

[28] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. **The FAIR Guiding Principles for scientific data management and stewardship**. *Scientific data*, **3**(1):1–9, 2016. 7

[29] Emery D. Berger, Stephen M, Blackburn, Matthias Hauswirth, and Michael W. Hicks. **A Checklist Manifesto for Empirical Evaluation: A Preemptive Strike Against a Replication Crisis in Computer Science**, Aug 2019. 7

[30] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is Big Data Performance Reproducible in Modern Cloud Networks?** In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 513–527, Santa Clara, CA, Feb 2020. USENIX Association. 7

[31] Aurelien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. **PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level**. *ERCIM News*, **2013**(92), 2013. 9

[32] sustainable computing.io. **Kepler**, 2022. 9

[33] Mathilde Jay, Vladimir Ostapenco, Laurent Lefèvre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. **An experimental comparison of software-based power meters: focus on CPU and GPU**. In Yogesh Simmhan, Ilkay Altintas, Ana Lucia Varbanescu, Pavan Balaji, Abhinandan S. Prasad, and Lorenzo Carnevale, editors, *23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2023, Bangalore, India, May 1-4, 2023*, pages 106–118. IEEE, 2023. 10

[34] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. **Red Alert for Power Leakage: Exploiting Intel RAPL-Induced Side Channels**. In Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung, editors, *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 162–175. ACM, 2021. 10

[35] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. **An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18. Association for Computing Machinery, 2019. 10

[36] Tim van Kemenade. **thesis-energy-pipeline-and-scheduler**, 2024. 49, 50, 51

Appendix A

Reproducibility

A.1 Abstract

This thesis focused on an energy metric pipeline, the accuracy and frequency of the energy metrics, our energy-based scheduler Escheduler, and the reproducibility using Continuum and a Python script. The automated script can setup the pipeline, scheduler, and experiments. The measurements are also performed by the script for all experiments and everything is shutdown afterward (unless configured not to). We also provide the script that compiled the plots.

A.2 Artifact check-list (meta-information)

- **Program: Continuum, Scaphandre, Kubernetes scheduling, Prometheus, virtiofsd, and DeathStarBench**
- **Compilation: Rust, Golang, and Python**
- **Run-time environment: 1 controller node and 8 worker nodes**
- **How much disk space required (approximately)?: 25GB was configured per Continuum node**
- **How much time is needed to prepare workflow (approximately)?: 20 minutes**
- **How much time is needed to complete experiments (approximately)?:**
- **Publicly available?: Yes at (36)**

A.3 Description

A.3.1 How to access

All the code can be found publicly on GitHub (36). The repository contains **Escheduler** written in Golang, **block_cpu** written in Rust, the **automated experiments** written in Python, and the plotting script written in Python.

1. **Escheduler**: Located in the escheduler folder. Written in GoLang.
2. **block_cpu**: Located in the block_cpu folder. Written in Rust.
3. **Automated experiments**: Located in the continuum folder which also contains a modified version of Continuum (1).
   (a) **Continuum fork**: contains additional configurations in the configuration/tkemenade folder, additional documentation (–enable-virtiofsd needs to be used when configuring QEMU), adjusted code to enable virtiofsd, and adjusted code place the mount to /var/lib/libvirt/scaphandre/{domain_name}.
   (b) **Automated experiments main**: energy_metrics.py performs the automated experiments using Continuum to start the VMs and storing results in res and storing additional configuration in res/config (res/config/write_as_file are files used by the Python script to overwrite files in pulled git sources rather than creating a new fork).
4. **Graphing**: Located in the root of the project, graphing.py, and uses the res folder to pull results from and plot all uncommented plots.
5. **Original results**: Located in the res folder. We separated the original experiment results in case anything is rerun to prevent confusion.

A.3.2 Hardware dependencies

- **Cores**: The maximum cores used by the experiments is 20 cores for the node baseline to see Scaphandre reporting latency. The scheduler comparison uses 2 cores per node with 1 controller node and 8 worker nodes totalling 2 * (1 + 8) = 18 cores with 1 thread for the load and 1 main thread to measure this also brings us to 20 required cores (or 19 cores if every core can perform 1 thread).

- **Storage**: The storage configured for every node started by Continuum is 25GB, everything should fit in 15GB as well, but this hasn't been tested.
- **Additional**: In addition to the requirements above the machine also needs access to RAPl to be processed by Scaphandre and has virtualization enabled to start the Kubernetes cluster on the VMs.

A.3.3 Software dependencies

The following software dependencies are necessary to run this paper:

- Scaphandre
- virtiofsd
- QEMU
- libvirt
- Continuum
- Kubernetes
- Prometheus
- Golang
- Rust
- Lua
- DeathStarBench

A.4 Installation

git pull the repository (36) and git pull Scaphandre (3) v1.0.0 and run as seen below:

- **Escheduler**: go build -a –ldflags '-extldflags "-static"' -tags netgo -installsuffix netgo . && sudo docker build . -t 192.168.1.101:5000/escheduler && sudo docker push 192.168.1.101:5000/escheduler.
- **block_cpu**: cargo install.
- **Scaphandre**: Compile with features flag qemu.

- **energy_metrics.py**: python energy_metrics.py -mi {time in seconds to measure} -r {number of times to repeat experiment} (OPTIONAL to not destroy VMs –keep-vms) -en {space separated list of experiment names from [baseline1, baseline4, baseline8, baseline12, baseline16, baseline20, qemu, qemu_virtiofsd, qemu_cpu100, kube, kube_prom, kube_cpu100, kube_sca, kube_sca_sched, kube_sca_dsb, kube_sca_dsb_sched, kube-scheduler, esched] }
- **graphing.py**: python graphing.py.

A.5 Evaluation and expected results

Run the following command: python energy_metrics.py -mi 300 -r 1 -en qemu kube baseline_12 esched This runs 4 very different experiment. The results are expected to match the graphs as seen in this thesis, but with less datapoints as only 5 minutes are measured and only once.

Appendix B

Self Reflection

This thesis took way longer than intended, plagued by delays due to unforeseen circumstances. I started the thesis wanting to create a scheduler that acts on energy information. However, throughout the thesis I spent most of my time on infrastructure, getting the energy metric pipeline to work. Matthijs gave the suggestion to focus on comparing the available tools to prevent the complexity getting too high. I kept my original idea of creating a scheduler with wishful thinking that things would soon fall into place. This only introduced more hurdles and combined with being forced to start a full-time job for money reasons I kept needing more time. This could have been prevented by myself if I spent more time on the scope of this thesis and regularly looked at progress to adjust if necessary. Another thing I did not do enough is ask for help. The regular meetings were more like status updates, highlighting some struggles but not asking for more guidance when necessary. I did too many different things and spent too much time on things that could have been resolved with a single message.