Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

SPECIALIZATION PROJECT 1

Powerstack: Implementation of an energy monitoring environment in Kubernetes

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
January 31, 2025

Study program:
Computer Science, M.Sc.

Imprint

Project: Specialization Project 1
Title: Powerstack: Implementation of an energy monitoring environment in Kubernetes
Author: Caspar Wackerle
Date: January 31, 2025
Keywords: energy efficiency, cloud, kubernetes

Study program:
Computer Science, M.Sc.
Zurich University of Applied Sciences

Supervisor 1:
Prof. Dr. Thomas Bohnert
Zurich University of Applied Sciences
Email: thomas.michael.bohnert@zhaw.ch
Web: Link

Supervisor 2:
Christof Marti
Zurich University of Applied Sciences
Email: christof.marti@zhaw.ch
Web: Link

Abstract

Energy efficiency in cloud computing has become a critical concern as data centers consume an increasing share of global electricity. This thesis investigates energy consumption at the container and node level in Kubernetes-based infrastructures, using KEPLER (Kubernetes-based Efficient Power Level Exporter) to monitor and analyze power consumption in a controlled test environment.

A bare-metal Kubernetes cluster was deployed on three identical servers, configured using K3s for lightweight orchestration and managed through Ansible for full automation. The entire system was designed to be fast to deploy, highly reproducible, and adaptable to different hardware environments. Configurations were centralized for easy reusability in future projects, ensuring that modifications could be made with minimal effort. Prometheus and Grafana were integrated to collect and visualize KEPLER's real-time energy consumption metrics. A series of controlled benchmarking experiments were conducted to stress CPU, memory, disk I/O, and network I/O, assessing KEPLER's accuracy in reporting power usage under varying workloads.

The results indicate that KEPLER credibly tracks workload-induced power variations at the CPU package level, though inconsistencies arise in non-CPU power domains. High idle power consumption was observed at the node level, suggesting that infrastructure energy efficiency must account for static consumption beyond dynamic workloads.

This thesis provides a foundation for further research into energy-efficient Kubernetes environments, including improving KEPLER's accuracy, extending workload profiling, and exploring automation-driven energy optimization strategies. The modular and automated deployment architecture ensures that the findings and methodologies can be readily adapted for use in other energy-related cloud research projects.

The accompanying source code for this thesis, including all deployment and automation scripts, is available in the **PowerStack**[**PowerStack**] repository on GitHub.

Contents

Abstract **iii**

1 Introduction and Context **1**
1.1 Significance of Energy Efficiency in Cloud Computing . . . 1
1.2 The Need for Energy-Efficient Kubernetes Clusters . . . 2
1.3 Objectives and Scope of this Thesis . . . 2
1.3.1 Context . . . 2
1.3.2 Scope . . . 2
1.3.3 Objectives . . . 3
1.3.4 Use of AI Tools . . . 4
1.3.5 Project Repository . . . 4

2 Architecture and Design **5**
2.1 Overview of Test Environment . . . 5
2.1.1 Hardware and Network . . . 5
2.2 Key Technologies . . . 6
2.2.1 Ubuntu . . . 6
2.2.2 Bare-Metal K3s . . . 7
2.2.3 Ansible, Helm, Kubectl . . . 7
2.2.4 Kube-Prometheus Stack . . . 7
2.2.5 KEPLER . . . 8
2.3 Architecture and Design . . . 9
2.3.1 Kubernetes Cluster Design . . . 9
2.3.2 Persistent Storage . . . 9
2.3.3 Monitoring Architecture . . . 9
2.3.4 Metrics Collection and Storage . . . 9
2.3.5 Repository Structure . . . 10
2.3.6 Automation Architecture . . . 11
2.4 KEPLER Architecture and Metrics Collection . . . 12
2.4.1 KEPLER Components . . . 12
2.4.2 KEPLER Data Collection . . . 12
2.4.3 KEPLER Power Model . . . 14
2.4.4 Metrics produced by KEPLER . . . 14

3 Implementation **15**
3.1 K3s Installation . . . 15
3.1.1 Preparing the Nodes . . . 15
3.1.2 K3s Installation with Ansible . . . 16
3.2 NFS Installation and Setup . . . 16
3.2.1 NFS Installation with Ansible . . . 16
3.3 Rancher Installation and Setup . . . 17
3.3.1 Rancher Installation with Ansible and Helm . . . 17
3.4 Monitoring Stack Installation and Setup with Ansible . . . 17
3.4.1 Prometheus and Grafana Installation with Ansible and Helm . . . 17
3.4.2 Removal Playbook . . . 18
3.5 KEPLER Installation and Setup with Ansible and Helm . . . 19
3.5.1 Preparing the Environment . . . 19

3.5.2 KEPLER Deployment with Ansible and Helm . . . 19
3.5.3 Verifying KEPLER Metrics . . . 20

4 Test Procedure **22**
4.1 Test Setup . . . 22
4.1.1 Benchmarking Pod . . . 22
4.1.2 Testing Pods . . . 22
4.1.3 Disk Formatting and Mounting . . . 23
4.2 Test Procedure . . . 23
4.2.1 CPU Stress Test . . . 23
4.2.2 Memory Stress Test . . . 23
4.2.3 Disk I/O Stress Test . . . 23
4.2.4 Network I/O Stress Test . . . 24
4.3 Data Analysis . . . 24
4.3.1 Data Querying . . . 24
4.3.2 Diagrams . . . 24

5 Test Results **25**
5.1 CPU Stress Test Results . . . 25
5.1.1 Container-Level Metrics During a CPU Stress Test . . . 25
5.1.2 Node-Level metrics during a CPU stress test . . . 30
5.1.3 Overall Conclusions . . . 32
5.2 Memory Stress Test Results . . . 32
5.2.1 Container-Level Metrics During a Memory Stress Test . . . 32
5.2.2 Node-Level Metrics During a Memory Stress Test . . . 34
5.2.3 Overall Conclusions . . . 36
5.3 Disk I/O Stress Test Results . . . 36
5.3.1 Container-Level Metrics During a Disk I/O Stress Test . . . 36
5.3.2 Node-Level Metrics During a Disk I/O Stress Test . . . 39
5.3.3 Overall Conclusions . . . 41
5.4 Network I/O Stress Test Results . . . 41
5.4.1 Container-Level Metrics During a Network I/O Stress Test . . . 41
5.4.2 Node-Level Metrics During a Network I/O Stress Test . . . 44
5.4.3 Overall Conclusions . . . 45

6 Discussion **47**
6.1 Conclusion and Evaluation . . . 47
6.1.1 Evaluation of Cluster Setup . . . 47
6.1.2 Evaluation of Monitoring Setup . . . 47
6.1.3 Evaluation of KEPLER . . . 48
6.1.4 Credible Takeaways from the test results . . . 49
6.2 Future Work . . . 49
6.2.1 Detailed Analysis of KEPLER . . . 49
6.2.2 KEPLER metrics verification through elaborate tests, possibly using measuring hardware . . . 50
6.2.3 Kubernetes Cluster Energy Efficiency Optimization . . . 50
6.3 Final Conclusion . . . 50

A KEPLER-provided power metrics **51**
A.1 Summary of KEPLER-Produced Metrics, according to KEPLER documentation[**KEPLERDocumentation**] . . . 51
A.1.1 Container-Level Metrics . . . 51
A.1.2 Node-Level Metrics . . . 53

Chapter 1

Introduction and Context

1.1 Significance of Energy Efficiency in Cloud Computing

Cloud computing has revolutionized how computing resources are shared and utilized, offering increased operational efficiency through resource sharing. While economic benefits such as reduced costs and improved scalability are often highlighted, energy efficiency and environmental impact are equally important. By maximizing shared resource utilization, cloud computing inherently reduces energy waste, supporting global sustainability goals.

The rapid adoption of cloud computing has transformed it into a dominant segment of global IT infrastructure. Hyperscalers like Amazon Web Services, Google Cloud, and Microsoft Azure contribute significantly to global energy consumption, attracting attention from policymakers. While cloud providers have incorporated renewable energy sources into their operations, this alone does not address the efficiency of energy utilization for workloads.

Technological advancements have improved the energy efficiency of data centers, with modern facilities achieving Power Usage Effectiveness (PUE) values near 1. However, PUE measures facility-level efficiency, not workload-level efficiency. Even with a perfect PUE of 1, significant energy waste can occur if computational resources are underutilized. This highlights the need to focus on workload-level energy efficiency.

Containers, as lightweight virtualization technology, improve resource density and energy efficiency compared to traditional virtual machines (VMs). Despite these advantages, containers introduce additional complexity, especially in measuring energy consumption. Accurate container-level energy consumption requires granular monitoring tools, a challenge compounded by Kubernetes' dynamic resource allocation and scaling mechanisms.

Despite the importance of energy efficiency in cloud computing, research on this topic remains limited. While data center operations have been optimized and green coding practices promoted, Kubernetes' energy efficiency is underexplored. Addressing this gap is crucial for balancing economic and environmental goals.

1.2 The Need for Energy-Efficient Kubernetes Clusters

Kubernetes has become the standard for container orchestration, managing containerized workloads at scale. However, its success brings new challenges, particularly in energy efficiency. Kubernetes environments are complex, featuring dynamic scaling, resource allocation, and workload distribution across multiple nodes. These features, while essential for performance, complicate energy measurement and optimization.

Commonly, Kubernetes clusters are hosted on VMs to simplify infrastructure management, adding another abstraction layer and further complicating energy measurement. To improve energy efficiency, robust methods for measuring energy consumption in Kubernetes environments are necessary. These methods must translate measurements into valuable insights to that will lay the foundation for optimization efforts.

Given the growing energy footprint of cloud computing and the limited research focus on this area, energy-efficient Kubernetes clusters represent a pressing research topic. By addressing this gap, this work contributes to the broader goal of sustainable cloud computing.

1.3 Objectives and Scope of this Thesis

1.3.1 Context

This thesis is part of the Master's program in Computer Science at the Zurich University of Applied Sciences (ZHAW) and represents the first of two specialization projects. The current project (VT1) focuses on the practical implementation of a test environment for energy efficiency research in Kubernetes clusters. The second project (VT2) will explore theoretical aspects and methodologies for measuring and improving energy efficiency.

This thesis builds upon prior works focused on performance optimization and energy measurement. EVA1 covered topics such as operating system tools, statistics, and eBPF, while EVA2 explored energy measurement in computer systems, covering hardware, firmware, and software aspects. These foundational topics provide the basis for the current thesis but will not be revisited in detail.

1.3.2 Scope

This thesis focuses on the practical implementation of a test environment, excluding detailed theoretical analysis and literature reviews. The primary goal is to document the creation of a reliable and reproducible test environment that supports future research on energy efficiency in Kubernetes clusters.

This thesis builds upon prior work conducted in EVA1 and EVA2, leveraging existing knowledge while extending the research focus. As a result, fundamental concepts from these works are assumed as background knowledge and are not reintroduced in detail.

The EVA1 presentations contribute essential principles related to Linux performance monitoring, system optimization, and a conceptual understanding of eBPF. While

these concepts play a role in this research, they are not re-explained, as the focus of this thesis lies in their practical application within an energy-efficient Kubernetes cluster.

Similarly, the EVA2 presentations provide the foundation for understanding energy consumption measurement at multiple levels, including hardware, firmware, and software. Key topics such as CPU states (ACPI, C-states, P-states), Intel CPU Performance Scaling Drivers, and Intel RAPL for power monitoring and control are considered essential but are not explicitly covered again in this thesis.

By building on these existing foundations, this thesis narrows its scope to investigate energy efficiency at the Kubernetes cluster level, incorporating relevant techniques from prior research where necessary.

1.3.3 Objectives

The main objective is to design and implement a test environment that facilitates:

- Analysis of key parameters affecting energy efficiency in Kubernetes clusters.
- Reliable and consistent experimentation.
- Reproducibility and automation in deployment and configuration.

The outcomes will provide the necessary infrastructure for subsequent research projects.

Parameters for Analysis

This project aims to reuse established tools and components where feasible. The parameters to be analyzed include:

- CPU utilization and energy consumption.
- Memory usage and its impact on power draw.
- Disk I/O and storage-related power consumption.

Additional parameters may be incorporated based on further evaluation.

Data Integrity and Persistence

Ensuring data integrity and persistence is critical for reliable analysis. Key requirements include:

- Persistent storage that survives system shutdown.
- A unified data store accessible by all nodes.
- Data retention across Kubernetes cluster reinstallations.
- The ability to power down unused worker nodes without data loss.

Reproducibility and Automation

Reproducibility and automation are bonus goals aimed at enhancing research efficiency. Benefits include:

- Simplified recovery from misconfiguration through rapid redeployment.
- Reduced troubleshooting time.
- Improved stack cleanliness by eliminating residual configurations.

Security

Security, while not a primary focus, will be addressed by implementing basic best practices. Key security measures include:

- Use of encrypted passwords.
- Adherence to basic Kubernetes security best practices.
- Minimization of potential vulnerabilities through careful configuration.

1.3.4 Use of AI Tools

During the writing of this thesis, *ChatGPT*[**OpenAI_ChatGPT_2024**] (Version 4, OpenAI, 2024) was used as an auxiliary tool to enhance efficiency in documentation and technical writing. Specifically, it assisted in:

- Structuring and improving documentation clarity.
- Refining descriptions of code and technical implementations.
- Beautifying and formatting smaller code snippets.
- Assisting in LaTeX syntax corrections and debugging.

All AI-generated content was critically reviewed, edited, and adapted to fit the specific context of this thesis. **ChatGPT was not used for literature research, conceptual development, methodology design, or analytical reasoning.** The core ideas, analysis, and implementation details were developed independently.

1.3.5 Project Repository

All code, configurations, and automation scripts developed for this thesis are publicly available in the PowerStack[**PowerStack**] repository on GitHub. The repository contains Ansible playbooks for automated deployment, Kubernetes configurations, monitoring stack setups, and benchmarking scripts. This allows for full reproducibility of the test environment and facilitates further research or adaptation for similar projects.

Chapter 2

Architecture and Design

2.1 Overview of Test Environment

The test environment consists of a Kubernetes cluster deployed on three bare-metal servers housed in a university datacenter. The three servers are identical in hardware specifications and connected through both a private network and the university network. The setup allows complete remote management and ensures direct communication between the servers for Kubernetes workloads. Below is a detailed description of the hardware and network topology. A diagram illustrating the architecture and network setup is provided in figure 2.1.

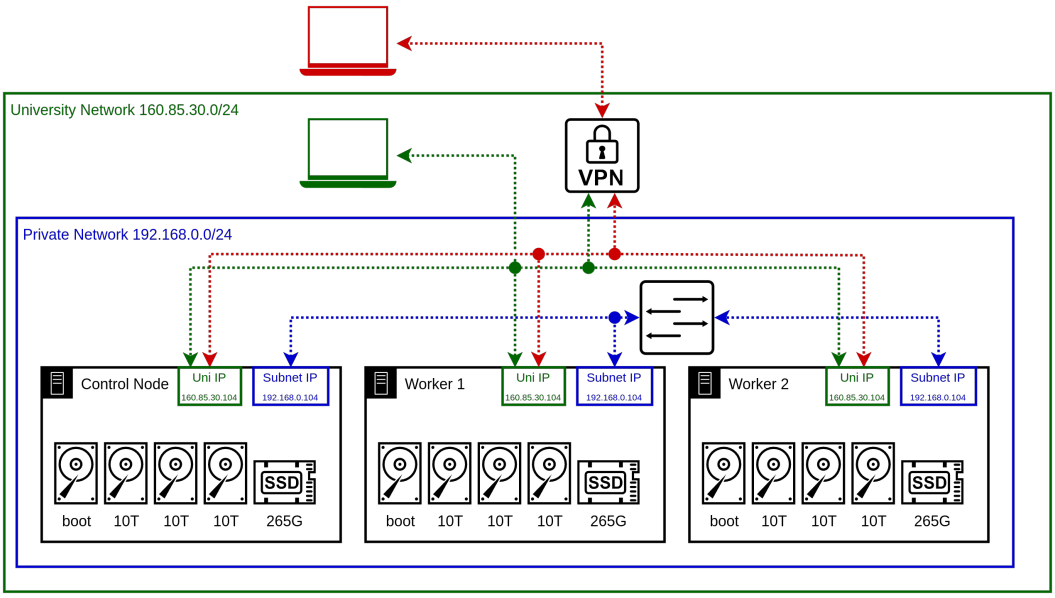


FIGURE 2.1: Physical Infrastructure Diagram

2.1.1 Hardware and Network

Bare-Metal Servers

The cluster is built using three identical Lenovo ThinkSystem SR530 servers, each equipped with the following hardware:

- CPU: 1x Intel(R) Xeon(R) Bronze 3104 @ 1.70GHz, 6 cores.

- Memory: 4x 16GB DDR4 DIMMs, totaling 64GB of RAM per server.
- Storage:
  - 2x 32GB M.2 SATA SSD for the operating system boot drive.
  - 1x 240GB 6Gbps SATA 2.5" SSD for persistent storage.
  - 3x 10TB 7.2K RPM 12Gbps SAS 3.5" HDDs for bulk storage.
- Power Supply: Dual redundant power supplies.
- Cooling: 4 out of 6 possible fans installed.
- Firmware:
  - BMC Version: 8.88 (Build ID: CDI3A4A)
  - UEFI Version: 3.42 (Build ID: TEE180J)
  - LXPM Version: 2.08 (Build ID: PDL142H)

The servers are equipped with Lenovo XClarity Controller (BMC) for remote management. Each server can be accessed via its BMC IP address for out-of-band management and monitoring.

Network Topology

The servers are connected using two distinct networks:

- **Private Network:** Each server has a private IP address (192.168.0.104–192.168.0.106), allowing direct, high-speed communication between nodes. This reduces the load on the university network and improves Kubernetes workload performance.
- **University Network:** Public-facing IP addresses (160.85.30.104–160.85.30.106) allow access within the university network, with external access enabled via VPN.

Note: Detailed switch and gateway configurations are managed by the university IT department and are beyond the scope of this document.

2.2 Key Technologies

2.2.1 Ubuntu

Ubuntu was chosen as the operating system for this project primarily due to the author's familiarity with it. Additionally, it was already installed on the servers when they were received, which saved time and reduced setup complexity. While there are other Linux distributions specifically designed for Kubernetes, using a familiar distribution ensured smoother initial configuration and operation.

2.2.2 Bare-Metal K3s

Installing Kubernetes directly on bare-metal servers (without using a hypervisor or virtual machines) was a fundamental decision to ensure direct access to hardware-level data. This approach allows Kubernetes to interact with the underlying hardware more effectively, which is critical for accurate energy consumption monitoring.

K3s was chosen for several reasons:

- It is lightweight, making it suitable even for weaker servers, while potentially also lowering energy consumption.
- Despite its lightweight nature, it remains fully compatible with stock Kubernetes, ensuring that standard Kubernetes resources and configurations can be used without modification.
- K3s is optimized for ARM architectures, making it ideal for deployment on devices like Raspberry Pis in a homelab environment.
- The author had prior experience with K3s and Rancher, which contributed to a faster and smoother deployment.

2.2.3 Ansible, Helm, Kubectl

For automation, Ansible and Helm were selected. Helm and Kubectl were an obvious choice due to their widespread use in Kubernetes for managing and deploying applications.

Ansible was chosen for its flexibility and ease of use in managing server configurations and automating repetitive tasks across multiple nodes. Additionally, Ansible's agentless architecture simplifies the management of bare-metal servers by requiring only SSH access and Python installed on the target machines.

2.2.4 Kube-Prometheus Stack

The Kube-Prometheus stack was chosen because it is the de-facto standard for monitoring in Kubernetes environments. This project has reached a high level of maturity, offering robust features and a wide range of integrations. Installation and configuration using Helm are straightforward, and the abundance of available resources makes troubleshooting easier.

Prometheus

Prometheus was selected as the primary monitoring tool because it is the standard in the Kubernetes ecosystem. Despite its advantages, Prometheus has some downsides: it can introduce significant overhead, and it is not suitable for monitoring low-second or sub-second intervals due to typical scrape intervals being longer. However, for container orchestration, where longer container lifetimes are expected, this limitation is acceptable.

Grafana

Grafana was chosen for its ability to provide excellent, customizable visualizations of metrics collected by Prometheus. It enables easy interpretation of complex data through dashboards and visual aids, making it a valuable addition to the monitoring stack.

AlertManager

AlertManager is included in the Kube-Prometheus stack and is used to handle alerts generated by Prometheus. While it was not utilized in this project, its inclusion is welcomed for potential future use in managing alerts and notifications in a production environment.

2.2.5 KEPLER

Purpose of KEPLER

KEPLER, or *Kubernetes-based Efficient Power Level Exporter*, is a promising project focused on measuring energy consumption in Kubernetes environments. It provides detailed power consumption metrics at the process, container, and pod levels, addressing the growing need for energy-efficient cloud computing.

With cloud providers and enterprises under increasing pressure to improve energy efficiency, KEPLER offers a practical solution. By enabling detailed real-time measurement of power usage, it bridges the gap between high-level infrastructure metrics and workload-specific energy consumption data. This capability makes KEPLER a valuable tool in advancing energy-efficient Kubernetes clusters.

Limitations of KEPLER

Despite its potential, KEPLER has some limitations in the context of this project:

- Active Development: KEPLER is still in active development, meaning its features and APIs may change over time. Additionally, the documentation is currently limited, and there are few community resources available for troubleshooting.
- Complexity: As a large and complex project, adapting KEPLER beyond basic configuration requires a deep understanding of its architecture. Implementing custom changes or enhancements can be challenging without significant expertise.

While KEPLER may not be perfect, it is currently the most promising approach to addressing the challenge of measuring energy consumption in Kubernetes environments. Consequently, a large focus of this thesis will be on evaluating KEPLER's capabilities and identifying areas for improvement.

2.3 Architecture and Design

2.3.1 Kubernetes Cluster Design

The Kubernetes cluster is deployed on three bare-metal servers running Ubuntu. One server is designated as the control plane, while the other two serve as worker nodes. This setup avoids high availability (HA) for simplicity, given the scope of this project. The servers are connected via their internal IP addresses, enabling direct communication without routing through external networks. All Kubernetes components, including the API server, controller manager, and scheduler, run exclusively on the control plane node, while workloads are distributed across all nodes by Kubernetes. Figure 2.1 provides an overview of the system architecture, including components and data flow.

2.3.2 Persistent Storage

Persistent storage is provided using the spare SSD disk on the control node. A partition on the disk is created, formatted with the BTRFS file system, and mounted. The NFS server is installed on the control node, and NFS clients are installed on the worker nodes, enabling them to access the shared storage. This centralized approach was chosen because the control node is the only server guaranteed to remain powered on throughout the experiment, ruling out the need for a distributed storage solution like CEPH.

Within the NFS share, separate directories are created for Prometheus and Grafana data. Persistent volumes (PVs) are defined in Kubernetes, and persistent volume claims (PVCs) are created for each service. The size of these PVs can be configured during installation, allowing flexibility for future storage needs.

2.3.3 Monitoring Architecture

The monitoring stack is deployed using the kube-prometheus-stack Helm chart. This stack includes Prometheus, Grafana, and AlertManager, providing a complete solution for monitoring, visualizing, and managing alerts in Kubernetes. Prometheus is configured to scrape metrics from KEPLER and Kubernetes endpoints (such as the kubelet API) at regular intervals. Grafana connects to Prometheus, enabling real-time visualization of metrics through customizable dashboards.

2.3.4 Metrics Collection and Storage

KEPLER generates metrics by collecting data from various sources:

- **Hardware-level metrics:** Using eBPF and kernel tracepoints to gather low-level data such as CPU cycles and cache misses.
- **Power-related metrics:** Collected via RAPL (Running Average Power Limit) and IPMI (Intelligent Platform Management Interface) to monitor CPU and platform energy consumption.
- **Container-level metrics:** Retrieved from the Kubernetes kubelet API, which provides cgroup resource usage data for running containers and pods.

KEPLER aggregates this data, calculates power consumption metrics, and exposes them in a Prometheus-friendly format. Prometheus scrapes these metrics at a configurable interval, storing them as time series data on the persistent volume. The time series format allows Prometheus to track changes over time, enabling detailed analysis of resource usage patterns. In chapter 2.4, the KEPLER architecture is briefly explained with a focus on metrics collection and generation. The information flow is pictured in diagram 2.2.

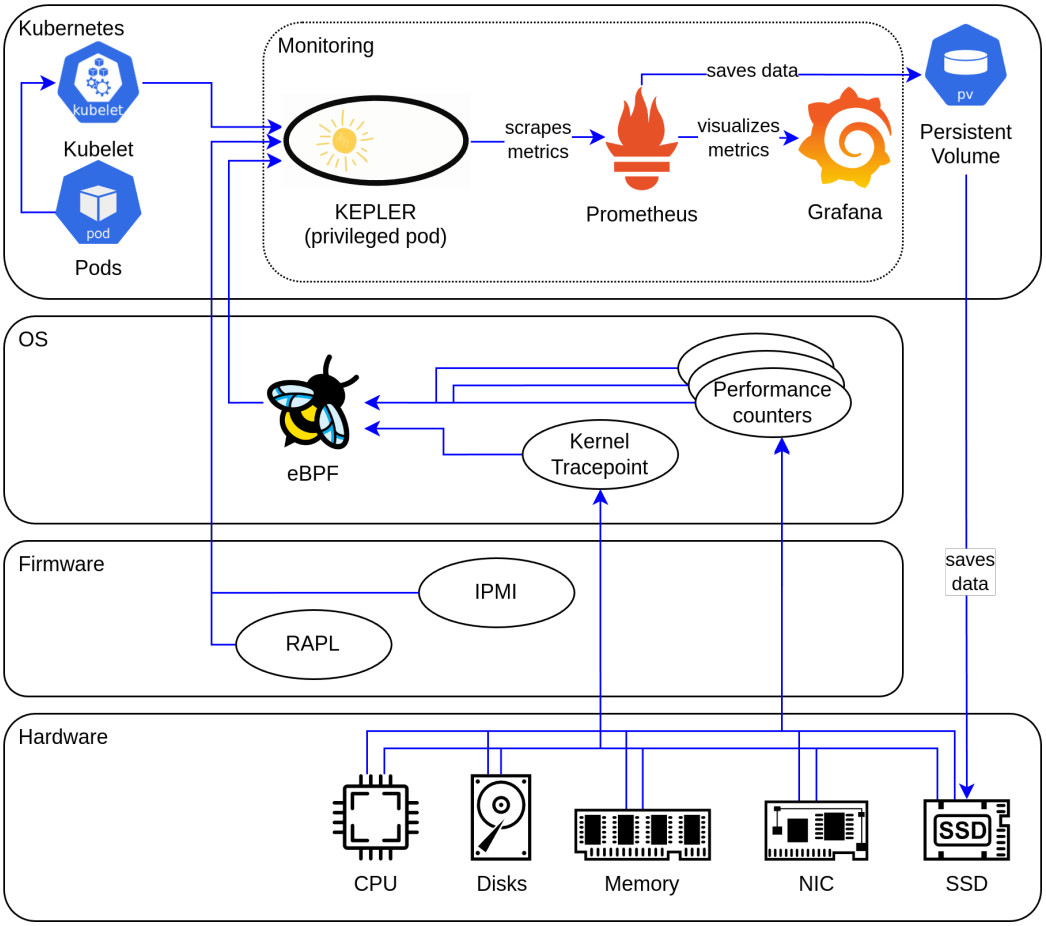


FIGURE 2.2: Monitoring data flow diagram of the entire stack

Since the KEPLER project has already done a significant amount of research, as well as practical implementation, the decision was made to utilize KEPLER as a core component in this project.

2.3.5 Repository Structure

The repository for this project is designed to include all aspects of the Kubernetes-based energy efficiency test environment, from deployment automation to documentation. Given the reliance on various external projects, a hybrid approach was adopted for managing dependencies:

Submodules for External Repositories

Several external projects with frequent updates were forked and included as submodules in the repository. This approach allows easy configuration and customization while maintaining the ability to sync changes from upstream repositories. Additionally, by freezing submodules at specific commits, the project is protected from unexpected upstream changes that could introduce instability.

Direct Deployment from External Repositories

For other external projects that require minimal customization, direct deployment from their original repositories was chosen. This reduces the complexity of repository maintenance and ensures that stable, tested versions are always used.

Structure Overview

The repository is organized to maintain clarity and separation of concerns:

- **ansible/:** Ansible playbooks and roles for automated deployments.
- **helm/:** Custom or external Helm charts managed through Ansible.
- **scripts/:** Bash scripts for executing Ansible playbooks.
- **config/:** Centralized configuration file and Ansible vault.
- **docs/:** Documentation files containing setup and usage of the project.
- **thesis/:** Contains all files related to the thesis, written in LaTeX.

2.3.6 Automation Architecture

Automation was a key focus in this project to ensure reproducibility, consistency, and ease of deployment. The automation architecture is primarily based on Ansible, with Helm nested into Ansible playbooks for Kubernetes-specific deployments.

Ansible and Helm Integration

Ansible was used for automating the setup of the base environment, including system-level configurations and Kubernetes deployments. All Helm installations, such as the kube-prometheus-stack, were wrapped in Ansible playbooks. This approach provided a unified automation framework where both system configurations and Kubernetes resources could be managed together. This also allowed for clear version control and logging of every deployment step.

Execution Scripts

Custom Bash scripts were written to handle the execution of Ansible playbooks. Apart from convenience, these scripts ensured:

- Correct execution context and configuration for playbook execution.
- Automatic log creation aiding in troubleshooting and auditing.

Centralized Configuration

All configuration values, such as IP addresses, storage paths, and deployment options, were centralized in a single configuration file. This design simplifies re-deployment on different hardware by only requiring changes in one location. When necessary, Jinja templates were used in Ansible to dynamically adapt configurations based on this central file.

Security

Sensitive information, such as passwords and API keys, was encrypted using an Ansible Vault. This ensured that confidential data could be securely managed within the repository without compromising security during deployment.

2.4 KEPLER Architecture and Metrics Collection

Since KEPLER plays a central role in this project, it is important to understand its architecture and how it collects metrics. This section provides a brief overview of KEPLER's components and its data collection methods. For more detailed information, the official KEPLER documentation[**KEPLERDocumentation**] should be consulted.

2.4.1 KEPLER Components

KEPLER Exporter

The core component of KEPLER is the Exporter, which runs as a privileged daemonset pod on each node in the Kubernetes cluster. This exporter directly interacts with the hardware and kernel, collecting energy consumption and resource utilization metrics. It estimates power usage at the process, container, and pod levels, exposing the collected metrics in a Prometheus-friendly format.

A service monitor is also deployed, allowing Prometheus to scrape metrics from the KEPLER exporter endpoints.

KEPLER Model Server

Although the KEPLER Model Server is not used in this project, it is worth noting its purpose. The model server provides power estimation models based on available metrics, supporting different granularities such as node, pod, or processor component levels. It can also include an online trainer to update models dynamically during runtime.

2.4.2 KEPLER Data Collection

Process and Container Data

KEPLER employs eBPF to collect detailed CPU event data. eBPF programs run in a privileged context, enabling efficient, low-overhead monitoring of kernel-level events. Specifically, KEPLER hooks into the `finish_task_switch` kernel function, which handles task context switching, to collect process-level metrics, specifically the following Perf counters:

- `PERF_COUNT_HW_CPU_CYCLES`

- `PERF_COUNT_HW_REF_CPU_CYCLES`,
- `PERF_COUNT_HW_INSTRUCTIONS`
- `PERF_COUNT_HW_CACHE_MISSES`

By maintaining a BPF hash of process IDs, CPU IDs, and context switch timestamps, KEPLER correlates resource usage to individual processes and containers. This data is essential for deriving energy consumption estimates. The hash is shown in table 2.1.

TABLE 2.1: Hardware CPU events monitored by KEPLER

Key	Value	Description
pid	pid	Process ID
	cgroupid	Process CGroupID
	process_run_time	Total time a process occupies CPU (calculated each time process leaves CPU on context switch)
	cpu_cycles	Total CPU cycles consumed by process
	cpu_instr	Total CPU instructions consumed by process
	cache_miss	Total Cache miss by process
	page_cache_hit	Total hit of the page cache
	vec_nr	Total number of soft irq handles by process (max 10)
	comm	Process name (max length 16)

CPU Power Data

KEPLER leverages Intel RAPL (Running Average Power Limit) to monitor energy consumption across various CPU domains, including cores, DRAM, and integrated GPUs. RAPL provides real-time power consumption data with fine granularity and high sampling rates, allowing KEPLER to measure energy usage accurately. The supported power domains include:

- **Package (PKG):** Total energy consumption of the CPU socket, including cores, caches, and memory controllers.
- **Power Plane 0 (PP0):** Energy consumption of CPU cores.
- **Power Plane 1 (PP1):** Energy consumption of integrated GPUs (if present).
- **DRAM:** Energy consumption of memory attached to the CPU.

KEPLER uses the following methods to access RAPL data (in order of preference):

1. **RAPL Sysfs:** Direct access to energy counters via the Linux power capping framework located in `/sysfs`. This requires root access to the powercap driver and is the method used in this project.
2. **RAPL MSR:** Direct access through Model-Specific Registers (MSRs), providing detailed energy readings.
3. **Kernel Driver xgene-hwmon:** Used in specific ARM architectures.

Platform Power Information

KEPLER can also collect platform-level power consumption data, representing the total power usage of the node. This is achieved through:

- **ACPI (Advanced Configuration and Power Interface):** Used to access system-level power information.
- **IPMI (Intelligent Platform Management Interface):** Provides remote access to power data via the Baseboard Management Controller (BMC).

2.4.3 KEPLER Power Model

KEPLER uses a combination of two power modeling approaches, choosing a suitable approach based on available data: If total power is known, KEPLER uses a power ration modelling to compute finer-grained power figures for individual components at the node and container level. If detailed hardware-level power metrics are unavailable, such as in virtualized environments, KEPLER estimates power consumption based on system utilization metrics using a pretrained model (currently based on a Intel Xeon E5-2667 v3-processor). Since this modeling is inherintly flawed for any other processor, a goal of the project is to offer a higher number of models with other architectures.

In previous experiments conducted by the author, KEPLER was deployed on a Kubernetes cluster with virtualized nodes in an Openstack environment. Since no hardware-level power information was available, KEPLER attempted to estimate power consumption solely based on system metrics. The results were inconsistent and unreliable, highlighting the importance of accurate hardware data for meaningful power consumption analysis.

2.4.4 Metrics produced by KEPLER

KEPLER collects and exports a wide range of metrics related to energy consumption and resource utilization. The full list and description of metrics is provided in Appendix A, but they are summarized here.

Container-level metrics

On a container level, KEPLER estimates the total energy consumption in joules. The consumption is broken down to the components Core, DRAM, 'Uncore' (such as fast-level cache and memory controllers), the entire CPU package, GPU and other. Additionally, several resource utilization metrics are calculated, namely the total CPU time, cycles, instructions and cache misses. Some IRQ metrics are provided, namely the total number of transmitted and received network packets, and the number of block I/O operations.

Node-level metrics

On a node level, total energy consumption is once again estimated in joules. Estimations are provided for the entire node, as well as the Core, DRAM, uncore, CPU package, GPU, platform and other. Additionally, Node-specific metadata (such as the CPU archtecure), aggregated metrics (used by the KEPLER model server), and Intel QAT utilization are provided.

Chapter 3

Implementation

This chapter describes the implementation and configuration of the various components used in this project. All automation scripts are designed to be idempotent. All scripts can be executed with shell scripts in the `Powerstack/scripts`-directory. Generally, all configuration is to be done in the central configuration file (`/Powerstack/configs/inventory.yml`) unless otherwise stated. Sensitive information is to be defined in the ansible-vault file (`/Powerstack/configs/vault.yml`). To keep this chapter brief, instructions for verification are written in Appendix **??**

3.1 K3s Installation

This section describes the steps involved in setting up a Kubernetes cluster using K3s on bare-metal servers. The installation was automated using an Ansible playbook forked from the official k3s-io/k3s-ansible **[k3s-ansible]** repository, with necessary customizations for internal IP-based communication.

3.1.1 Preparing the Nodes

Before running the Ansible playbook, the following prerequisites need to be in place on all servers:

- **Operating System:** Ubuntu 22.04 (used kernel version 5.15.0)
- **Passwordless SSH:** Passwordless SSH access must be configured for a user with sudo privileges on all servers.
- **Networking:** Each server should have both an internal IP (for cluster communication) and an external IP (for access via VPN or external management).
- Local Ansible Control Node Setup:
  - **Ansible-community** 9.2.0 (must be 8.0+).
  - **Python** 3.12.3 and Jinja 3.1.2 installed as dependencies.
  - **kubectl** 1.31.3

3.1.2 K3s Installation with Ansible

The playbook supports x64, arm64, and armhf architectures. For this project, it was tested on x64 architecture only.

Configuration Details

- Internal and external IP addresses of all servers must be specified.
- One server must be designated as the control node.
- Default configurations such as ansible-port, ansible-user, and k3s-version can be changed if needed.

Kubectl Configuration

- The playbook automatically sets up kubectl for the user on the Ansible control node by copying the Kubernetes config file from the control node to the local machine.
- The user must rename the config file from 'config-new' to 'config' and set the context to powerstack using the following command:
  `kubectl config use-context powerstack`

3.2 NFS Installation and Setup

3.2.1 NFS Installation with Ansible

Setting up the NFS server and clients was fully automated using an Ansible playbook. Before beginning the automated setup, the following manual step must be completed:

- **Disk Selection:** A suitable disk must be chosen on the control node to act as persistent storage. It is important to note that this disk will be reformatted, and all existing data will be lost.

The Ansible playbook performs the following actions:

- **Disk Preparation:** The selected disk is partitioned (if necessary) and formatted with a single Btrfs partition. The entire disk space is allocated to this partition. The partition is then mounted to `/mnt/data`, and an entry is added to `/etc/fstab` to ensure persistence across reboots.
- **NFS Server Setup:** The `nfs-kernel-server` package is installed and configured on the control node. The directory `/mnt/data` is exported as an NFS share, accessible to the worker nodes.
- **NFS Client Setup:** On each worker node, the `nfs-common` package is installed. The NFS share is mounted, and an `/etc/fstab` entry is created to ensure persistence across reboots.

Configuration Details

- The nfs network must be specified, and the control and worker nodes must be in that network
- The export path must be specified

3.3 Rancher Installation and Setup

3.3.1 Rancher Installation with Ansible and Helm

Although not strictly necessary for the project, Rancher was deployed in the `cattle-system` namespace to assist with debugging and system analysis. The installation was automated using an Ansible playbook, which integrates Helm for deploying Rancher and its dependencies. The key steps are as follows:

- **Helm Installation:** Helm was installed on the control node to facilitate the deployment of Rancher and its dependencies.
- **Namespace Creation:** The `cattle-system` namespace was created to host the Rancher deployment.
- **Cert-Manager Deployment:** Cert-Manager, a prerequisite for Rancher, was installed to manage TLS certificates.
- **Rancher Deployment:** Rancher was installed using the official Helm chart. During installation, the following parameters were configured:
  - **Hostname:** A Rancher hostname was defined to enable access.
  - The Helm chart was configured with the `-set tls=external` option to enable external access to Rancher.
  - **Bootstrap Password:** A secure bootstrap password was set for the default Rancher administrator account.
- **Ingress Configuration:** An ingress resource was configured to route traffic to Rancher, allowing access through the defined hostname.

3.4 Monitoring Stack Installation and Setup with Ansible

The monitoring stack, comprising Prometheus, Grafana, and AlertManager, was deployed using the kube-prometheus-stack[**prometheus_helm_charts**]-Helm chart from the `prometheus-community/helm-charts` repository. While the repository was forked for convenience, no changes were made to the upstream chart, ensuring compatibility with future updates.

3.4.1 Prometheus and Grafana Installation with Ansible and Helm

The installation process was automated using Ansible roles, ensuring idempotency and centralization of configurations. The following key steps were executed:

- **Persistent Storage Configuration:**
  - Directories for Prometheus, Grafana, and AlertManager were created on the NFS-mounted disk.
  - A custom `StorageClass` was defined for the NFS storage. The default storage `StorageClass` local-path was overridden to be non-default.
  - PersistentVolumes (PVs) were created for Prometheus, Grafana, and AlertManager. A PersistentVolumeClaim (PVC) was explicitly created for Grafana, while PVCs for Prometheus and AlertManager were managed by the Helm chart.
- **Helm Chart Installation:**
  - A Helm values file was generated dynamically using a Jinja template. This template incorporated variables from the central Ansible configuration file to ensure consistency. Sensitive information, such as the Grafana admin password, was included in the values file. To mitigate potential security risks, the values file was removed from the control node after installation.
  - The Helm chart was installed using an Ansible playbook. The following customizations were applied via the generated values file:
    * PVC sizes for Prometheus and AlertManager were set based on the central configuration.
    * A Grafana admin password was defined.
    * Prometheus scrape configurations were adjusted to include the KEPLER endpoints.
    * Changes to the `securityContext` were made to allow Prometheus to scrape KEPLER metrics.
- **Service Port Forwarding:**
  - Prometheus, Grafana, and AlertManager services were exposed using static `NodePort`s defined in the central configuration file, enabling external access.
- **Cleanup:**
  - A cleanup playbook was executed to remove sensitive configuration files from both the control node and the Ansible control node.

3.4.2 Removal Playbook

An Ansible playbook was created to handle the complete uninstallation of the monitoring stack. This was necessary to ensure that PVs and PVCs were explicitly removed to avoid residual artifacts in the Kubernetes cluster.

3.5 KEPLER Installation and Setup with Ansible and Helm

3.5.1 Preparing the Environment

The KEPLER deployment uses the official KEPLER Helm chart repository. Before deploying KEPLER, several prerequisites must be addressed to ensure proper functionality.

Redfish Interface

The Redfish Scalable Platforms Management API is a RESTful API specification for out-of-band systems management. On the Lenovo servers used in this project, Redfish exposes IPMI power metrics, which KEPLER accesses through its Redfish interface. To verify Redfish functionality, navigate to the Lenovo XClarity Controller and ensure the following setting is enabled:

- **IPMI over LAN:** This option can be found under `Network -> Service Enablement and Port Assignment`.

The Redfish API can be tested by visiting the following endpoints in a web browser:

- General Redfish information: `https://<BMC-IP>/redfish/v1`
- Power metrics: `https://<BMC-IP>/redfish/v1/Chassis/1/Power#\PowerControl`

Kernel Configuration

KEPLER requires kernel-level access for eBPF tracing, which involves setting a tracepoint using the `syscall perf_event_open`. By default, this syscall is restricted. To allow KEPLER to function properly, an Ansible role is used to modify the kernel parameter `perf_event_paranoid` via `sysctl` without requiring a reboot.

The restriction level can be verified by checking the value of `/proc/sys/kernel/perf_event_paranoi`. For this project, all restrictions were removed by setting the value to `-1`.

3.5.2 KEPLER Deployment with Ansible and Helm

KEPLER was deployed using the KEPLER Helm chart[**kepler_helm_chart**] from the `sustainable-computing-io/kepler-helm-chart` repository, with Ansible automating the configuration and deployment process. The deployment configuration was centralized in a Jinja template, which was rendered locally and copied to the control node before applying it.

Key configurations in the Helm values file include:

- **Enabled metrics:** Various metric sources are enabled for detailed energy monitoring.
- **Service port:** The KEPLER service port is defined for Prometheus to scrape metrics.
- **Service interval:** The KEPLER service interval is set to 10 seconds.

- **Redfish metrics:** Redfish/IPMI metrics are enabled, and Redfish API credentials are provided. The Redfish credentials are the same as those used for the Lenovo XClarity Controller interface. Note that the BMC IP address differs from the node IP address.

3.5.3 Verifying KEPLER Metrics

After deployment, it was essential to verify that KEPLER was correctly collecting and exposing metrics. Verification involves the following steps:

Prometheus scraping

After deploymnet, successful Prometheus scaping of the KEPLER endpoints can be verified using the Prometheus web interface.

Metric availability

All KEPLER metrics were checked individually in the Promehteus web interace to ensure that non-zero values were being written. Each metric would disclose a single metric, which was a welcome additional confirmation that the listed data source was being monitored correctly.

Kepler logs

The KEPLER logs were inspected for insight into used data sources:

LISTING 3.1: `kepler.log`

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:103] Kepler running on version: v0.7.12-dirty
1 config.go:293] using gCgroup ID in the BPF program: true
1 config.go:295] kernel version: 5.15
1 config.go:322] The Idle power will be exposed. Are you running on Baremetal or using single VM per node?
1 power.go:59] use sysfs to obtain power
1 node_cred.go:46] use csv file to obtain node credential
1 power.go:79] using redfish to obtain power
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:84] Number of CPUs: 6
1 watcher.go:83] Using in cluster k8s config
1 reflector.go:351] Caches populated for *v1.Pod from pkg/kubernetes/watcher.go:211
1 watcher.go:229] k8s APIserver watcher was started
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_TOTAL Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms]
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_COMPONENTS Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms bpf_cpu_time_ms bpf_cpu_time_ms   gpu_compute_util]
1 node_component_energy.go:62] Skipping creation of Node Component Power Model since the system collection is
    supported
1 prometheus_collector.go:90] Registered Process Prometheus metrics
1 prometheus_collector.go:95] Registered Container Prometheus metrics
1 prometheus_collector.go:100] Registered VM Prometheus metrics
1 prometheus_collector.go:104] Registered Node Prometheus metrics
1 exporter.go:194] starting to listen on 0.0.0.0:9102
1 exporter.go:208] Started Kepler in 2.2651724s
```

Power metrics from ACPI / IMPI

In the event that both ACPI and IPMI were configured to measure platform power, KEPLER prefered to use IPMI as its primary data source, meaning that it would use the IPMI overall energy consumption, and calculate lower-level energy consumption using ACPI. This is to be expected, since IPMI allows to measure power at a higher level than ACPI, and can get detailed power data for the entire plaform. In the absence of IPMI data, KEPLER uses ACPI as the only power data source.

Redfish issues

KEPLER was sporadically unable to correcly handle individual redfish data values. These incedents were sparse for different data values. Unfortunately, the source of this issue could not be eliminated in the context of this thesis. The log below shows an instance of the following error:

```
Failed to get power: json: cannot unmarshal number 3.07 into Go struct
    field Voltages.Voltages.ReadingVolts of type int
```

Error Message cpu0/online

The following error message is notable:

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/
    online: no such file or directory
```

The missing file warning can be attributed to the fact that the Intel Xeon used in this project does not support Core Offlining, i.e. the dynamic disableling of individual CPU Cores at runtime. While Core Offlining is an interesting feature for energy-efficiency ananlysis, this can be acccepted as a hardware limitation of this project.

Chapter 4

Test Procedure

This chapter describes the test procedure used to verify KEPLER-produced metrics. The verification process involves executing dynamic workloads on a Kubernetes cluster and analyzing the correlation between the workload and the KEPLER-reported metrics. For better intuitive understanding, all Joule-based metrics are converted to Watts, and all operations-based metrics are converted to IOPS.

The collected data is visualized through diagrams for easier interpretation. However, this thesis does not provide a detailed energy efficiency analysis. Instead, the goal is to verify whether KEPLER metrics reliably correlate with workload fluctuations, ensuring its suitability for a more in-depth energy efficiency study in future work.

4.1 Test Setup

All test workloads were Ansible-created within a dedicated Kubernetes namespace, referred to as the *testing-namespace*. While the cluster itself was designed to be largely hardware-independent, the test setup requires manual adjustments when deployed on different hardware. Specifically, CPU and memory allocations for test pods should be reviewed, and the storage disk used for disk I/O experiments must be empty and correctly identified.

4.1.1 Benchmarking Pod

A dedicated Ubuntu-based *benchmarking pod* was provisioned (using Ansible) to serve as the central test agent for all experiments. This pod would allow completely self-contained testing in the cluster, without external traffic or - more importantly - an external machine available for longer tests. The benchmarking pod is configured with a fully functional `kubectl` setup, an OpenSSH client for authentication, and essential tools such as `wget`, `curl`, `vim` and `git`.

4.1.2 Testing Pods

Test workloads were deployed as DaemonSets to ensure that every node in the cluster hosted the required test pods. Depending on the experiment, different ressource allocations were made:

- For the CPU stress testing, a test and a load pod were deployed with 2.5 vCPU, 1 GB memory each.

- For the Memory stress testing, a test and a load pod were deployed with 150m vCPU, 25 GB memory each.
- For Network I/O and disk I/O stress testing, a test pod was deployed with 2.5 vCPU, 20 GB memory.

The resources were allocated to ensure an even split between CPU *testing* and *background load pods*, while memory was similarly distributed between memory-intensive workloads. A resource margin was maintained to prevent system instability. Benchmarking tools were installed on each pod, including `stress-ng`[**stress-ng**] for CPU and memory stress tests, `fio`[**fio**] for disk I/O testing, and `iperf3`[**iperf3**] for network performance measurement .

4.1.3 Disk Formatting and Mounting

For disk I/O experiments, an unused HDD on each worker node was partitioned, formatted, and mounted using Ansible. To ensure persistence, the mounting process was configured with an entry in `/etc/fstab`.

4.2 Test Procedure

Since energy consumption is not calculated beyond the node level, all tests were conducted on a single worker node. The test pod (either high-CPU or high-memory) generated workloads at predefined levels of 10%, 30%, 50%, 70%, and 90% for a fixed duration of 30 minutes per workload level. For CPU and meory testing, Tests were executed under two conditions: an idle cluster and a busy cluster, where the busy cluster was simulated by running background load pods at 90% utilization. Since disk and network usage are not by default restricted in Kubernetes, this distinction as not made for Disk I/O and Network I/O tests.

4.2.1 CPU Stress Test

CPU-intensive workloads were generated using `stress-ng`, with a CPU worker initiated on each available core. This test was executed under both idle and busy cluster conditions.

4.2.2 Memory Stress Test

Memory-intensive workloads were generated using `stress-ng`, where a single virtual memory worker allocated all available memory. The test was executed under both idle and busy cluster conditions.

4.2.3 Disk I/O Stress Test

Disk performance was assessed using `fio`. The experiment consisted of two phases: measuring maximum IOPS through read operations on the mounted HDD, followed by read operations at predefined percentages of the maximum IOPS. To eliminate caching effects, random read operations were used exclusively, and direct I/O was enabled.

4.2.4 Network I/O Stress Test

Network performance was evaluated using `iperf3`. First, the maximum bandwidth between pods on different nodes was measured. Next, controlled tests were conducted at various percentages of the maximum bandwidth. To mitigate server-side overhead, only client-side results were analyzed.

4.3 Data Analysis

Data collected from each experiment was analyzed in two main steps using Python.

4.3.1 Data Querying

Prometheus was queried to extract KEPLER metrics for the duration of each experiment. The retrieved data was formatted and stored as CSV files. The analysis relied on the python libraries `pandas`, `requests` and `datetime` for data querying and processing.

4.3.2 Diagrams

Visualization of KEPLER metric data was performed using `matplotlib`. Each diagram featured:

- X-axis: Time
- Primary Y-axis: KEPLER metric values (Watts or operations per second)
- Secondary Y-axis: Test workload percentage
- A moving average overlay to improve readability

By correlating workload levels with KEPLER metrics, the structured analysis validated the suitability of KEPLER for future energy efficiency studies.

Chapter 5

Test Results

This chapter presents the results of the test procedures conducted to analyze KEPLER-produced metrics. Each section corresponds to a specific resource type—CPU, Memory, Disk I/O, and Network I/O—with further division into container-level and node-level metrics. The results are discussed alongside their respective figures, which illustrate KEPLER-deduced energy consumption and performance trends.

It is important to note that all KEPLER metrics exhibit high oscillations. A detailed analysis reveals that these oscillations follow a highly regular pattern, suggesting an issue with the metric publication intervals in KEPLER or the scraping intervals of Prometheus. The data has been analyzed as-is, with moving averages provided to improve readability. The implications of this irregularity will be discussed further in chapter 6.

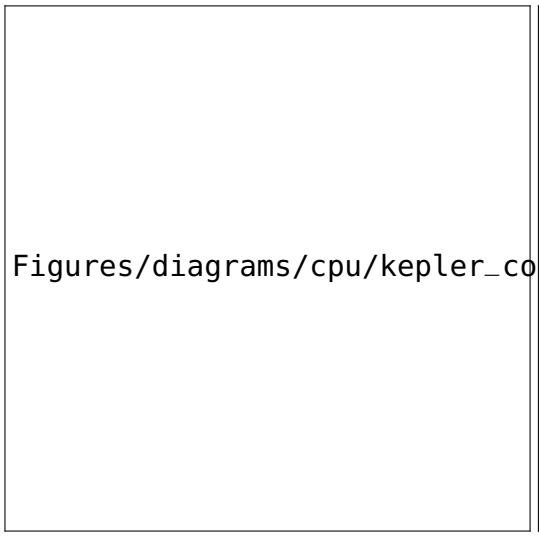
For clarity and to avoid confusion, two KEPLER metric concepts should be reiterated before discussing the results:

- **Package Metrics:** These metrics represent the entire CPU package, including all cores and uncore components.
- **Platform Metrics:** These metrics represent the entire node.
- ***Other* Metrics:** These metrics capture the entire platform except the CPU package and DRAM.

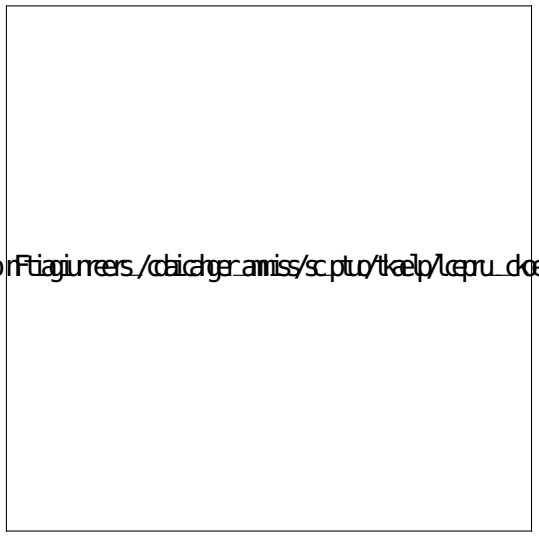
5.1 CPU Stress Test Results

5.1.1 Container-Level Metrics During a CPU Stress Test

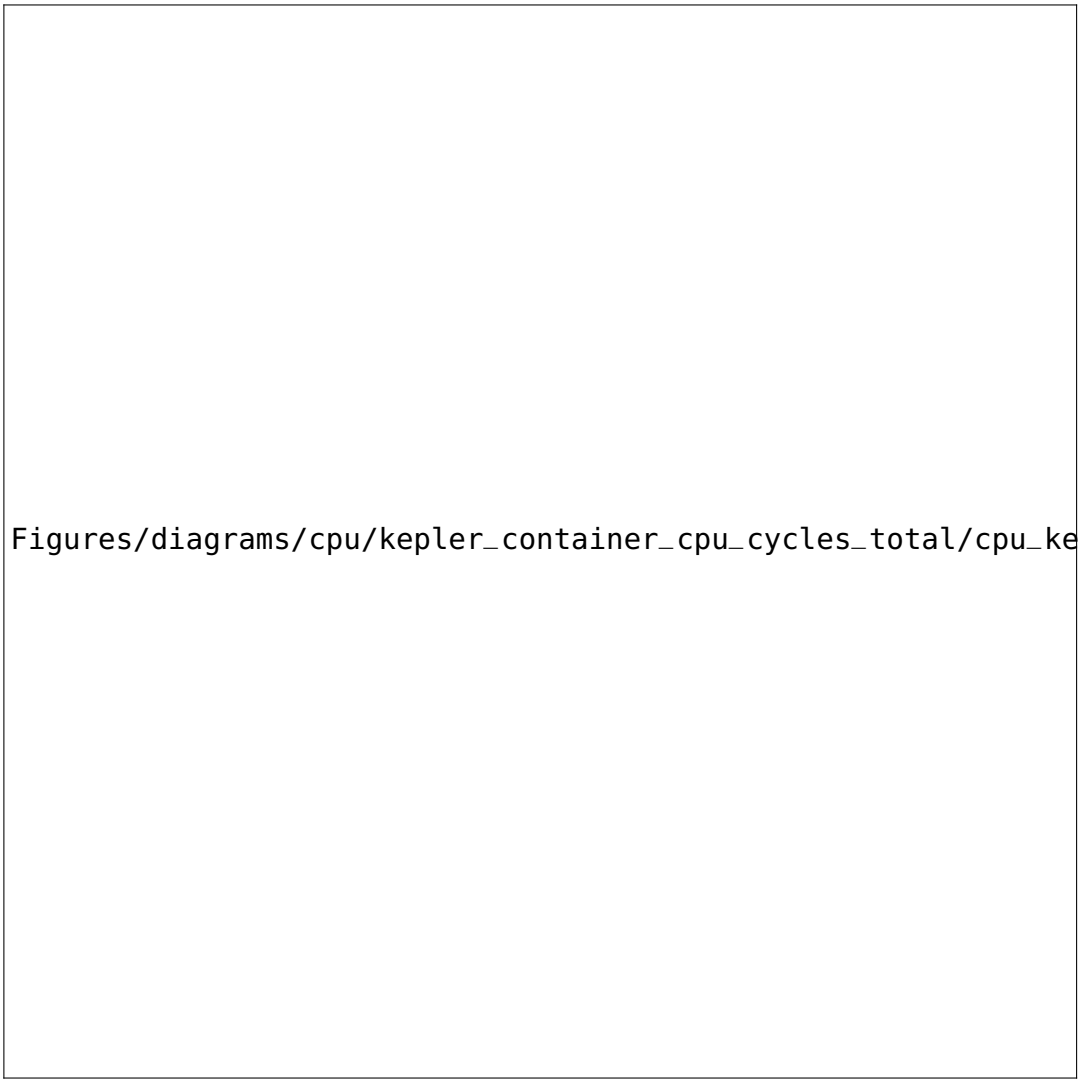
A set of figures illustrating cache misses, CPU cycles, and CPU instructions during testing is provided in Figures 5.7a to 5.1c.



(A) Cache misses



(B) CPU instructions



(C) CPU cycles

FIGURE 5.1: KEPLER container-level CPU metrics during a CPU stress test

These figures illustrate cache misses, CPU cycles, and CPU instructions of the test container during test execution. Notably, the diagrams display uniform trends, as the three metrics directly mirror the work generated by stress-ng, which is expected to be consistent and stable. The strong correlation between the applied workload and cache misses, CPU cycles, and CPU instructions confirms the correct execution of the test.

Since the CPU workload running on the rest of the cluster should not affect the test container's workload, the metric values under 'idle' and 'busy' cluster conditions should remain the same. This is indeed the case, verifying that the testing procedure was conducted correctly.

A figure illustrating KEPLER's deduced package energy consumption is provided in Figure 5.2.

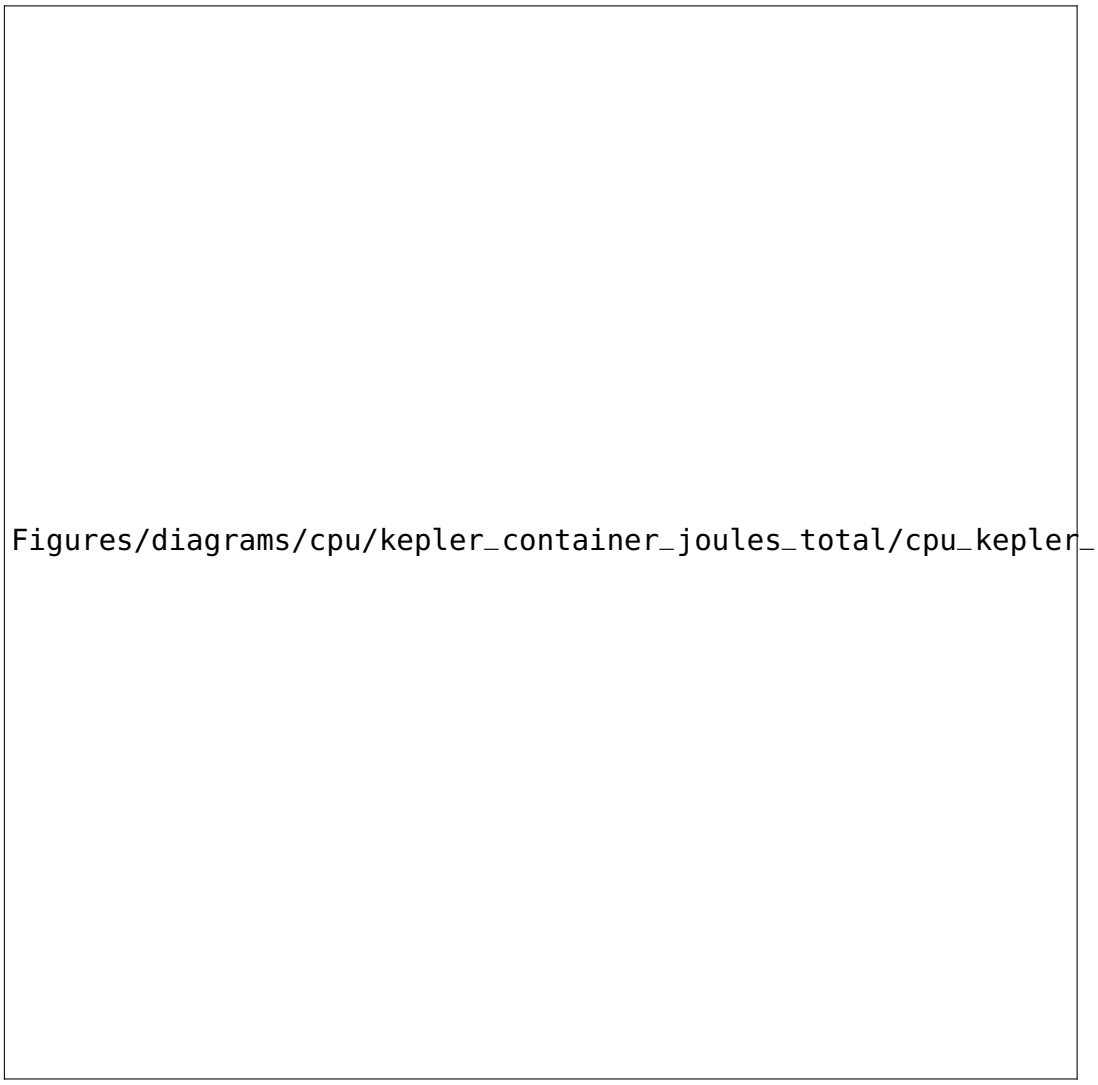
FIGURE 5.2: KEPLER container-level package energy consumption

The figure indicates a clear upward trend in package energy consumption, with distinct steps that correspond to the expected workload increases. A strong correlation is observed between KEPLER's reported package energy consumption and the test

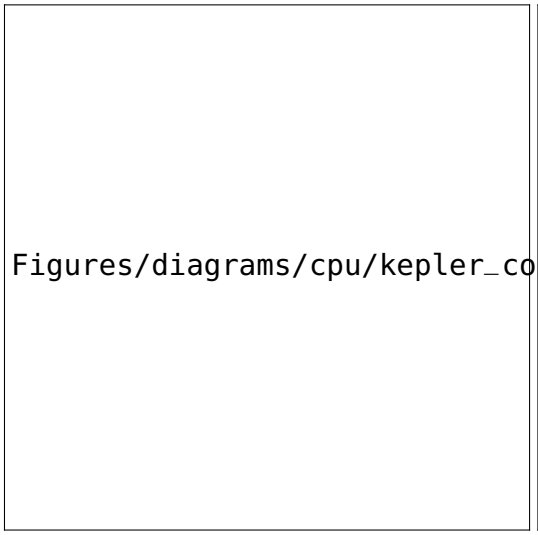
workload. However, the relationship between energy consumption and workload is non-linear: while a 10% workload averages around 2.5 Watts, a 90% workload results in only a doubling of energy consumption despite the workload increasing by a factor of nine.

Additionally, KEPLER's package energy measurements remain consistent regardless of whether the node is idle or busy, showing no statistically significant difference.

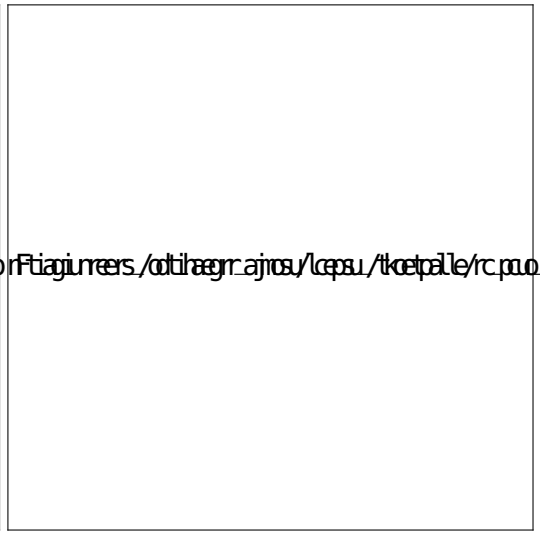
A set of figures illustrating container energy consumption, DRAM energy consumption, and *Other* energy consumption components is provided in Figures 5.3a to 5.3c.



(A) Total container-level energy consumption.



(B) *Other* energy consumption



(C) DRAM energy consumption.

FIGURE 5.3: KEPLER container-level energy consumption during a CPU stress test

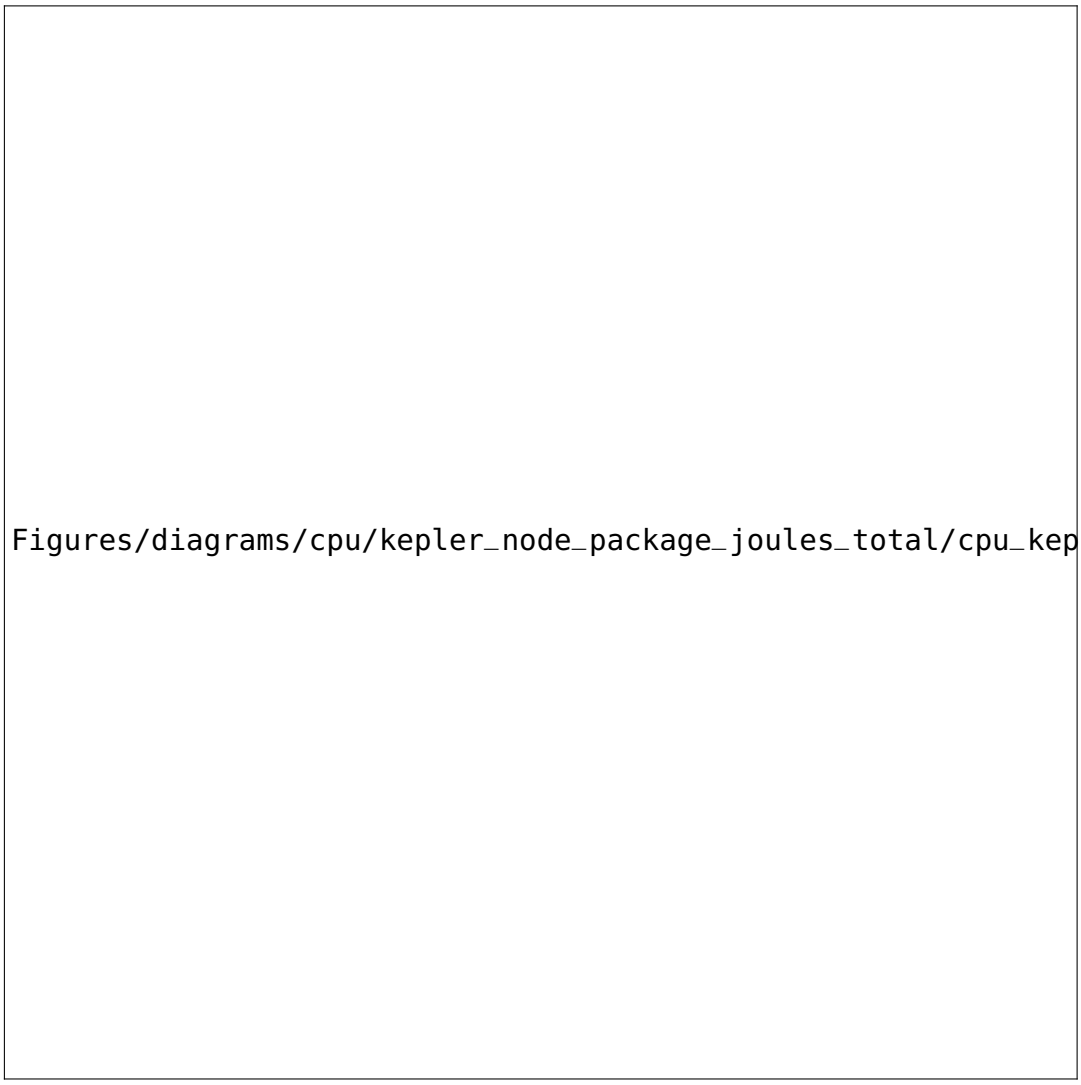
The three figures for container energy consumption, DRAM energy consumption, and *Other* components (representing host components excluding CPU and DRAM) show less direct correlation with workload than the package power.

Observations:

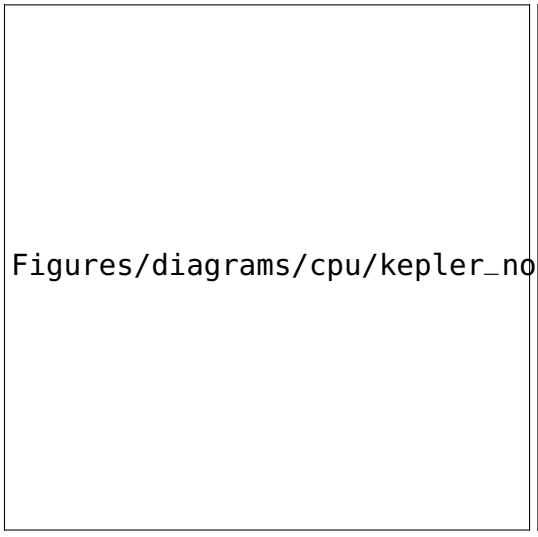
- In Figure 5.3a, a slight upward trend is detectable as the measured container energy consumption rises by about 5 Watts. This change mirrors the increase in package energy consumption from the previous Figure 5.2, where an approximately 5 Watt increase can be seen.
- In Figure 5.3b, showing the *Other* (i.e., Non-CPU/DRAM) container energy consumption, no clear trend can be seen in the data. This is expected since only the CPU itself was stressed. However, the high amount of *Other* energy consumption is surprising, given that it is roughly double that of the CPU package energy consumption.
- The measured DRAM is not affected by CPU stress, which is as expected. With between 0.5 and 1 Watt, DRAM energy consumption is comparably low.
- During the second part of the experiment (i.e., testing with a busy node), all metrics seem to have a slightly smoother curve but are not significantly higher or lower when compared to the experiment on an idle node.

5.1.2 Node-Level metrics during a CPU stress test

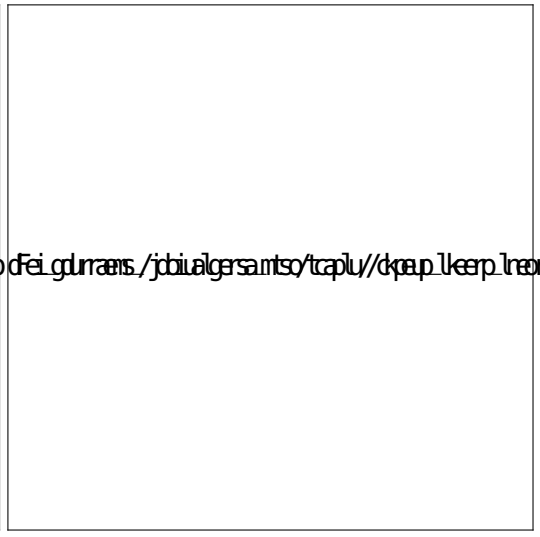
Figures illustrating node-level package energy consumption, DRAM energy consumption, and *Other* energy consumption are provided in Figures 5.4a to 5.4c. For Node-level energy consumtion, KEPLER distinguishes between idle and dynamic power consumption.



(A) Node Package energy consumption



(B) Node DRAMenergy consumption



(C) Node *Other* energy consumption

FIGURE 5.4: KEPLER node-level energy consumption during a CPU stress test

The following observations can be made:

- The most strinking obervation is the relatively high idle energy consumption of the node, seen in all figures. While figure 5.4a shows the rising dynamic energy consumption resulting from the CPU stress test, the idle energy consumption still far exceeds the dynamic energy.
- A key deduction is that the dynamic "Other"'-energy consumption seen in figure 5.4c is independent from the CPU stress test load. This further supports the conclusion that while "Other"' system components contribute most to the overall platform energy consumption, the are generally unaffected by CPU workload.

5.1.3 Overall Conclusions

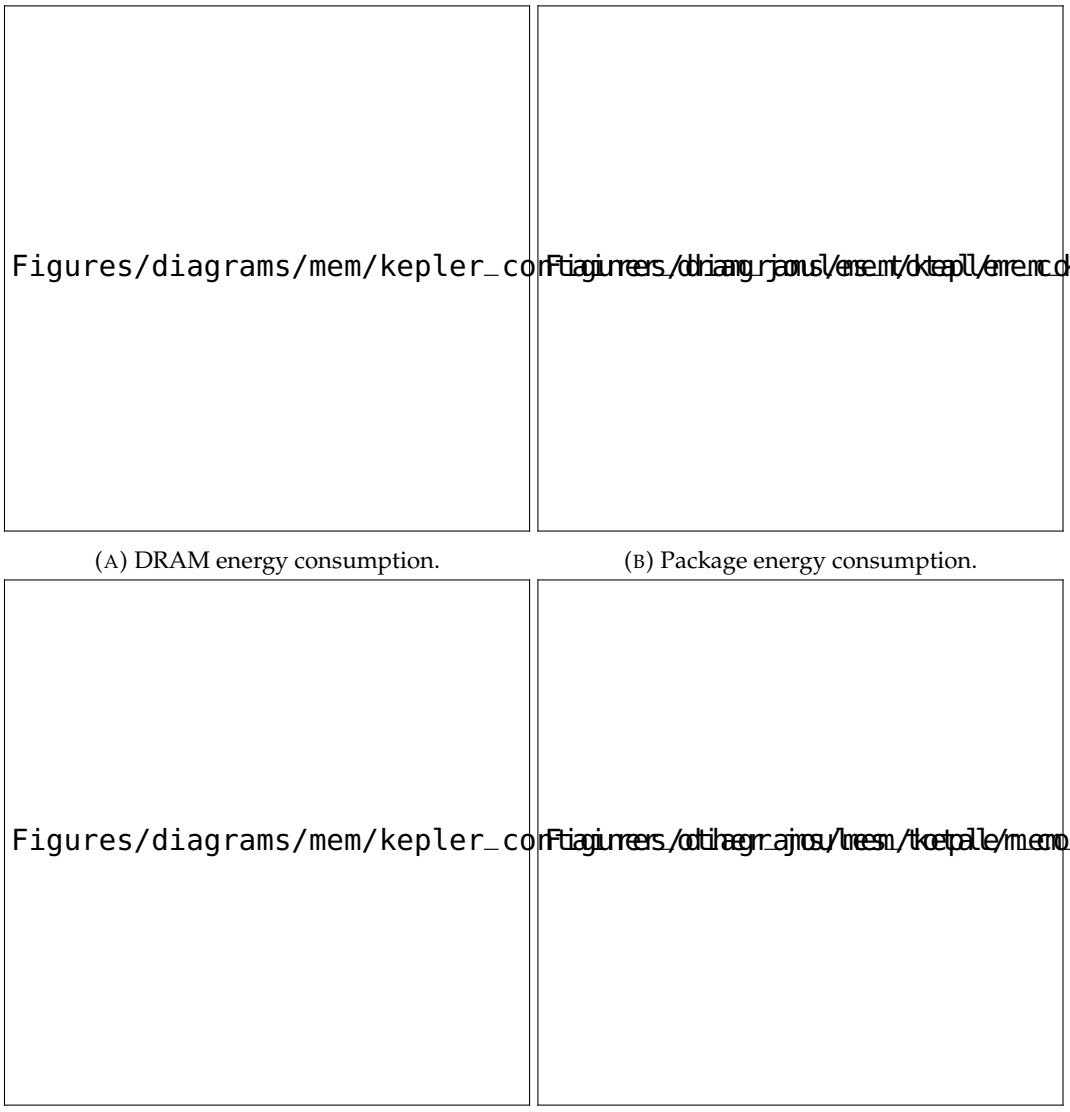
The CPU stress test results demonstrate that KEPLER accurately captures workload-dependent variations in energy consumption. Key takeaways from the analysis include:

- KEPLER's CPU package energy measurements exhibit a correlation with workload intensity, although the relationship is non-linear.
- High idle energy consumption at the node level suggests that a significant portion of energy use is independent of CPU workload.
- The *Other* component energy consumption remains largely static, reinforcing that *Other* components contribute primarily to baseline energy consumption rather than dynamic variations.

5.2 Memory Stress Test Results

5.2.1 Container-Level Metrics During a Memory Stress Test

The following figures 5.5a to 5.5d show the container-level energy consumption metrics published by KEPLER during the memory stress test.



(A) DRAM energy consumption.

(B) Package energy consumption.

(C) *Other* energy consumption.

(D) Total KEPLER container energy consumption.

FIGURE 5.5: Container-level energy consumption during a memory stress test.

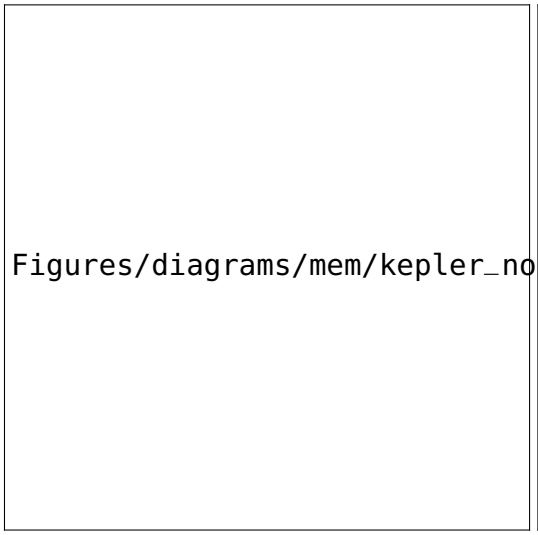
The following observations can be made:

- None of the four published energy metrics correlate with the applied memory stress load. There is also no significant difference between the stress test on an idle versus a busy node. None of the energy metrics indicate that a memory stress test is being performed.
- Figure 5.5b shows an average container-level DRAM energy consumption of approximately 0.3 Watts. This is significantly lower than the 0.7 Watts measured during the CPU stress test (Figure 5.3c), which also displays a notable upward trend during higher CPU workloads.

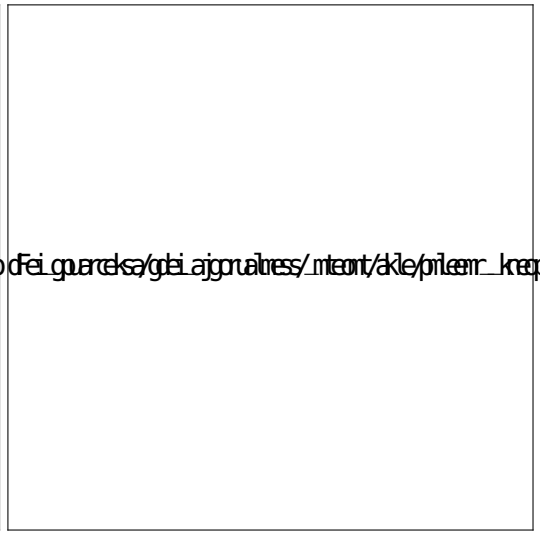
5.2.2 Node-Level Metrics During a Memory Stress Test

The following figures 5.6a to 5.6c show the node-level idle and dynamic energy consumption metrics published by KEPLER during the memory stress test.

(A) DRAM energy consumption



(B) Package energy consumption



(C) *Other* energy consumption

FIGURE 5.6: Node-level energy consumption during a memory stress test

The node-level metrics collected during the DRAM stress test present a similar picture as the container-level metrics. The following observations can be made:

- No node-level energy consumption metric displays a correlation with the memory stress applied during the test.
- All node-level energy consumption metrics exhibit significantly higher idle energy consumption compared to dynamic power consumption.

5.2.3 Overall Conclusions

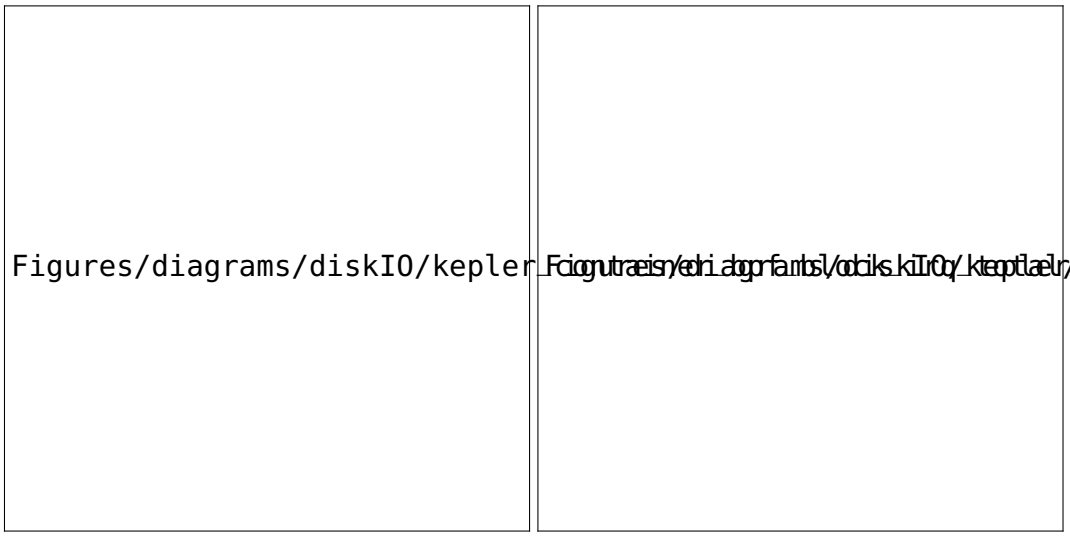
The following key takeaways can be deduced from the memory stress test results:

- The memory stress test results do not indicate any capability of KEPLER to reliably track memory energy consumption. None of the metrics respond to the various stimuli of the test scenario.
- However, the container-level DRAM metric appears to be affected by CPU stress, as observed in the CPU stress test.

5.3 Disk I/O Stress Test Results

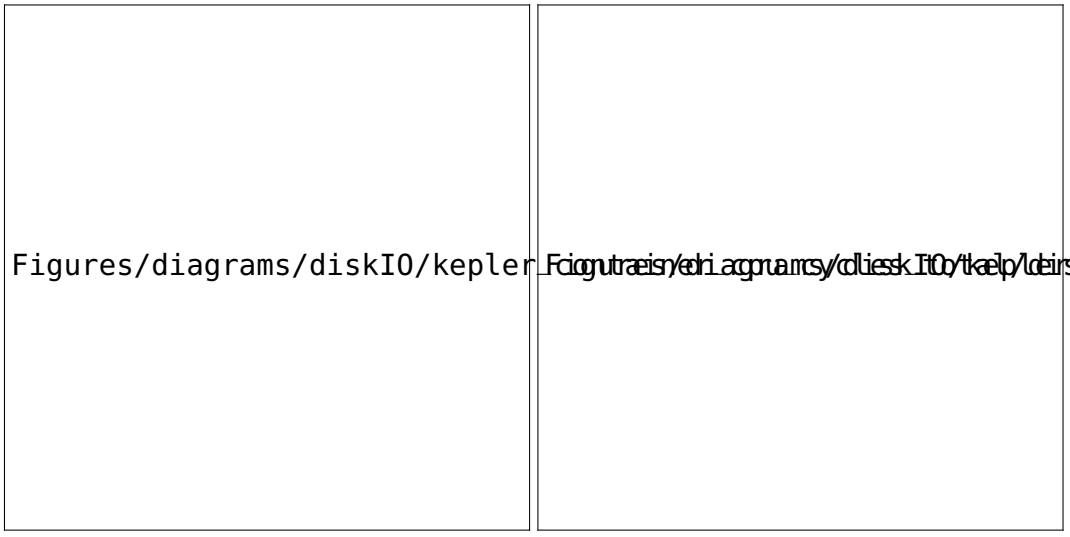
5.3.1 Container-Level Metrics During a Disk I/O Stress Test

The following figures 5.7a to 5.7d show the CPU metrics published by KEPLER during the Disk I/O stress experiment.



(A) Block IRQ

(B) Cache misses

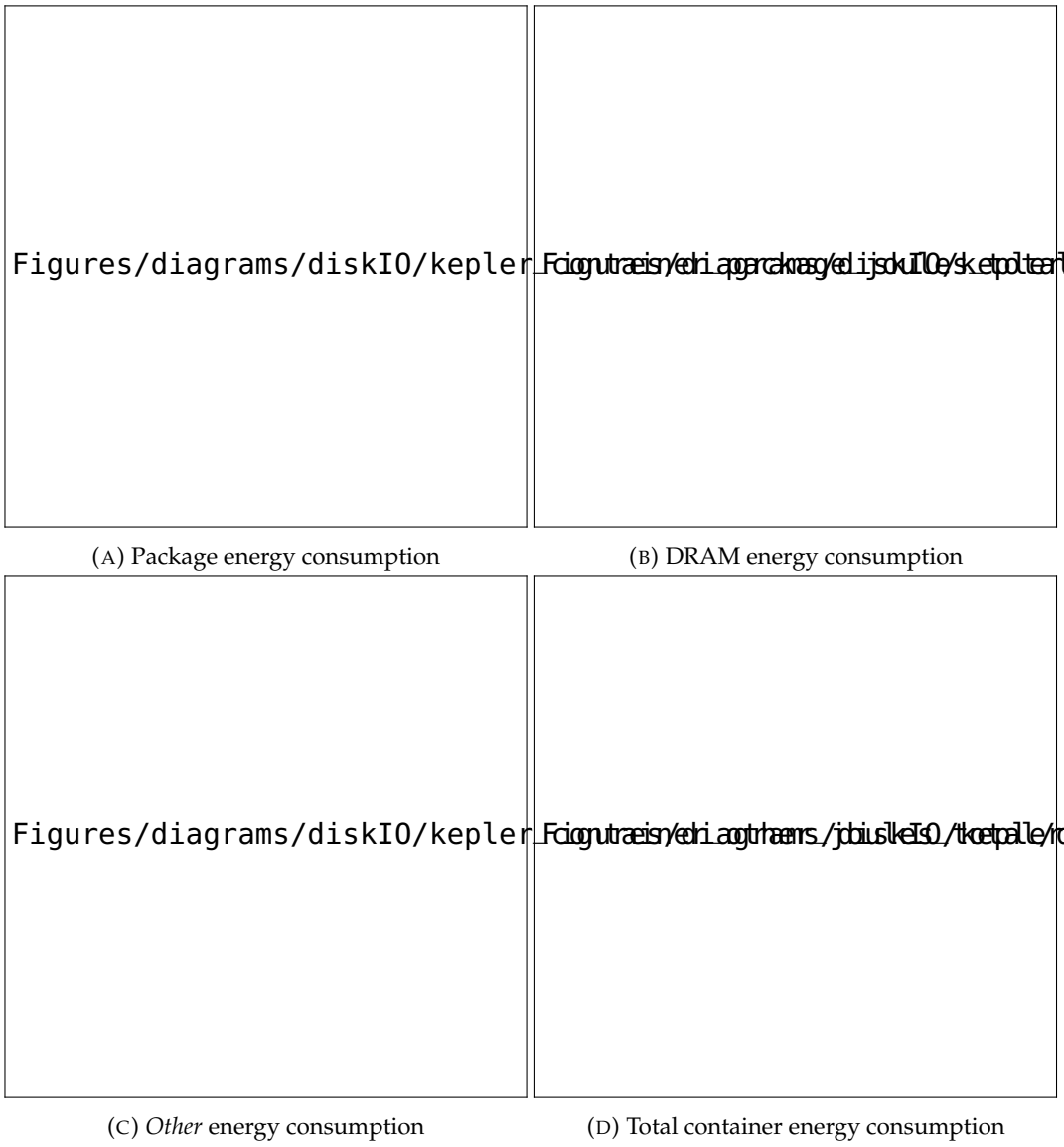


(C) CPU instructions

(D) CPU cycles

FIGURE 5.7: Container-level CPU metrics during Disk I/O stress test

All four figures behave as expected and validate the general test procedure. Notably, the operations per second in the figures for cache misses, CPU instructions, and CPU cycles do not scale linearly with the workload. The relative difference between the low-workload and high-workload tests is approximately 330% (cache misses), 250% (CPU instructions), and 350% (CPU cycles), all of which are significantly lower than the 900% relative difference in the applied load.



(A) Package energy consumption (B) DRAM energy consumption

(C) *Other* energy consumption (D) Total container energy consumption

FIGURE 5.8: Container-level energy consumption during a Disk I/O stress test

Figures 5.8a to 5.8d show the KEPLER metrics for the energy consumption of the package, DRAM, *Other* components, and total container energy consumption. Several observations can be made:

- The KEPLER-measured values across all four figures appear to follow a bimodal distribution, where values exhibit either *low* or *high* states. While varying HDD rotation speeds might explain the curve of *Other* components' energy consumption and total container energy consumption, they do not account for the behavior seen in package energy consumption and DRAM.
- In all four figures, the relative energy consumption appears nearly identical regardless of the measured component. This is particularly evident in the *Other* components and total container energy consumption figures, which seem identical except for a scaling factor of 1.5.

- All figures show strong temporal correlation with the test intervals, demonstrating that KEPLER detects changes in disk load.

To further analyze the metrics during a Disk I/O stress test, the test was repeated, with the package energy consumption results shown in Figure 5.9.

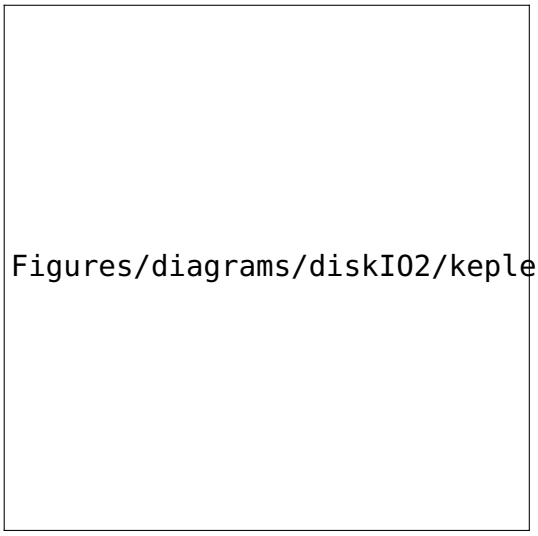


FIGURE 5.9: Container-level package energy consumption during a second experiment

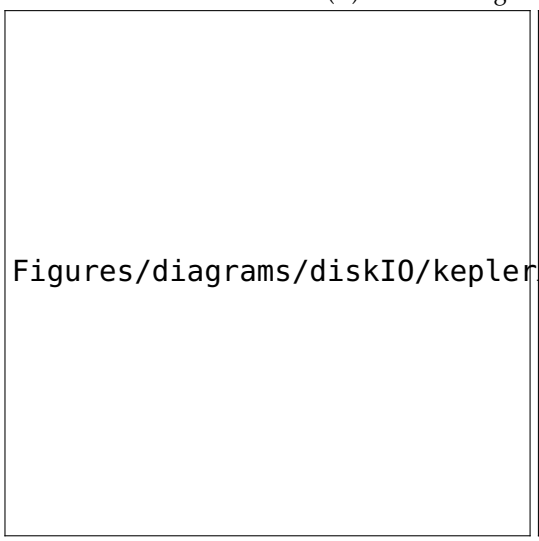
The following observations were very comparable to the first experiment:

- All joule-based metrics still displayed a bimodal distribution.
- All joule-based metrics appear identical aside from differences in scale.
- Joule-based metrics show an observable upwards trend.

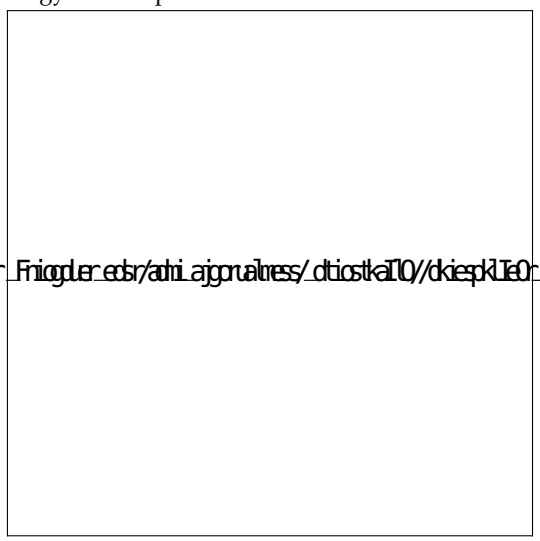
5.3.2 Node-Level Metrics During a Disk I/O Stress Test

The following figures 5.4a to 5.4a show the node-level energy consumption for the package, DRAM, and *Other* components, separated into idle and dynamic node energy consumption.

(A) Node Package energy consumption



(B) Node DRAM energy consumption



(C) Node *Other* energy consumption

FIGURE 5.10: KEPLER node-level energy consumption during a Disk I/O stress test

The following observations can be made about the node-level energy consumption:

- Across all components, the difference between idle and dynamic power is pronounced. The majority of the node's energy consumption occurs as idle power.
- The figures for node DRAM and *Other* components do not show an increasing trend for dynamic energy consumption. Figure 5.10a reveals an upward trend of approximately 50% in dynamic energy consumption, which is still significantly overshadowed by idle node energy consumption.

5.3.3 Overall Conclusions

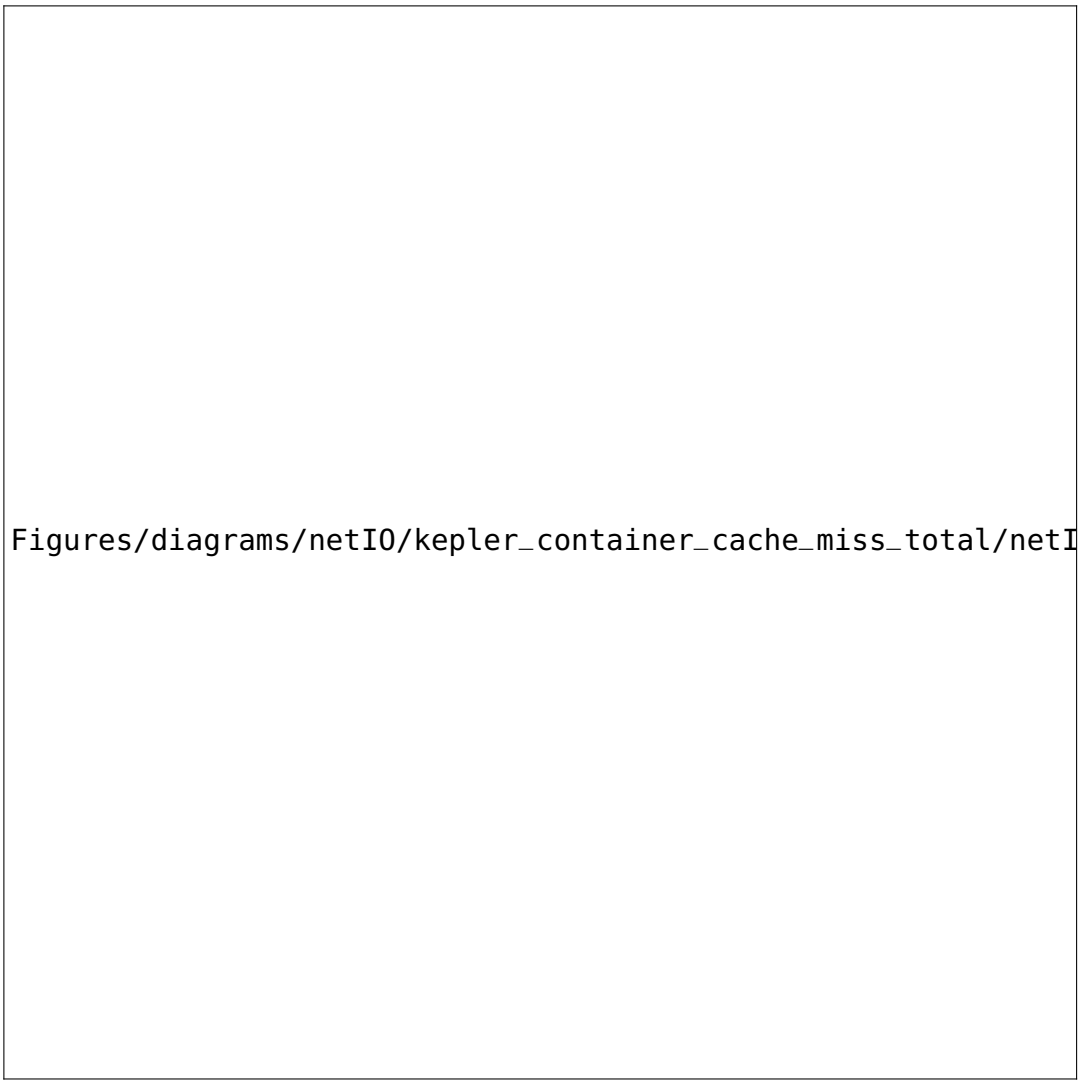
The following overall conclusions can be drawn regarding KEPLER metrics during a Disk I/O test:

- KEPLER provides plausible metrics for Block IRQ, Cache Misses, CPU Instructions, and CPU Cycles during a Disk I/O stress test.
- The accuracy of KEPLER's container-level joule-based metrics is questionable. While they exhibit an intriguing bimodal distribution, they do not necessarily align with the Disk I/O workload.
- KEPLER's container-level joule-based metrics reliably *detect* changes in Disk I/O workload, but their predictability remains uncertain.
- Node-level KEPLER metrics do not suggest any significant impact of Disk I/O workload on DRAM or *Other* components.
- The fact that the variations in energy consumption shown in container-level metrics cannot clearly be traced in eighter idle or dynamic node energy consumption raises further questions.
- The hypothesis that disk I/O stress does not significantly contribute to node power cannot be rejected or proved. Further research is necessary.

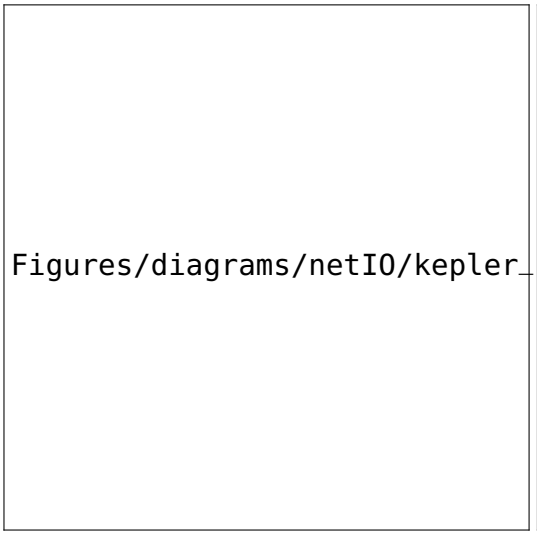
5.4 Network I/O Stress Test Results

5.4.1 Container-Level Metrics During a Network I/O Stress Test

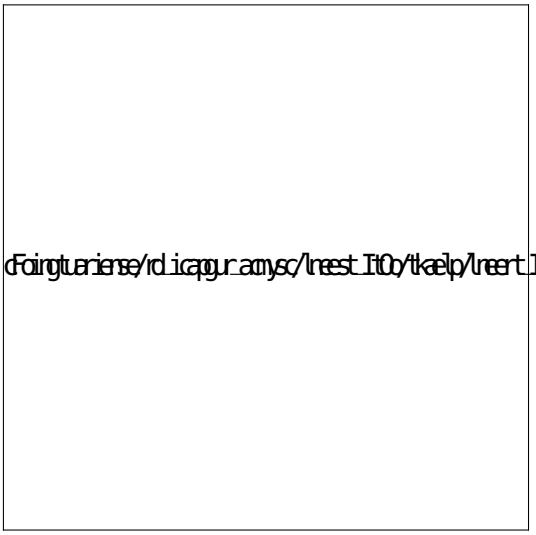
The following figures 5.11a to 5.11c show the CPU metrics published by KEPLER during the Network I/O stress experiment.



(A) Cache misses



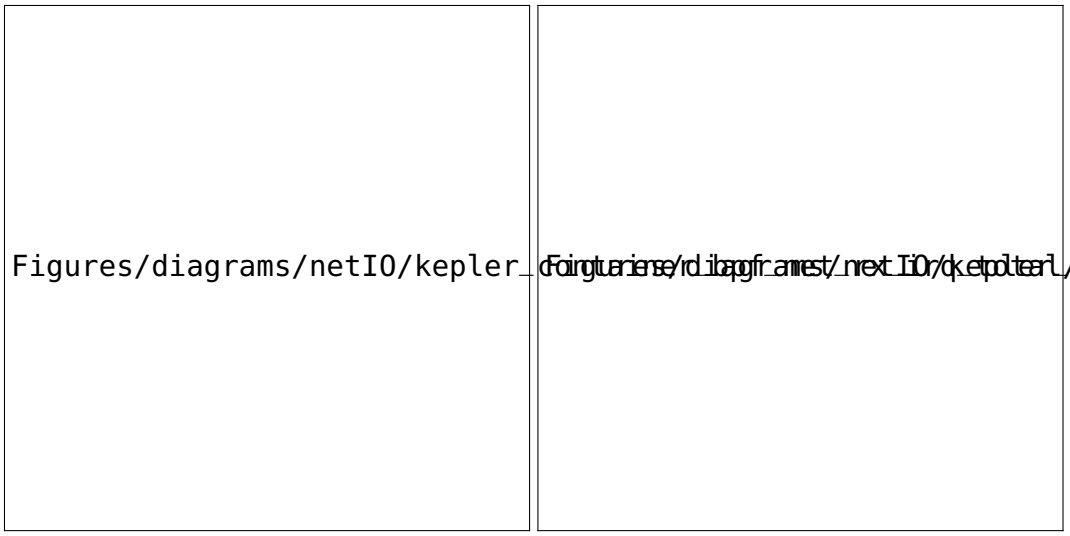
(B) CPU instructions



(C) CPU cycles

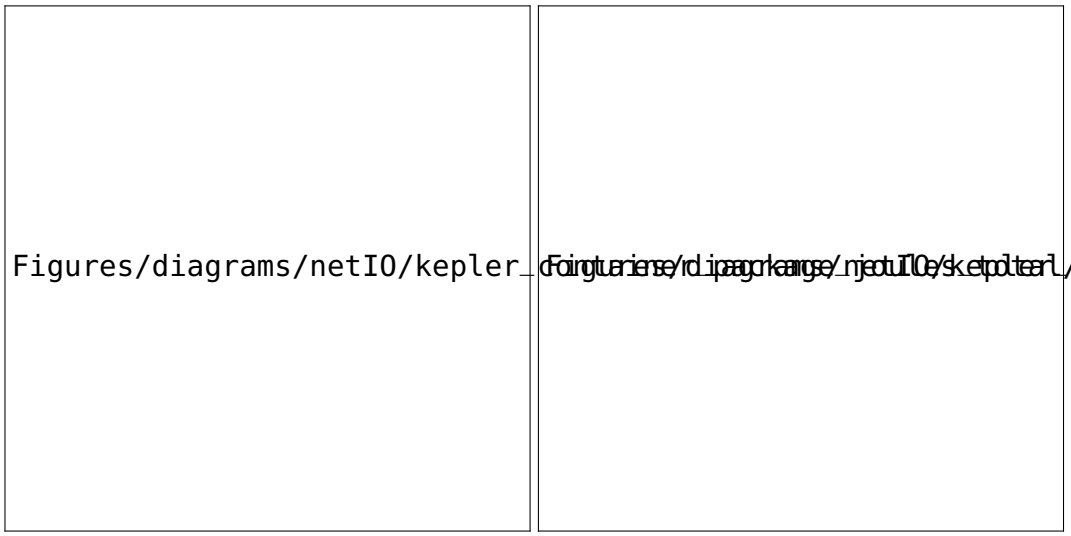
FIGURE 5.11: Container-level CPU metrics during a Network I/O stress test

The results do not provide a clear explanation for the applied test stress. While the metrics correlate in time with the applied Network I/O load, their values do not clearly correspond to the applied stress. The KEPLER metrics for RX IRQ and TX IRQ in Figures 5.12a and 5.12b present a similar trend, though with significantly higher variance.



(A) RX IRQ (B) TX IRQ

FIGURE 5.12: Container-level IRQ metrics during a Network I/O



(A) Package energy consumption (B) DRAM energy consumption

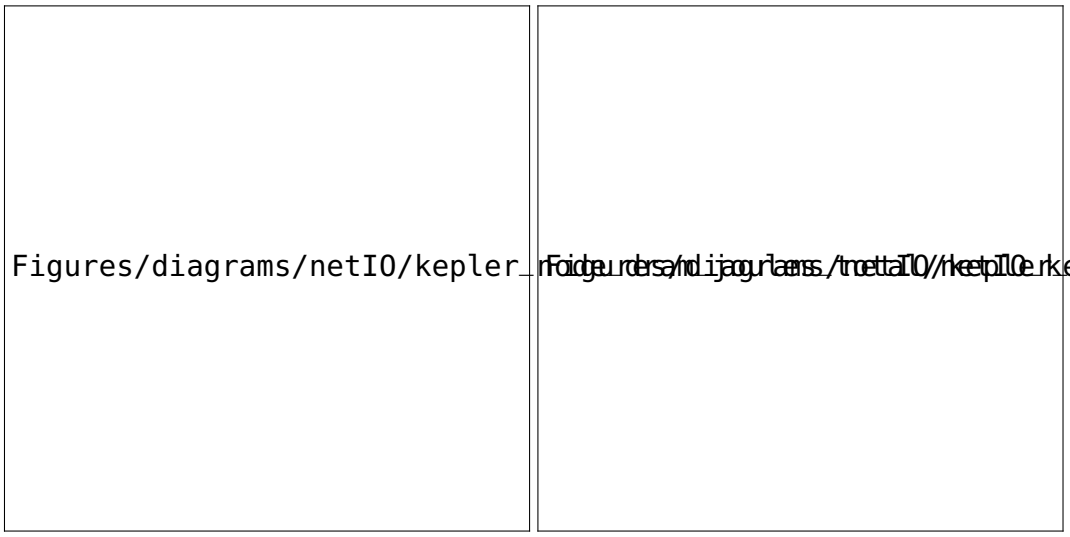
(C) *Other* energy consumption (D) Total container energy consumption

FIGURE 5.13: Container-level energy consumption during a Network I/O stress test

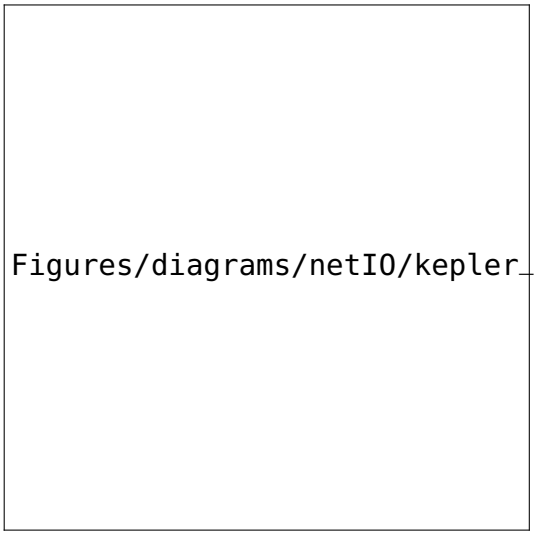
Figures 5.13a to 5.13d display KEPLER metrics for package, DRAM, *Other*, and overall container-level energy consumption. The patterns remain consistent, with the metrics indicating transitions between different workloads, but the values (while nearly constant for each experiment) appear unrelated to the Network I/O activity.

5.4.2 Node-Level Metrics During a Network I/O Stress Test

The following figures 5.14a to 5.14c show the node-level idle and dynamic energy consumption metrics published by KEPLER during the Network stress test.



(A) DRAM energy consumption (B) Package energy consumption



(C) *Other* energy consumption

FIGURE 5.14: Node-level energy consumption during a Network stress test

Much like the previous Disk I/O test, the node-level energy consumption metrics do not show any clear indication of the applied network load. The only notable observation is the stepwise increase in dynamic package energy consumption, which is likely related to the actual generation of network traffic.

5.4.3 Overall Conclusions

Unfortunately, the test does not indicate a clear relationship between the applied network I/O load and the KEPLER metrics. Similar to the Disk I/O test, the following conclusions can be drawn:

- The KEPLER metrics are affected by the network stress test, as evidenced by their time-wise correlation with the applied workload.

- However, the metric values do not logically correlate with the actual network traffic being generated.

This test was repeated multiple times with consistent results, where the workload continued to time-correlate with the test steps, but the metric values plateaued at seemingly arbitrary levels for the duration of each load test.

Chapter 6

Discussion

6.1 Conclusion and Evaluation

6.1.1 Evaluation of Cluster Setup

The cluster setup has been a success and has proven viable for further Kubernetes testing. While implementing the entire setup in an automated manner introduced significant additional effort, the resulting cluster deployment functioned reliably throughout the entire project. The ability to tear down and re-deploy the cluster to any desired depth—ranging from Kubernetes deployments and configurations to a complete reinstallation—proved invaluable during testing. This ensured that any misconfigurations introduced during implementation could be entirely removed, preventing any residual effects from failed installations or incorrect configurations.

The automated cluster setup was explicitly designed to be easily transferable to different hardware, a feature that, while not tested in this project, significantly enhances its reusability. Future projects could adopt and modify the setup with minimal adjustments, allowing researchers and engineers to rapidly deploy an experimental Kubernetes cluster in diverse environments.

One of the main constraints of this project was the decision not to implement a high-availability (HA) cluster. Given the project's focus on energy efficiency measurements and not on production-ready reliability, this was a valid trade-off. However, in large-scale production environments, HA clusters are the norm. Energy efficiency research in these environments would provide additional insights into how energy optimizations affect large, distributed clusters in real-world workloads.

Finally, graphical tools such as Rancher proved invaluable during the configuration and experimentation phases. Rancher's centralized UI provided a clear overview of the cluster state, significantly reducing the complexity of Kubernetes troubleshooting and management. While the project was fully automated, Rancher complemented the setup by allowing real-time monitoring and rapid identification of configuration issues.

6.1.2 Evaluation of Monitoring Setup

The monitoring setup proved to be effective for energy consumption testing. The use of Prometheus, the de facto standard for Kubernetes monitoring, ensured compatibility with a broad range of tools and provided access to a large knowledge base of community support, documentation, and third-party integrations. This was

particularly beneficial for KEPLER, which is explicitly designed to integrate with Prometheus, making its deployment and data collection seamless.

A notable limitation of Prometheus is the overhead it introduces. Due to its reliance on periodic metric scraping, it is best suited for system monitoring at multi-second or minute-level intervals. While this is sufficient for general observability, it is a limiting factor in high-resolution energy consumption analysis. KEPLER itself collects an extensive amount of data from eBPF and RAPL, but the necessity of reducing data density for Prometheus-compatible metrics results in a loss of granularity. This makes Prometheus an excellent tool for tracking long-term trends but suboptimal for capturing rapid changes in power consumption.

The use of an NFS-based persistent storage solution on the Kubernetes control node proved successful. Throughout the project's duration—including multiple cluster redeployments—no data was lost. The NFS configuration allowed for a seamless storage experience, ensuring that Prometheus and Grafana retained their monitoring data even when the cluster was reset.

While this monitoring setup is well-suited for research and experimentation, deploying it in a production environment would require significant modifications to ensure data integrity, resilience, and security. For example, Prometheus' data retention settings and storage backend would need to be adjusted for long-term reliability, authentication mechanisms would need to be strengthened, and redundancy mechanisms would need to be introduced to prevent data loss in the event of node failure.

6.1.3 Evaluation of KEPLER

The implementation of KEPLER in this project was largely straightforward, facilitated by the provided Helm chart. However, several configurations were necessary to ensure compatibility with the existing infrastructure. The project documentation, while informative, has not yet reached full maturity, leading to some challenges in setup and troubleshooting. Despite these hurdles, KEPLER's fundamental concept is well-founded, leveraging appropriate data sources to estimate energy consumption at both the container and node levels.

General Observations

- All metrics showed a high and consistent oscillation. Since all metrics are calculated from simple counters, this indicates a sync issue, possibly between KEPLER metrics publishing and Prometheu scrape intervals. While this does not reduce the credibility of the data, fixing it would greatly unprove the usability of the metrics.
- **CPU Energy Metrics:** KEPLER successfully captures workload-dependent energy variations, demonstrating a high correlation between CPU stress levels and estimated power consumption. Given that CPU is the dominant factor in overall server energy consumption, this is a significant strength of KEPLER.
- **Memory Energy Metrics:** Unlike CPU metrics, memory energy estimates did

not show a clear correlation with applied workload. However, this is not necessarily a flaw in KEPLER's methodology. Memory generally consumes a relatively small proportion of a server's total power budget and does not flutuate as distinctly as CPU energy consumtion. Thus, the lack of a drastic and distict correlation is not entirely unexpected. However, the experiment did not verify KEPLERs ability to measure memory energy consumption.

- **Disk I/O and Network I/O Metrics:** KEPLER's energy estimates for these categories did not exhibit expected trends. Although the metrics responded to workload variations in a synchronized manner, the exact correlation remains unclear. In particular, disk and network energy consumption values were not directly proportional to the applied stress levels. This anomaly warrants further investigation, especially considering that HDD power consumption tends to be largely independent of workload intensity.

6.1.4 Credible Takeaways from the test results

KEPLER's package metrics appear to work well and provide believable results. While the actual numbers were not verified in this project, the reported metrics closely correlate with CPU workload trends observed during testing.

Energy consumption does not scale linearly with workload. For package power, an increase from 10% to 90% workload resulted in only an approximate 250% increase in KEPLER-measured energy consumption. This aligns with the general understanding that servers operate most efficiently at high utilization.

Idle energy consumption was consistently estimated to be much higher than dynamic energy consumption. While it is known that older servers often have high idle power usage, KEPLER's estimation that idle energy consumption reaches 90% on a CPU-stressed server seems unlikely and warrants further investigation.

These results suggest that KEPLER's CPU workload estimation is reliable, but the inconsistencies in memory, disk, and network energy metrics highlight areas requiring further validation. This provides a natural transition into the areas of future research required to refine and validate KEPLER's performance.

6.2 Future Work

6.2.1 Detailed Analysis of KEPLER

While KEPLER demonstrates strong capabilities in CPU power estimation, the inconsistencies in its memory, disk, and network energy metrics indicate areas requiring further research. A key limitation of this study was the focus on a single server type. A valuable next step would be to compare KEPLER's energy estimates across different hardware configurations to assess generalizability.

6.2.2 KEPLER metrics verification through elaborate tests, possibly using measuring hardware

KEPLER metrics should be further verified through more extensive and varied test scenarios, utilizing different tools. In some cases, integrating physical power measurement hardware could provide additional validation. With a deeper understanding of KEPLER's functionality, it may become possible to directly verify the reported metrics. Ultimately, ensuring that KEPLER metrics accurately describe energy consumption at both the node and container levels is crucial to establishing its reliability.

6.2.3 Kubernetes Cluster Energy Efficiency Optimization

If KEPLER metrics can be reliably used to assess cluster-wide energy consumption, further research could focus on optimizing energy efficiency in Kubernetes environments. Potential areas of study include evaluating existing energy-saving techniques such as carbon-aware schedulers, developing new efficiency measures, and assessing the impact of different cluster configurations. For example, experiments could explore potential energy savings achieved by removing high-availability features or dynamically shutting down and restarting servers based on workload demand.

6.3 Final Conclusion

This project successfully established an experimental Kubernetes cluster with integrated energy monitoring. The results demonstrate that KEPLER is a promising tool for CPU energy estimation, though further refinement is needed for other resource types. Moving forward, improved metric validation, hardware comparisons, and research into cluster-wide optimizations will be key to leveraging KEPLER for practical energy efficiency improvements.

Appendix A

KEPLER-provided power metrics

A.1 Summary of KEPLER-Produced Metrics, according to KEPLER documentation[KEPLERDocumentation]

KEPLER collects metrics at both container and node levels, focusing on energy consumption, resource utilization, and platform-specific data.

A.1.1 Container-Level Metrics

Energy Consumption

- **kepler_container_joules_total**: This metric is the aggregated package/socket energy consumption of CPU, dram, gpus, and other host components for a given container. Each component has individual metrics which are detailed next in this document. This metric simplifies the Prometheus metric for performance reasons. A very large promQL query typically introduces a very high overhead on Prometheus.

- **kepler_container_core_joules_total**: This measures the total energy consumption on CPU cores that a certain container has used. Generally, when the system has access to RAPL metrics, this metric will reflect the proportional container energy consumption of the RAPL Power Plan 0 (PP0), which is the energy consumed by all CPU cores in the socket. However, this metric is processor model specific and may not be available on some server CPUs. The RAPL CPU metric that is available on all processors that support RAPL is the package, which we will detail on another metric. In some cases where RAPL is available but core metrics are not, Kepler may use the energy consumption package. But note that package energy consumption is not just from CPU cores, it is all socket energy consumption. In case RAPL is not available, Kepler might estimate this metric using the model server.

- **kepler_container_dram_joules_total**: This metric describes the total energy spent in DRAM by a container.

- **kepler_container_uncore_joules_total**: This measures the cumulative energy consumed by certain uncore components, which are typically the last level cache, integrated GPU and memory controller, but the number of components may vary depending on the system. The uncore metric is processor model specific and may not be available on some server CPUs. When RAPL is not

available, Kepler can estimate this metric using the model server if the node CPU supports the uncore metric.

- **kepler_container_package_joules_total**: This measures the cumulative energy consumed by the CPU socket, including all cores and uncore components (e.g. last-level cache, integrated GPU and memory controller). RAPL package energy is typically the PP0 + PP1, but PP1 counter may or may not account for all energy usage by uncore components. Therefore, package energy consumption may be higher than core + uncore. When RAPL is not available, Kepler might estimate this metric using the model server.

- **kepler_container_other_joules_total**: This measures the cumulative energy consumption on other host components besides the CPU and DRAM. The vast majority of motherboards have a energy consumption sensor that can be accessed via the kernel acpi or ipmi. This sensor reports the energy consumption of the entire system. In addition, some processor architectures support the RAPL platform domain (PSys) which is the energy consumed by the "System on a chipt" (SOC). Generally, this metric is the host energy consumption (from acpi) less the RAPL Package and DRAM.

- **kepler_container_gpu_joules_total**: This measures the total energy consumption on the GPUs that a certain container has used. Currently, Kepler only supports NVIDIA GPUs, but this metric will also reflect other accelerators in the future. So when the system has NVIDIA GPUs, Kepler can calculate the energy consumption of the container's gpu using the GPU's processes energy consumption and utilization via NVIDIA nvml package.

Resource Utilization

- **Base Metric**

  - **kepler_container_bpf_cpu_time_us_total**: This measures the total CPU time used by the container using BPF tracing. This is a minimum exposed metric.

- **Hardware Counter Metrics**

  - **kepler_container_cpu_cycles_total**: This measures the total CPU cycles used by the container using hardware counters. To support fine-grained analysis of performance and resource utilization, hardware counters are particularly desirable due to its granularity and precision.. The CPU cycles is a metric directly related to CPU frequency. On systems where processors run at a fixed frequency, CPU cycles and total CPU time are roughly equivalent. On systems where processors run at varying frequencies, CPU cycles and total CPU time will have different values.

  - **kepler_container_cpu_instructions_total**: This measure the total cpu instructions used by the container using hardware counters. CPU instructions are the de facto metric for accounting for CPU utilization.

  - **kepler_container_cache_miss_total**: This measures the total cache miss that has occurred for a given container using hardware counters. As there

is no event counter that measures memory access directly, the number of last-level cache misses gives a good proxy for the memory access number. If an LLC read miss occurs, a read access to main memory should occur (but note that this is not necessarily the case for LLC write misses under a write-back cache policy).

- **IRQ Metrics**
  - **kepler_container_bpf_net_tx_irq_total**: This measures the total transmitted packets to network cards of the container using BPF tracing.
  - **kepler_container_bpf_net_rx_irq_total**: This measures the total packets received from network cards of the container using BPF tracing.
  - **kepler_container_bpf_block_irq_total**: This measures block I/O called of the container using BPF tracing.

A.1.2 Node-Level Metrics

Energy Consumption

- **kepler_node_core_joules_total**: Similar to container metrics, but representing the aggregation of all containers running on the node and operating system (i.e. "system_process").
- **kepler_node_dram_joules_total**: Similar to container metrics, but representing the aggregation of all containers running on the node and operating system (i.e. "system_process").
- **kepler_node_uncore_joules_total**: Similar to container metrics, but representing the aggregation of all containers running on the node and operating system (i.e. "system_process").
- **kepler_node_package_joules_total**: Similar to container metrics, but representing the aggregation of all containers running on the node and operating system (i.e. "system_process").
- **kepler_node_other_joules_total**: Similar to container metrics, but representing the aggregation of all containers running on the node and operating system (i.e. "system_process").
- **kepler_node_gpu_joules_total**: Similar to container metrics, but representing the aggregation of all containers running on the node and operating system (i.e. "system_process").
- **kepler_node_platform_joules_total**: This metric represents the total energy consumption of the host. The vast majority of motherboards have a energy consumption sensor that can be accessed via the acpi or ipmi kernel. This sensor reports the energy consumption of the entire system. In addition, some processor architectures support the RAPL platform domain (PSys) which is the energy consumed by the "System on a chipt" (SOC). Generally, this metric is the host energy consumption from Redfish BMC or acpi.

Node Metadata

- **kepler_node_info**: This metric shows the node metadata like the node CPU architecture.
- **kepler_node_energy_stat**: This metric contains multiple metrics from nodes labeled with container resource utilization cgroup metrics that are used in the model server. This metric is specific to the model server and can be updated at any time.

Accelerator Metrics

- **kepler_node_accelerator_intel_qat**: This measures the utilization of the accelerator Intel QAT on a certain node. When the system has Intel QATs, Kepler can calculate the utilization of the node's QATs through telemetry.