FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards Fine-grained, Holistic Energy Control in Large-Scale Computing Infrastructures

Sara Lima Pereira

Mestrado em Engenharia Informática e Computação

Supervisors: Carlos Baquero-Moreno, Ricardo Macedo, Cláudia Brito

July 21, 2024

Towards Fine-grained, Holistic Energy Control in Large-Scale Computing Infrastructures

Sara Lima Pereira

Mestrado em Engenharia Informática e Computação

July 21, 2024

Abstract

In the era of digitalization and exponential data growth, the demand for infrastructures capable of supporting large-scale applications with requirements for high scalability, low latency, and high throughput has surged. Cloud computing data centers and high-performance supercomputers have emerged to fulfill these needs, offering capabilities beyond traditional private infrastructures. Although these infrastructures are optimized for performance, their composition of thousands of nodes results in significant energy consumption. This growing energy consumption has become a major concern, leading to substantial investments in more efficient hardware and related cluster services, such as cooling. Despite these efforts, data centers' energy use remains substantial, potentially accounting for 8% to 13% of global electricity by 2030.

Addressing this challenge requires optimizing both hardware and software stacks. Current software systems are suboptimal, focusing on individual applications or components with partial infrastructure visibility or treating all applications equally, regardless of priority and workload characteristics. To address these issues, this work introduces FINER, a novel control system for granular and adaptable energy management in large-scale computing infrastructures. FINER employs a hierarchical architecture with a decoupled data and control plane, enabling fine-tuned energy allocation and optimization strategies. The hierarchical structure includes cluster, rack, and node controllers, ensuring efficient communication and dynamic adjustment to system and workload changes. Further, FINER introduces two new energy control algorithms. The fairness algorithm ensures proportional energy distribution based on resource usage, while the priority algorithm allocates energy according to node priorities and consumption.

FINER's thorough evaluation demonstrates its effectiveness in dynamically adapting to system changes, efficiently distributing leftover energy, and managing consumption at various levels of granularity, including compute nodes and racks. FINER successfully controls components such as CPU cores and compute nodes with acceptable control cycle latency.

FINER offers a viable solution for hierarchical energy management in data centers, representing a significant step towards sustainable and cost-effective management of large-scale infrastructures. This approach aligns with future trends in high-performance computing, particularly the shift towards ARM-based supercomputers.

Keywords: Energy Consumption, Hierarchical Control, Large-scale Systems

Resumo

Na era da digitalização e do crescimento exponencial dos dados, a procura por infraestruturas capazes de suportar aplicações em larga escala com requisitos de elevada escalabilidade, baixa latência e elevado rendimento aumentou significativamente. Centros de dados de computação em nuvem e supercomputadores de alto desempenho surgiram para atender a essas necessidades, oferecendo capacidades além das infraestruturas privadas tradicionais. Embora essas infraestruturas sejam otimizadas para elevado desempenho, a sua composição de milhares de nós resulta num consumo significativo de energia. Este aumento considerável no consumo de energia tornou-se uma preocupação importante, levando a investimentos substanciais em *hardware* mais eficiente e noutros serviços relacionados como o arrefecimento. Apesar desses esforços, o consumo de energia dos centros de dados continua substancial, podendo representar de 8% a 13% da eletricidade global até 2030.

Abordar este desafio requer a otimização das pilhas de *hardware* e *software*. Os sistemas de *software* atuais são subótimos, focando-se em aplicações ou componentes individuais com visibilidade parcial da infraestrutura ou tratando todas as aplicações de format igual, independentemente da sua prioridade ou características das cargas de trabalho. Para abordar essas questões, este trabalho propõe o FINER, um novo sistema de controlo para gerir de forma granular e adaptável a energia em infraestruturas de computação em larga escala. O FINER emprega uma arquitetura hierárquica com um plano de dados e controlo desacoplados, permitindo estratégias refinadas de alocação e otimização de energia. A estrutura hierárquica inclui controladores de *cluster*, *rack* e nó, garantindo a comunicação eficiente e ajuste dinâmico às mudanças do sistema e da carga de trabalho. Ainda, o FINER introduz dois novos algoritmos de controlo de energia. O algoritmo de equidade assegura uma distribuição proporcional de energia baseada no uso de recursos, enquanto o algoritmo de prioridade aloca energia de acordo com as prioridades e o consumo dos nós.

A avaliação compreensiva do FINER mostra a sua eficácia na adaptação dinámica às mudanças do sistema, na distribuição eficiente da energia sobrante e na gestão do consumo em vários níveis de granularidade, incluindo nós de computação e *racks*. O FINER controla com sucesso componentes como CPU *cores* e nós de computação com uma latência aceitável no ciclo de controlo.

O FINER oferece uma solução viável para a gestão hierárquica de energia em centros de dados, representando um passo significativo em direção à gestão sustentável e económica de infraestruturas em larga escala. Esta abordagem está alinhada com as tendências futuras da computação de alto desempenho, particularmente a mudança para supercomputadores baseados em ARM.

Palavras-chave: Consumo de Energia, Controlo Hierárquico, Sistemas de Grande Escala

Acknowledgements

Completing this dissertation has been a challenging yet rewarding journey, and it would not have been possible without many people's support, guidance, and encouragement. I would like to express my deepest gratitude to those who have contributed to my academic and personal growth during my master's degree in Informatics Engineering. Their invaluable assistance, insights, and support have been instrumental in the successful completion of this research.

First and foremost, I would like to extend my heartfelt gratitude to my supervisors, Professor Carlos Baquero-Moreno, Professor Ricardo Macedo, and Professora Cláudia Brito, for their invaluable guidance and support throughout this research. Firstly, as teachers, they actively contributed to my baseline knowledge in Informatics Engineering and captivated my interest in this field. Now, as supervisors, they encouraged me to pursue this work and provided expertise, insightful feedback, and constructive criticism that have contributed significantly to the quality and direction of my project. Their dedication and patience have been instrumental in helping me navigate the challenges of this research, and I sincerely appreciate their mentorship and encouragement.

Second, I want to thank my family for their support throughout this journey. Their reminders to balance my work with enjoying life have been crucial. Their encouragement and belief in me have kept me motivated and focused.

Third, I express my gratitude to my friends, especially Diana, Mariana, and José, who have been with me throughout this whole journey. We've shared both the tough times and the joyful moments, providing constant support and understanding. Their presence has made this process more meaningful, and I sincerely appreciate their encouragement and friendship.

Finally, I would like to thank the institutions that supported this work. Fundação para a Ciência e a Tecnologia (FCT) supports this work through scholarships 10372/BII-E_B4/2023, and 10911/BI-L-EM_B3/2024. The Informatic Department of the University of Minho, HASLab, and INESC TEC offered me the necessary conditions to develop this work. This work is co-financed by Component 5 - Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56.

Sara Lima Pereira

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AI	Artificial Intelligence
AHP	Analytic Hierarchy Process
API	Application Programming Interface
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
Eq	Equation
FIFO	First In, First Out
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
HPC	High Performance Computing
I/O	Input/Output
IP	Internet Protocol
IT	Computers and Information Technology
KIter	Thousand Iterations
LLM	Large Language Model
MOps	Million Operations
MW	Megawatts
NVMe	Non-Volatile Memory Express
OS	Operating System
PDU	Power Distribution Unit
PID	Process Identifier
PSU	Power Supply Units
QoS	Quality of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SATA	Serial Advanced Technology Attachment
SDN	Software-Defined Networking
SDS	Software-Defined Storage
SLA	Service Level Agreements
SSD	Solid State Drive
UPS	Uninterruptible Power Supply
VM	Virtual Machine
YCSB	Yahoo! Cloud Serving Benchmark

Chapter 1

Introduction

In recent years, it has been observed a significant increase in the number of applications and online services deployed in large-scale computing infrastructures, ranging from social media and streaming platforms to AI-powered tools like virtual assistants (*e.g.*, *ChatGPT*), and content generators (*e.g.*, *Dall-E*, *Copilot*). Unprecedented volumes of data in a wide variety of forms are being produced globally, and to provide the best possible service, large-scale computing resources are essential to process and analyze such data. To accommodate this overwhelming amount of digital services, existing computing infrastructures, such as cloud computing data centers and high-performance computing (HPC) supercomputers, have been scaling up in size [25]. For example, in the last 5 years, the fastest supercomputer listed in the Top 500 list has increased the number of CPU cores by 3.6× (specifically, in 2019, Summit had 2,414,592 CPU cores, while the current number 1, Frontier, comprises 8,699,904 cores) [85]. To ensure these infrastructures are scalable, reliable, and able to provide high throughput and low latency to applications, researchers and practitioners have been mainly focused on addressing performance-related issues, by analyzing existing bottlenecks [39] and designing new techniques and mechanisms to address them, such as caching [58, 22], scheduling [34, 13], I/O-oriented parallelism [95, 67, 90], tiering [82, 30], resource disaggregation [71, 32]. Moreover, given the costs of building and maintaining such infrastructures, a significant emphasis has been placed on maximizing resource utilization through virtualization [51, 10, 84, 87, 68], and containerization techniques [84, 68], while providing isolation, fairness, and quality-of-service (QoS) guarantees to ensure a certain level of performance to applications. Fault tolerance is also a key concern in such infrastructures, to ensure a dependable computing infrastructure, thereby reducing the risk of data loss during failures (*e.g.*, power outages) or service unavailability.

However, with the growth of computational infrastructures, environmental and sustainability concerns have also increased, particularly in reducing energy consumption. Indeed, large-scale computing infrastructures emit high volumes of CO_2 and entail substantial electricity consumption [14]. Recent reports show that in 2018, data centers contributed to approximately 0.3% of overall carbon emissions [52], and projections suggest that by 2030, data centers could account for a substantial portion (8% to 13%) of the world's total electricity usage [25], highlighting the

urgency to address this challenge.

Over the years, this challenge has been addressed mainly by updating the infrastructure with more energy-efficient hardware, including different types of processing units (*i.e.*, CPU, GPU), replacing storage components by using SATA or NVMe SSD devices instead of HDD devices, and introducing specialized hardware for specific tasks in the infrastructure, such as the use SmartNICs for improving the network performance and programmability [63], FPGAs for specialized databases [41], and CXL for disaggregated memory [12, 44], as well as related cluster services, such as cooling [31, 97]. Nevertheless, since data centers continue to grow, their energy consumption remains substantial. For example, the *Frontier* and *Fugaku* supercomputers, currently placing first and fourth in the Top 500 list, respectively, consume approximately 22.8 MW and 29.9 MW when used at peak performance [85], even though they are equipped with efficient hardware. As such, solely improving the hardware components is not enough to solve the problem, as energy consumption in large-scale computing infrastructures is heavily dependent on the applications and workloads they service. Additionally, compute nodes also consume a significant amount of energy even in an *idle* state [33].

In this context, adjusting the software stack[1] where applications are deployed is essential, as it lacks mechanisms for improving their energy efficiency. In particular, solutions focused on improving the energy consumption of the software stack apply two main techniques: workload consolidation and dynamic voltage and frequency scaling (DVFS).

Workload consolidation focuses on optimizing and managing applications with disparate workload needs by grouping them in the same compute node, maximizing resource utilization, and reducing the number of hardware resources needed to service such workloads (for example, combining I/O-bound and compute-bound applications or throughput-oriented and latency-oriented workloads) [76]. As the number of running servers decreases, this practice significantly lowers the overall energy usage of the data center [48].

Alternatively, infrastructures rely on DVFS, a power management technique that dynamically adjusts the voltage and frequency at which a given processing unit operates based on the workload and performance requirements [98, 56]. By adjusting power and speed on a computing device's processors, the number of computation cycles decreases, and the energy consumption is reduced.

1.1 Problem Statement

Despite the existence of the aforementioned energy management techniques, their current application remains suboptimal. This work focuses on two main interconnected causes: coarse-grained energy control and partial visibility.

Coarse-grained energy control Existing approaches, mainly using DVFS, operate at a coarse granularity, treating all applications and services running within a given compute node in the same

[1] By software stack, all the software components running in a given compute node (or set of compute nodes) which are used to servicing data centers' applications, including the operating system, file systems, caches, applications, libraries, schedulers, device drivers, and more, are considered.

way, in terms of energy requirements [62, 57, 45, 20]. Specifically, the same frequency is set to all CPU cores, regardless of the workload or type of task being performed in each. Different applications vary in importance, resource allocation, performance requirements, and overall impact on the system. As a result, applying uniform energy optimizations across an entire data center can degrade performance, failing to address the specific needs and criticalities of individual applications and services. For instance, a CPU-intensive application and an application with periods of inactivity would run at the same frequency. Setting the frequency to its minimum would compromise the CPU-intensive application, degrading its performance. Conversely, setting the frequency to its maximum would lead to energy wastage by the application with inactivity periods. Such approaches lead to non-optimal energy control and potential performance compromises.

Partial visibility Most approaches are often applied in isolation, targeting specific applications, compute nodes, or racks without considering the broader demands of other resources in the cluster [42, 81, 61, 20, 21, 53]. This isolated approach leads to suboptimal decisions for the overall infrastructure, as the lack of coordinated control and global visibility results in inefficiencies. Consequently, one component's energy optimizations might negatively impact interconnected services' performance and energy requirements, leading to suboptimal system-wide energy utilization. For instance, increasing the power of all compute nodes in a rack individually makes it difficult to guarantee consistent power levels at the rack level due to the lack of coordination between the rules applied to each compute node. This lack of holistic visibility prevents efficient dynamic energy adjustments, leading to wasted energy and reduced overall system performance. Additionally, without an overarching control mechanism, unutilized energy resources cannot be dynamically allocated to other tasks, resulting in wasted energy.

Considering the volatile nature of workloads on servers, where workload demands can change rapidly and unpredictably, the lack of fine-grained and dynamic energy management mechanisms compromises the performance of specific tasks or results in wasted energy that could be potentially allocated to other resources. Adaptive energy allocation strategies that can respond to varying workloads, application priorities, and system conditions at different granularity levels are essential. The absence of such adaptive strategies restricts the efficient utilization of energy resources, preventing optimal performance while maintaining energy efficiency.

1.2 Objectives

Considering the challenges in current energy management techniques for large-scale computing infrastructures, this work focuses on two main objectives.

The first objective is to make energy control in large-scale infrastructures more granular by enabling system designers and system administrators to control the energy consumption at different granularity levels, namely at the infrastructure level, rack level, compute node level, or even

specific applications. The system can dynamically adjust energy allocation based on different applications and infrastructure requirements by enabling fine-grained control. This granular control aims to improve the energy efficiency of applications and minimize the overall energy consumption of the infrastructure.

The second objective is to control energy management decisions holistically. Specifically, the aim is to balance the overall energy usage while maintaining individual application performance and quality of service (QoS) requirements, ensuring efficient and effective energy management across the entire data center.

1.3 Contribution

To achieve the aforementioned objectives, this work proposes three main contributions.

Design and prototyping of FINER As the primary contribution, FINER is introduced as a novel energy control system for large-scale computing infrastructures. Inspired by the Software-Defined paradigm [66, 50], FINER follows a decoupled architecture comprising a control plane and a data plane. The control plane follows a hierarchical distribution consisting of multiple controllers (cluster, rack, and node controllers), comprising system-wide visibility of all resources. It continuously monitors and manages the energy at which each component operates, hierarchically disseminating and enforcing energy limits to the infrastructure components. The data plane provides the actual mechanisms (*e.g.*, DVFS-based mechanisms) that enforce the energy decisions generated by the control plane, at the compute node level. To ensure the system can adapt to dynamic workloads and system changes, the control logic follows a feedback loop that continuously monitors the system state and adjusts it accordingly.

Control algorithms As a second contribution, two control algorithms have been proposed to enhance FINER's energy management capabilities, namely, a Priority and a Fairness Algorithm. The *Priority Algorithm* assigns energy targets to each node based on predefined priorities for each component. This means that components such as racks, servers, CPU cores, and applications with higher priority receive a larger energy share, while those with lower priority receive smaller portions of energy. Conversely, the *Fairness Algorithm* allocates energy targets based on the proportional usage of resources. Therefore, components with higher CPU usage percentages, for instance, are granted a larger share of energy. When a component is in an *idle* state or reaches its maximum energy limit, both algorithms distribute the leftover energy among the remaining components. These algorithms ensure that energy optimization is both fine-grained and holistic, addressing the specific needs of individual applications while considering the broader impact on the infrastructure.

Experimental evaluation The proposed energy control system's effectiveness, applicability, and scalability are demonstrated across various testing scenarios. Experiments conducted with different applications show that FINER (i) can effectively control energy resources at different levels of granularity, from application level to server level, and with global visibility across the system; (ii) enables the specification of custom control that can be enforced through different levels of the control hierarchy; and (iii) maintains efficient control decision latency, ensuring prompt response to energy consumption monitoring and algorithm enforcement, even under scaling conditions.

1.4 Document Structure

This document is structured as follows: Chapter 2 presents the background with the fundamental knowledge needed to develop this work concerning data centers, their definition, organization, and energy consumption. Furthermore, it reports a literature review regarding energy control systems and energy control algorithms. Chapter 3 details the proposed solution, FINER, a novel energy control system that holistically manages the energy consumption in large-scale infrastructures with fine-granularity. It begins by outlining the objectives of the proposed solution and its significance in addressing the identified gaps from the literature review. It comprehensively describes the system architecture, implementation decisions, and the proposed control algorithms. In Chapter 4 the solution's effectiveness through a comprehensive experimental evaluation is demonstrated. Finally, Chapter 5 concludes this work by summarizing the key findings and contributions of the research. Additionally, it outlines potential directions for future work by identifying limitations of the current research and proposing areas for further investigation and improvement.

Chapter 2

State Of The Art

This chapter presents the state-of-the-art regarding energy consumption control in large-scale computing infrastructures. It presents the background in Section 2.1 in this context regarding data centers (Section 2.1.1), their architecture (Section 2.1.1.1), power distribution (Section 2.1.1.2), energy consumption (Section 2.1.1.3), and power-controlling methodologies (Section 2.1.2). Moreover, this chapter discusses related work in Section 2.2, namely existing energy consumption control systems (Section 2.2.1) and control algorithms (Section 2.2.2). Accordingly, the following sections will provide valuable information regarding the problems and needs of current large-scale computing infrastructures for energy control.

2.1 Background

This section presents the key concepts for understanding this work. It explains what a data center is, its architecture, and energy expenditure. Furthermore, it explores energy-controlling techniques and methodologies implemented in state-of-the-art solutions.

2.1.1 Data Centers

For a long time, most services operated locally or on a small set of servers. However, with the emergence of web-based services, data centers (an extensive collection of machines providing a service) emerged [18]. As their utilization grew exponentially, the quantity and scale of available data centers had to expand accordingly [25]. Nowadays, in the era of advanced web services, data centers are typically structured with thousands of machines. These data centers are sometimes geo-distributed to cater to clients across different global regions, ensuring minimal latency [18]. Their ability to efficiently process and disseminate data across vast distances has become fundamental in meeting the diverse needs of a globally connected world.

Data centers house critical computing resources, such as mainframes, web and application servers, file servers, messaging servers, application software and the operating systems that run them, storage subsystems, and the network infrastructure. They generally include backup power

supplies, redundant data communications connections, various security devices, and environmental controls (*e.g.*, air conditioning and fire suppression). These infrastructures provide a controlled environment under centralized management and play a crucial role in modern IT (Information Technology), supporting various services and applications, including websites, cloud computing, and large-scale data processing [16].

The primary purpose of a data center is to ensure the continuous operation of its hosted computer systems, which may include servers, networking equipment, and storage devices. More precisely, data centers provide support for business operations around the clock (resiliency), lower total cost of operation and maintenance needed to sustain the business functions (total cost of ownership), and rapid deployment of applications and consolidation of computing resources (flexibility) in a secure and controlled environment.

Data centers can be owned and operated by organizations for their internal use (*e.g.*, *Netflix*, *Uber*, *Meta*) or can be third-party facilities that offer hosting services to multiple clients (*e.g.*, *AWS*, *GCP*, *Azure*).

Moreover, data centers can be classified according to their services. Firstly, there is the typical **cloud** model, in which services (such as compute, storage, and network resources) are available over virtualization (virtual machines or containers). Alternatively, data centers can be used as **supercomputers** in which computing and storage nodes are divided. The access to these systems is bare-metal, and they are typically used to run large-scale applications (*e.g.*, scientific simulations, deep learning, and large-language model training). Finally, data centers can be used as **private infrastructures** (*e.g.*, *Meta*, *Netflix*) with services that serve as a basis for distributed applications (such as messaging, streaming, and gaming).

2.1.1.1 Architecture

The architecture of a data center can vary based on the organization's or service provider's specific needs and requirements. In general, a data center is composed of clusters (typically a single cluster), which have various racks. Each rack includes multiple servers, which in turn have several components such as CPUs, GPUs, disks, and network cards, among others [18]. Figure 2.1 illustrates the typical elements of a data center. In detail:

- **Cluster** — Clusters are groups or configurations of connected computers or servers that work together to perform tasks as a single system.
- **Rack** — A Rack is a standardized framework or enclosure used to house and organize computing equipment. Racks are designed to hold various hardware components, such as **servers**, networking devices, and storage units, in a compact and organized manner. They are a fundamental part of data center infrastructure and are crucial in optimizing space, power distribution, and cooling.
- **Server** — Servers are the computing powerhouses running applications, processing data, and handling requests. They are composed of multiple processors (CPUs), GPU, disks,

and network cards, among others. In modern data centers, various types of servers may be optimized for different tasks, such as storing and managing data, running applications, hosting websites, or delivering various user services.

- **Hardware components** — Such as CPU, GPU, disks, network cards, power supply unit, motherboard, and RAM work together to handle and process data inside a server. In the context of this work, the focus will be on computing resources (CPU, and GPU). A Central Processing Unit (CPU) core is an independent processing unit that can handle tasks. Modern CPUs often consist of multiple cores, each capable of executing a set of instructions concurrently, which allows for parallel processing and improved overall performance. A Graphics Processing Unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images for output to a display. Similar to a CPU, a GPU contains multiple processing units called cores that work simultaneously to handle specific tasks related to rendering and processing graphical data. These cores enable parallel processing, allowing for the efficient handling of complex graphical computations, resulting in faster image rendering, video playback, and overall enhanced graphical performance in applications ranging from gaming to scientific simulations and visual content creation.

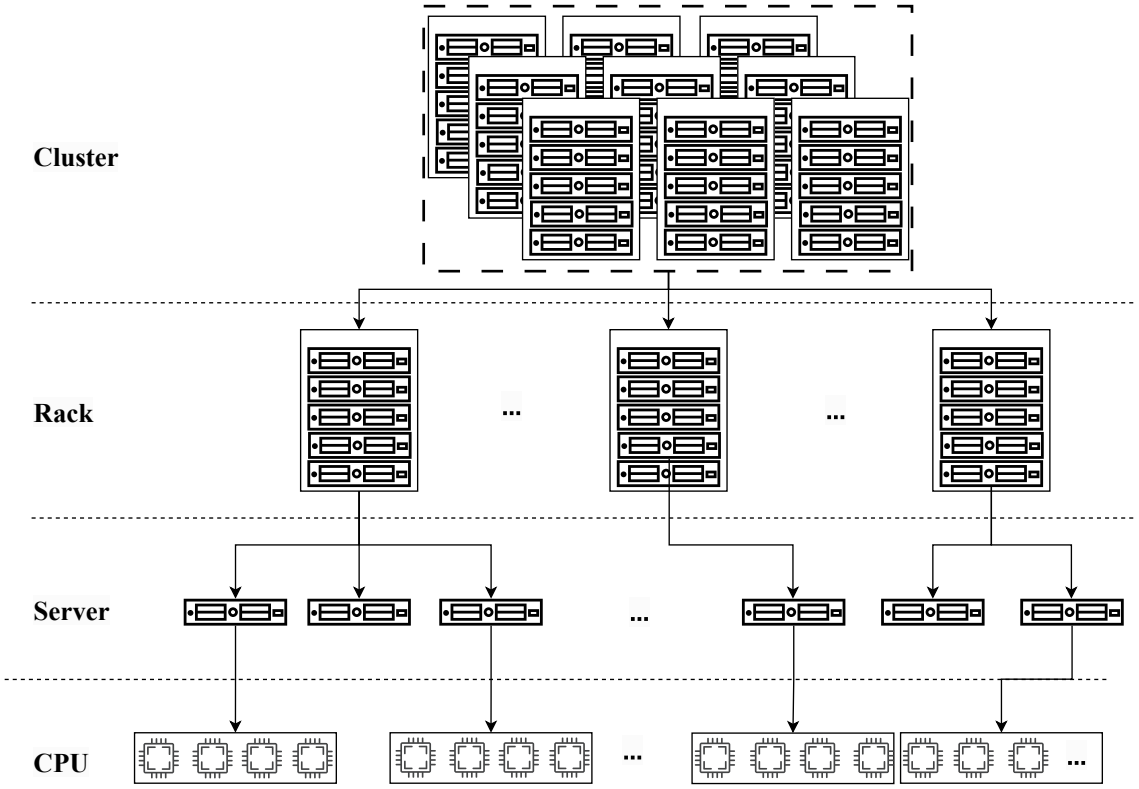


Figure 2.1: Data center architecture abstraction.

This organization constitutes a **hierarchy** as seen in Figure 2.1. However, it is essential to note that other components are crucial to having a working data center (*e.g.*, cooling systems, switches,

routers, disks). Nonetheless, they are orthogonal to this work and, thus, not represented in the illustration.

Although this representation is an abstraction, it is essential to simplify the dimension and complexity of real data centers, as they can be composed of thousands of CPUs in thousands of servers. For example, one can observe the exascale system *Frontier* at the Oak Ridge National Lab, which is composed of 8,699,904 CPU cores [85].

Since these infrastructures can provide different services (actuating as clouds, supercomputers, or application-specific), it is important to understand how they are organized.

Cloud Computing In cloud computing, identical IT resources are grouped and maintained by a system that automatically ensures they remain synchronized, called **resource pools** [38]. The workload (specific set of tasks of various types such as data processing, application execution, request handling, and other operations that require computing resources like CPU, memory, storage, and network bandwidth) is evenly distributed among the resources through a *load balancer* that provides runtime logic and reduces IT resource over- and under-utilization [38].

These resource pools can be:

- **Physical server pool** — composed of networked servers installed with the necessary operating systems and applications and ready for immediate use.
- **Virtual server pool** — composed of one of several available templates chosen by the cloud consumer during provisioning.
- **Storage pool** — composed of file-based or block-based storage structures that contain empty or filled cloud storage devices.
- **Network pool** — composed of different preconfigured network connectivity devices.
- **CPU pool** — composed of CPUs that can be allocated to virtual servers and are typically broken down into individual processing cores.
- **Memory pool** — composed of physical RAM, which can be used in newly provisioned physical servers or to scale physical servers vertically.

Dedicated pools can be created for each type of resource, and individual pools can be grouped into a larger pool. These pools can also be created for specific cloud consumers or applications and are isolated from one another so that each cloud consumer is only provided access to its respective pool [38]. The primary rationale behind organizing IT resources into pools is to avoid resource fragmentation and maximize utilization. By grouping identical resources together, cloud computing systems can dynamically allocate resources where they are most needed, ensuring efficient use.

Since resource pools can become highly complex, with various pools created for specific cloud consumers or applications, they are usually organized in a hierarchical structure that represents parent, sibling, and nested pools. Figure 2.2 illustrates a hierarchical representation of three pools.

In cloud computing, resource pools are utilized by users through a self-service model where clients access and allocate these computing resources via a web-based interface or API. Users can provision, configure, and manage these resources according to their specific needs without direct control over the underlying infrastructure, enhancing scalability and flexibility in deploying computing resources [38]. Given that the resources are shared among multiple users, allowing them to run their applications and store data on the same underlying infrastructure, these data centers are often referred to as *multi-tenant*.

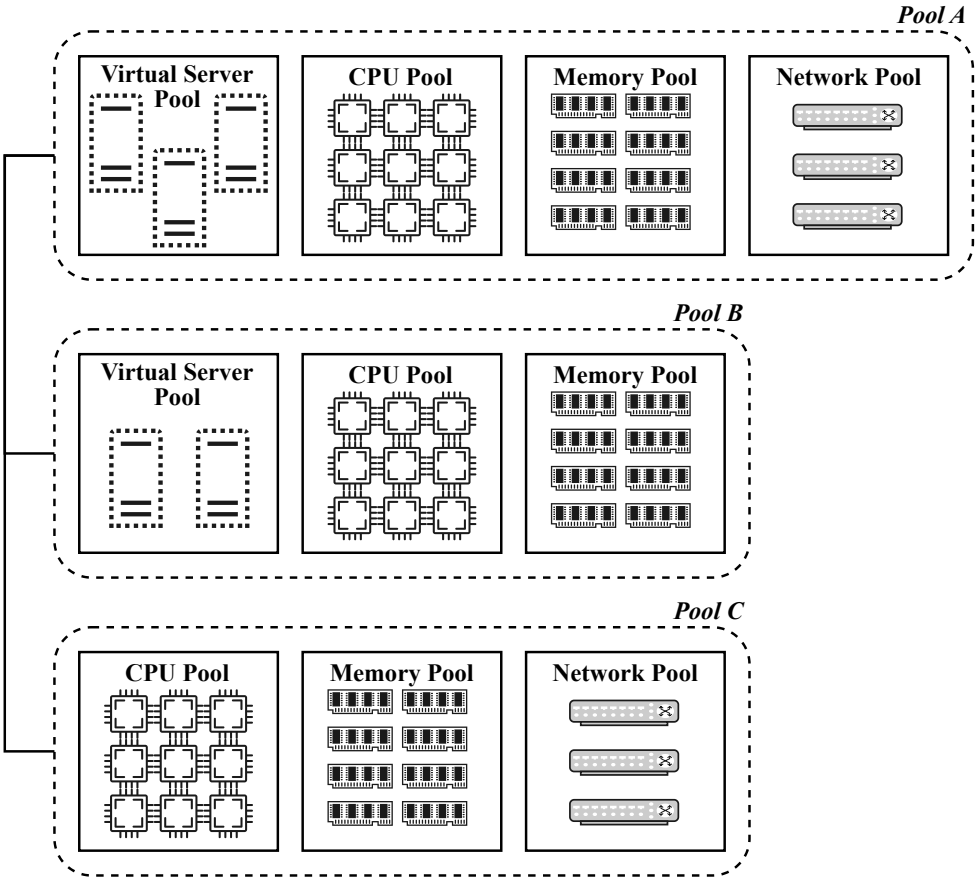


Figure 2.2: Hierarchical representation of pools B and C, which are sibling pools taken from the larger pool A. Adapted from [38].

Once resource pools are established, numerous instances of IT resources can be generated from each pool to form an in-memory collection of active IT resources.

Supercomputing High-performance computing is usually used for computationally intensive tasks. These systems are designed to handle massive amounts of data and perform calculations at an extraordinary rate.

As shown in Figure 2.3, the computing system (the core unit of a supercomputer, designed to achieve ultrahigh computing capability) is composed of **computing nodes**, which store and retrieve data to the **storage nodes** [66].

Compute nodes consist of many CPUs and GPUs and execute computational tasks by leveraging many-core processors that deliver massive parallelism and vectorization. Moreover, storage nodes include one or two CPUs and many disks, and persist applications' data in a shared parallel file system (*e.g.*, PVFS [23], GPFS [17]) that offers high-performance storage and archival accessibility across hundreds of storage drives.

The storage nodes can be **metadata** or **data nodes**. The **metadata nodes** are responsible for managing and organizing metadata information related to the stored data within the storage nodes. They are crucial in facilitating efficient access, retrieval, and manipulation of data stored across the high-performance parallel file system.

As supercomputers aggregate and isolate these components, they behave more like a single-node computer than a distributed system [55].

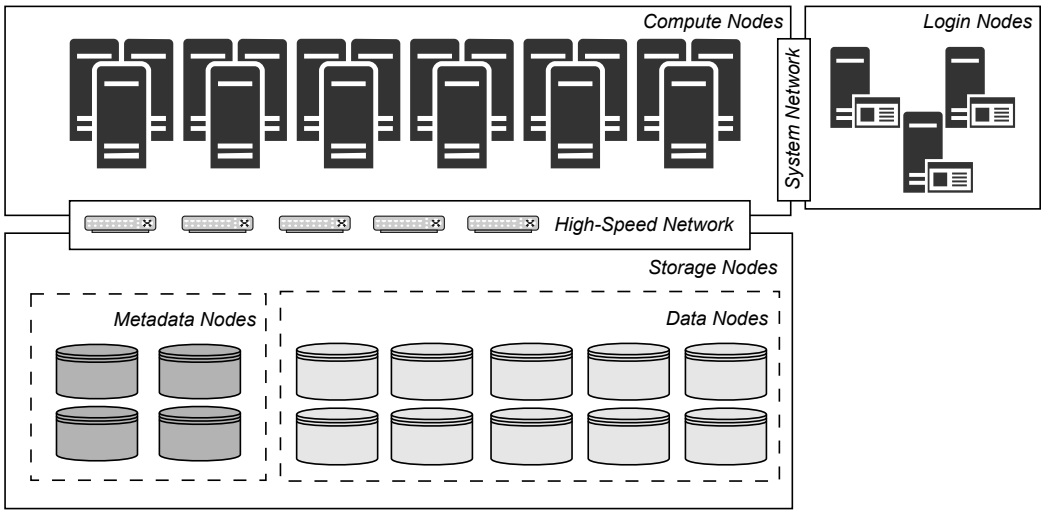


Figure 2.3: Example of an architecture of a supercomputer.

As an example, one can observe the architecture of the *TaihuLight* supercomputer, which is composed of 40,960 compute nodes (over ten million CPU cores in total), 288 storage nodes, and 2 metadata nodes [94]. These values reveal the dimension and complexity of these infrastructures.

Application-specific data centers While cloud and supercomputing systems are designed with a general-purpose approach to serve a wide range of applications and services, application-specific infrastructures are constructed to provide a single service. These services serve as a basis for distributed applications and are private for specific enterprises (*e.g.*, *Netflix, Meta*).

Since these data centers integrate particular contexts, the design and architecture are tailored to the exact purpose of the data center. Accordingly, these infrastructures have different designs, and the concept of "compute nodes" does not fit in this context as there is no distinction between compute and storage nodes. Nodes can be seen as compute and storage nodes at the same time. The data center's design is constructed considering the best for the particular service provided.

Although designed to operate on standard hardware, enterprise-level infrastructures can consist of hundreds to thousands of interconnected storage servers utilizing dedicated network links. Each server is equipped with multiple multicore processors and storage drives organized in a hierarchical structure.

Even though data centers can be classified as *cloud*, *high-performance computing*, or *application-specific* according to their services, their architecture follows the same principles and can be abstracted as illustrated in Figure 2.1.

2.1.1.2 Power Distribution

The power system within a data center is complex, with power losses resulting from multistage voltage conversions.

Data centers usually connect to the electrical grid for their primary power source. This power is received at high voltages and is transformed down to lower voltages using transfer switches. Then, the power is channeled through Uninterruptible Power Supply (UPS) systems, which are typically connected to the utility supply and backup generators, acting as a safeguard against grid failures and providing continuous power, thereby maintaining the reliability and uptime of the data center.

Power Distribution Units (PDUs) further streamline the process, delivering power to server racks, networking equipment, and cooling systems. These PDUs enable the precise allocation of power to individual hardware components within the data center and adjust the voltage to meet the specific requirements of individual computing units, allowing for scalability. These units are distributed across the data center in a wrapped topology [83]. The power is provided to each server through Power Supply Units (PSUs) located in the racks, as shown in Figure 2.4.

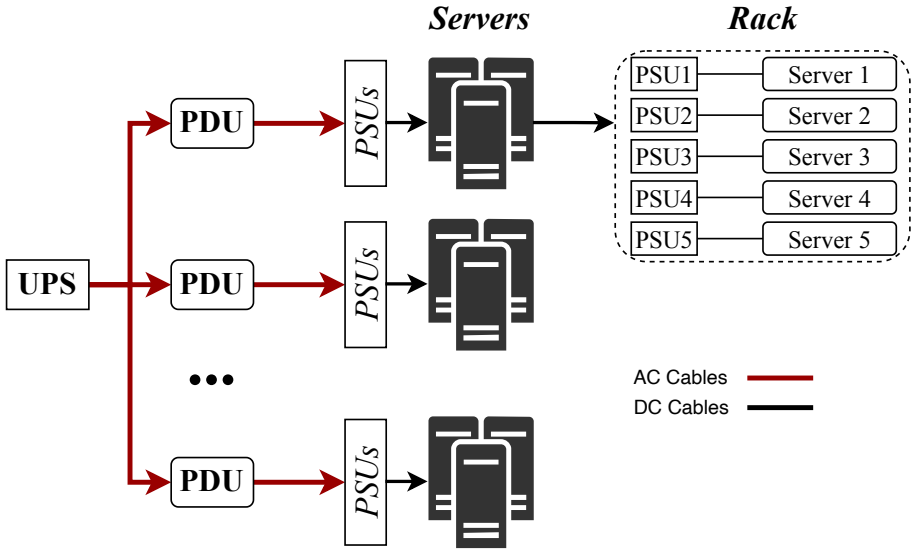


Figure 2.4: Example of power distribution in a data center. Adapted from [11].

Table 2.1: Power consumption proportionality in data centers. Based on [11].

Resources	Power Consumption (Proportionality)
IT equipment (CPU, storage, and others)	45%
Cooling loads	38%
Power conditioning equipment	8%
Network equipment	5%
Building switchgear and Protection devices	3%
Lighting	1%

To distribute the power through these components, setting the power budget for every server within the PDU, the power-scheduling method known as *power routing* is applied. If a server's power requirements exceed its allocated limit, the power routing system searches for extra power from the connected power feeds assigned to that server. Conversely, the power routing controller lowers power limits for underutilized servers, generating surplus power reserves that can be redistributed to other areas within the power distribution system.

2.1.1.3 Energy Consumption

Nowadays, HPC systems and data centers significantly contribute to global electricity consumption. For instance, the exascale system *Frontier* at the Oak Ridge National Lab consumes 20MW of power in continuous operation, while the *Aurora* system at the Argonne National Lab is estimated to draw 60MW [25]. Projections from general studies indicate that data centers alone are anticipated to consume between 8% to 13% of the world's total electricity by 2030 [25].

Furthermore, by comparing the estimated carbon footprint of the different hardware components (GPU, CPU, DRAM, and Storage), it was found that GPUs and CPUs have a significantly higher carbon embodied footprint than the other components [25]. As referred in previous works, at its peak, the CPU accounts for nearly 50% of the system power, but this percentage decreases to below 30% during periods of low activity, making it the most energy-proportional of all main subsystems [18].

As data centers are composed of many resources (servers, networks, storage, and cooling devices), it is important to understand how energy consumption is distributed among these components. Generally, 38% of the total data center power is utilized in the cooling system (*e.g.*, cooling tower, fans, compressor, chiller), 8% in the power system (power consumed in inefficient distribution and site equipment), and 50% in computing devices (*e.g.*, server, storage, network) [83]. More recent studies [11] confirm that these values remain the same. More precisely, as shown in Table 2.1, within the computing devices, network equipment accounts for 5%, lighting 1%, building and protection devices 3%, and IT equipment (CPU, storage, and others) 45%.

Given the data center hierarchy abstraction provided in Figure 2.1, one can reason about the different energy consumption values throughout the hierarchy levels related to CPU usage. The energy consumption of the entire data center (or cluster) will be the sum of the energy expended by all its elements. Additionally, the power consumed by a cluster will be the aggregated value

of the energy consumed by its racks. On each rack, the total energy can be obtained by summing the power consumed by all the servers, which can get their total energy by aggregating the power expended by all the CPUs. This energy distribution follows the hierarchy described in Section 2.1.1.2, in which it is important to consider the maximum physical power limits of the hardware components. Excessive power can lead to increased heat generation, causing the hardware components to overheat. This can result in thermal throttling (reduced performance to prevent damage) or, in severe cases, thermal shutdowns or permanent damage to the hardware. Besides, supplying too much power can cause electrical components to exceed their voltage or current ratings, leading to potential short circuits or component failure, and the hardware may become unstable, leading to system crashes and data corruption.

Although this work focuses on improving the energy consumption of computing devices within large-scale computing infrastructures, especially regarding the servers' resources, one must acknowledge the large amount of energy expended in the rest of the data center's resources (network, storage, cooling). Improvements in the energy consumed by cooling, storage, communication, and power systems are orthogonal to this work.

2.1.2 Controlling Energy Consumption

The optimization of energy consumption within large-scale computing infrastructures has remained a primary focus, with respect to cost-saving measures. Over recent years, diverse solutions have emerged, employing various approaches to effectively control the energy consumption of data centers. These approaches encompass a wide spectrum, varying in their applicability (namely, hardware-based or software-based control), the level of actuation (global or local), the policies applied, and the frameworks and methodologies used, among others [77].

Among these strategies, several stand out prominently as key techniques for energy optimization: powering off the underutilized resources [83, 64, 29], implementing Dynamic Voltage and Frequency Scaling (DVFS) [98, 56, 83, 29], applying "race to halt" methods [83, 29], enhancing hardware for improved energy efficiency [83, 63], and employing workload consolidation practices [76, 48].

One of the proposed techniques consists of **powering off** the underutilized resources, which stands as a dynamic resource sleeping mechanism [83, 64, 29]. This technique is based on the fact that, in some data centers, the average workload usually remains significantly lower than peak performance (in some data centers, it averages at 30%) and does not require the functioning of all computing resources [83]. Turning off or putting into a sleep state the underutilized resources (*e.g.*, servers, processors, disks, chipsets) makes it possible to achieve energy efficiency while fulfilling the data center workload demands [64]. However, while such a technique enables saving considerable amounts of energy, it can comprise the performance of the overall infrastructure under burstiness or unusual periods of high load.

Moreover, implementing dynamic speed scaling techniques, such as **Dynamic Voltage and Frequency Scaling** (DVFS), can effectively limit a server node's energy consumption by reducing the processor's frequency [98, 56, 64, 46, 29]. DVFS involves modifying the power and speed

of a computing device's processors by adjusting either the supply voltage or clock frequency [64]. This adjustment leads to decreased computation cycles and subsequently lowers average energy consumption [19]. DVFS can be employed at the CPU level and extended to assigning distinct frequencies to different CPU cores [46]. Nonetheless, it is important to note that reducing computation cycles may potentially impact the performance of applications. Reducing the clock frequency by employing DVFS might not be advisable when optimizing resource utilization. Conversely, when the objective is to minimize energy consumption, DVFS emerges as a favorable choice.

The Linux kernel provides DVFS-based mechanisms, such as CPUFreq [2]. CPUFreq is a subsystem that enables system administrators and system designers to change the clock frequency of CPU cores. It provides the notion of scaling governors, which are predefined algorithms that set the frequency of each (or all) CPU cores according to a given objective (*e.g.*, power requirements, application workload, etc). The Linux kernel supports a variety of CPU frequency scaling governors, including *Performance*, *Powersave*, *Userspace*, and *Ondemand*. *Performance* is the governor that runs the CPU at the maximum frequency, enhancing application performance but consuming more energy. *Powersave* is the opposite, setting the CPU core to the minimum frequency, spending the least amount of energy, but limiting application performance. *Ondemand* adjusts the frequency dynamically based on the current load. It initially ramps up to the highest frequency, then reduces if idle time increases. This is the governor applied by default. Finally, *Userspace* enables users to set frequencies as they wish.

Scaling drivers communicate directly with the CPU to implement the frequencies specified by the current governor [2]. These drivers handle the CPU-specific processes for setting frequencies as the governor directs. According to the Advanced Configuration and Power Interface (ACPI) standard, Performance states, or P-states, are the standard supervisory software method for communicating DVFS settings [1, 2]. These states represent different combinations of power and performance levels that a processor can operate within, allowing for efficient power management by adjusting both voltage and frequency. However, in practical applications, processors often allow for the specification of particular frequencies rather than being confined to predefined P-states. Scaling drivers manage these specific frequency adjustments. On the same note, Intel's Running Average Power Limit (RAPL) [7] combines these techniques while also allowing monitoring of the power consumption of various components within a CPU, including the package, cores, and DRAM. In detail, RAPL is an advanced power control mechanism that provides real-time power consumption monitoring and power-limiting capabilities to enforce power limits and optimize performance-per-watt metrics. DVFS can also be applied to GPU devices. A popular implementation is provided by the NVML framework from NVidia, which enables users to set custom clock frequencies at which the GPU operates [5]. However, this scope is orthogonal to this work.

DVFS and powering off the underutilized resources can be combined in "**race to halt**" techniques [83, 29, 37], which consist of executing a task at the highest operational frequency and sleep at completion.

Furthermore, the **hardware** can also be improved to consume less energy. Over time, some hardware components were replaced with more energy-efficient ones. For example, RISC archi-

tectures are more energy efficient than their counterpart CISC, as RISC-based processors have fewer transistors and gates. Besides, when comparing x86 with ARM processors, processors utilizing the ARM architecture demonstrate an energy efficiency that surpasses x86-based processors by a factor of three to four [83]. Based on ARM-based processors, the System-On-Chip (SoC) design consumes even less energy (35% less) than RISC architectures [83]. Another example is using phase-change memory (PCM drives) in storage systems, which has also been proposed as an energy-efficient main non-volatile memory alternative.

Another important strategy for minimizing energy consumption is applying **workload consolidation**. This technique involves raising server utilization and minimizing the number of servers needed for a particular set of workloads (*e.g.*, VMs, containers) [76]. By consolidating multiple workloads onto fewer servers or hardware resources, data centers can optimize the usage of available computing capacity [48]. This approach involves dynamically reallocating resources based on workload demands, ensuring that servers operate closer to their maximum capacity, thereby reducing the overall energy footprint of the data center [48]. Additionally, workload consolidation facilitates better resource management, improving operational efficiency and cost savings in data center operations.

In addition to these techniques, one can come across the concept of *overprovisioning* [43]. Given that even in highly-tuned clusters running a single workload, instances of peak utilization are infrequent and consistently remain below the allocated power capacity [74], researchers proposed oversubscribing power circuits to squeeze maximum value from the infrastructure (an overprovisioned data center has more nodes than a conventional data center with the same power budget). To guarantee availability, these proposals utilize power capping, which limits server performance when there are spikes in utilization, ensuring safe power limits. However, this technique focuses more on maximizing the infrastructure's efficiency (resource utilization) and not on controlling energy consumption.

2.2 Related Work

This section provides an overview of previous works that are more relevant in the context of this research. By leveraging the existing energy consumption control systems and control algorithms for large-scale infrastructures, one can reason about different problems, solutions, and respective advantages and limitations of current solutions.

2.2.1 Energy Consumption Control Systems

Large-scale infrastructures are actively looking for ways to limit their energy consumption and CO_2 emissions. Researchers have focused on achieving this purpose over the past few years by concentrating on different problems and using various solutions. These solutions can be categorized based on the granularity level in which they actuate (at the server level, data center level, or different levels). Nevertheless, it is also important to analyze the applied techniques (*e.g.*, DVFS,

workload consolidation) and the type of services provided by the target data centers (clouds, supercomputers, application-specific).

During the first decade of the 21st century, researchers have proposed several approaches to control energy consumption in different contexts. Some solutions focused on local resource management (at the server level) [20, 29, 37, 42, 61, 72, 81], and others on distributed resource scheduling (at the cluster level) [26, 36, 29, 37, 40, 47, 72, 75, 79]. Additionally, part of the solutions applied DVFS techniques [20, 21, 29, 35, 37, 40, 42, 53, 61, 72, 79, 81, 91], workload consolidation [26, 36, 29, 47, 61, 72, 75], powering off the underutilized resources [24, 29, 37, 47, 61, 72, 75], and hardware modifications [24, 79] as one can see in Table 2.2.

Deploying all these power management solutions simultaneously can pose unpredictable challenges due to their interference (*e.g.*, between systems that apply the same technique but for different metrics). In 2009, *Ramya Raghavendra et al.* [78] proposed a solution that splits the control between hardware and software while also imposing a hierarchy control (*i.e.*, at local and global level) to solve this challenge. It was designed a coordination framework consisting of five individual solutions: an efficiency controller (EC), which optimizes average power consumption per server by adjusting the processor P-state based on past resource utilization and estimated future demand, a server manager (SM), which enforces thermal power capping at the server level, reducing the P-state if a specified power budget is exceeded, an enclosure manager (EM) and group manager (GM), which extend this control to blade enclosures and data center levels, dynamically redistributing power to maintain group power budgets, and a virtual machine controller (VMC), which consolidates workloads and powers off unused machines to reduce overall power consumption. System designers or data center operators may establish the power budgets at the EM and GM by considering thermal budget constraints. Alternatively, high-level power managers can determine these budgets. Each blade's SM controller utilizes the lower value between the power budget suggested by the EM and its own local power budget as a reference input. The allocation of the total enclosure power budget to individual blades follows policy guidelines, allowing for the implementation of various policies (such as fair-share, FIFO, random, priority-based, and history-based). The work exemplifies the implementation of a proportional-share policy, where within each interval, the policy redistributes the total power allocation within the enclosure/group among its components based on their power usage from the previous interval, ensuring a fair share of the budget that can adapt to changing demands. Additionally, the interaction between layers primarily occurs through power budget configurations and the measurement of consumed power. The group manager (GM) enforces power capping at the rack or data center level, utilizing distinct time constants. It compares the actual power usage of the group to the assigned budget and distributes these budgets to the next-level servers and blade enclosures. Allocation decisions within the servers and enclosures prioritize the minimum value between the GM's recommendation and local budget values. However, aiming to create a versatile solution supporting all past solutions, this solution focused on developing a sophisticated architecture and led to the absence of a consistent prototype applicable in real-world scenarios. Though the approach seems promising, assessing its effectiveness in real-world scenarios is challenging due to its application in a

simulated context.

In the same year, *Wang et al.* [88] addressed the challenge of scalability in power control, contending that prevailing power control strategies lacked scalability for managing the power consumption of an entire large-scale data center. The limitations stemmed from their initial design for individual servers or rack enclosures. The significance of regulating energy at three distinct levels - rack enclosure, power distribution unit (PDU), and the entire data center - was emphasized, acknowledging each tier's physical and contractual power constraints. The solution is a scalable hierarchical power control architecture for large-scale data centers called *SHIP* [89]. For each mentioned level, there are specific power control solutions. The rack-level power controller adaptively manages the power consumption of a rack by manipulating the frequency scaling setting of the processors of each server in the rack, the PDU-level power controller manages the total power consumption of a PDU by manipulating the power budget of each rack, and finally the data center-level controller manages the total power consumption of the entire data center by manipulating the power budget of each PDU. The rack-level control loop (invoked periodically) is composed of a power controller (computes the new CPU frequency level for the processors of each server and then sends the level to the CPU frequency modulator) and a power monitor (measures the average value of the total power consumption of all the servers and sends the value to the controller). Each server has a CPU utilization monitor (sends the CPU utilization to the controller) and a CPU frequency modulator (changes the CPU frequency level of the processors accordingly). On the other hand, the PDU-level and data center-level power control loop also includes a power controller and a power monitor at the respective levels and the power controllers and utilization monitors of all the lower-level components. The responsibilities of each component are divided similarly to the rack-level power control loop. In each interval, the power budgets of the lower-level control loops for all the entities change. Emphasized is the significance of minimizing the impact on the stability of a rack-level control loop. This is achieved by choosing a control period for the PDU-level loop that surpasses the settling time of the rack-level control loop. The same principle applies to the power control loop at the data center level. In addition, the system assumes three constraints of the higher-level control loops: the power budget of each rack should be within an allowed range (based on the maximum and minimum possible power consumption of each server), power differentiation can be enforced for two or more racks (racks may have different priorities), and the total power consumption should not be higher than the desired power constraint. While resembling the proposed system and covering most requirements, *SHIP* focuses on specific policies and algorithms for distributing power consumption across the data center hierarchy. However, it may benefit from exploring a more diverse range of policies. Additionally, *SHIP* could consider aspects such as power variation and coordination among multiple controller instances.

With the adoption of *overprovisioning* power circuits to squeeze maximum value from the infrastructure, *power capping*, or peak power management techniques, have been proposed [92]. Although these techniques continually monitor server power consumption and use DVFS to suppress it if it approaches its power or thermal limit, they are applied to server-level power management, with control actions decided locally in isolation. To solve the problem of data center-wide

power management *Qiang Wu et al.* implemented *Dynamo* [92]. *Dynamo* is a power management system that monitors the entire power hierarchy and ensures safe and optimized utilization of allocated power within *Facebook*'s data centers. It focuses on the fact that in real-world data centers, different power and performance constraints at different levels in the power hierarchy require coordinated, data center-wide power management. To address this challenge, *Dynamo* has two major components: agents (programs deployed to every server in the data center that read power, execute power capping/uncapping commands, and communicate with the controllers) and controllers (that run on a group of dedicated servers, monitor data from the agents under their control, and are responsible for protecting data center power devices). In *Dynamo*'s design, each physical power device within the hierarchy requiring protection corresponds to a dedicated controller instance overseeing and managing the set of downstream servers associated with that device. This establishes a hierarchy of controllers mirroring the topology of the data center's power hierarchy. Each power device at the lowest level is assigned a leaf power controller. This controller establishes direct communication with agents across all downstream servers associated with the respective power device. Multiple tiers of controllers collaborate to safeguard the overall integrity of the power hierarchy.

During each sampling interval, a dedicated leaf controller is responsible for reading and consolidating power data from all downstream servers. Subsequently, this controller compares the aggregated power consumption with the predefined power limit of the device. The decision-making process (implemented as the algorithm described in Section 2.2.2) determines whether to implement power capping or uncapping measures (done through RAPL). RAPL (essentially based on DVFS [92]) is a power control mechanism at the individual server level, designed to enforce the overall system power budget. When power capping is deemed necessary, the leaf controller chooses the optimal set of servers to cap in order to minimize performance impact. This optimization aims to minimize the overall performance impact induced by the power capping, ensuring an efficient balance between power constraints and system functionality. Likewise, every power device situated at a non-leaf level is designated an upper-level power controller. This upper-level controller indirectly oversees and manages its downstream servers through communication with leaf controllers or other upper-level controllers situated downstream and shares the same functions as the leaf controller regarding power reading and aggregation and capping and uncapping decisions. The communion between controllers guarantees the convergence of decisions and safeguards the well-being of upper-level power devices (as described in Section 2.2.2). Although *Dynamo* has proven effective, it was designed to be applied to application-specific (owner-operated) data centers. Thus, the system can not be used as effectively in a cloud (multi-tenant) data center as the leaf controllers do not possess the meta-data information regarding the servers' priorities to choose the optimal set of servers to cap to minimize performance impact. Furthermore, the finest granularity level at which the system actuates is at the server level. *Dynamo* currently does not offer the capability to control the energy consumption of individual applications.

Table 2.2: Related energy consumption control systems classification.

Work	Technique	Granularity Level	Key Insight
[20]	DVFS	Server	Quantify the potential benefits of DVFS
[21]	DVFS	CPU	Dynamic thermal management for high-performance microprocessors
[24]	Hardware modifications Power resources off	Disk	Conserve disk energy in high--performance network servers
[26]	Workload consolidation	Cluster	Manage energy and server resources (server allocation and routing of requests) in hosting centers
[36]	Workload consolidation	Cluster	Energy-conscious server switching and deactivation
[29]	DVFS Power resources off Workload consolidation	Server Cluster	Server provisioning, DVS control and the costs of server shutdowns
[35]	DVFS	CPU	DFVS for multicore thermal management
[37]	DVFS Power resources off	Server Cluster	Combinations of DVS and node vary-on/vary-off
[40]	DVFS	Cluster	Estimate potential power and energy savings of power capping and DVFS in very large scale systems
[42]	Throttling mechanisms (similar to DVFS)	Server	Power estimation, dynamic power allocation among server components, and throttling mechanisms to regulate component-level activity
[47]	Power resources off Workload consolidation	Cluster	Adjust heterogeneous server cluster configuration and request distribution to optimize power
[53]	DVFS	CPU	Coordinated DVFS scheme for chip multiprocessors
[61]	DVFS Power resources off Workload consolidation	Server	Energy management and conservation for commercial servers using their architecture and workload characteristics

Continued on next page

Table 2.2 – continued from previous page

Work	Technique	Granularity Level	Key Insight
[72]	DVFS Power resources off Workload consolidation	Server Cluster	Coordinate power management in virtualized enterprise systems
[75]	Power resources off Workload consolidation	Cluster	Load balancing and unbalancing to dynamically turn cluster nodes on and off
[79]	Hardware modifications DVFS	Cluster	Ensemble-level power management by using statistical properties of current resource usage
[81]	DVFS	Server	Adaptative algorithms for DVFS in QoS-enabled web servers
[91]	DVFS	CPU	Formal control techniques for power-performance management
[78]	DVFS Power resources off Workload consolidation	Server Cluster	Coordination framework for unifying power delivery and energy consumption control past solutions
[92]	DVFS	Server Cluster	Data center-wide power management system that monitors the entire power hierarchy and makes coordinated control decisions
[89]	DVFS	Cluster	Scalable hierarchical power control at rack enclosure, PDU, and data center levels

2.2.2 Energy Consumption Control Algorithms

Energy consumption control algorithms are computational techniques and strategies designed to optimize and regulate energy usage in various systems and devices. These algorithms are crucial in managing energy consumption efficiently, reducing costs, and minimizing environmental impact. Within large-scale computing infrastructures, energy consumption control algorithms are responsible for deciding the amount of energy (or, at least, the threshold) to be spent by the different resources. In the past few years, several algorithms and policies have been proposed that guarantee different ways of controlling energy consumption in data centers.

The *Tracking-only* policy [27] regulates data center power through job scheduling and power-capping to match the actual power consumption with the target power. However, it does not include job QoS in its control decisions and views the target power as a hard limit that will never be surpassed.

On the other hand, the *EnergyQARE* policy [28] is similar to *Tracking-only* but also considers QoS degradation (*i.e.*, workload QoS constraints determined by users or Service Level Agreements - SLAs). The power consumption limit is imposed as an upper bound. Still, if the average QoS degradation is too high, more servers are activated to run more jobs regardless of the target power.

In 2021, *Zhang et al.* [96] proposed a power management policy that enables data centers to schedule computing jobs and applies server power-capping, ensuring power and QoS constraints. The policies above are considered insufficient, as HPC systems run multi-node jobs with long execution times, and they do not explicitly target these jobs, relying mostly on job scheduling to control power. Job scheduling does not provide sufficient control points for long-duration, uninterruptible jobs. Consequently, the *QoS-aware-Capping* policy was proposed to adjust server power caps considering the estimated QoS of jobs at run time. The policy calculates, for every job, an estimated QoS degradation (based on the job's minimum execution, waiting-in-queue, elapsed, and remaining times) that enables the system to predict which job's QoS degradation will exceed the threshold. Based on the QoS degradation metric, the policy prioritizes tasks with QoS nearing violation, and for others, it favors those with higher average QoS degradation. When power is limited, it lowers power caps only for tasks meeting their QoS requirements, *i.e.*, tasks waiting in the queue or running with low power caps will likely experience worse QoS degradation and receive higher priority.

All of these policies are based on the premise of running additional jobs and raising CPU power limits when the current energy consumption falls below the target limit. Conversely, when the current power exceeds the target, these policies often hold off on starting new jobs and reduce the CPU power caps. Furthermore, the average and the variation of the target power limit are obtained through another policy. The *Adaptive Bidding* policy configures these parameters by making power decisions based on the current workload and pending jobs instead of depending on a data center's long-term average usage (as in previous works [27, 28]). This holds significance as data centers running frequent, extensive, and large-scale multi-node jobs often experience significant fluctuations in power demands during specific periods compared to the general long-term average. Even though the presented work offers valuable insights into a robust energy-controlling policy, it was designed to actuate at the server level, not exploring how to apply these policies across the data center hierarchy. Given that a data center is composed of clusters of servers, it would be interesting to explore how the energy consumption should be distributed across all these nodes. Actuating only at the server level and neglecting the overall demands, this solution can compromise the performance of jobs running in different nodes. Moreover, since the mentioned policies are job-oriented, they do not ponder, for example, the energy consumption of idle servers, which is an important consideration [69]. Besides, the presented policy was evaluated by simulation, making the assessment of its effectiveness in real-world scenarios challenging.

Dynamo [92] also applies an energy consumption control algorithm at the leaf power controllers. After aggregating server power readings, the leaf power controller computes the overall power consumption of a power device and then compares it to the limit of the power breaker. The decision is made by employing a three-band algorithm. The algorithm establishes a capping

threshold typically set at 99% of the breaker limit. When the combined power surpasses this threshold, the power capping logic reduces consumption to the capping target. The capping target is deliberately set 5% below the breaker limit for safety. An uncapping threshold is implemented to prevent oscillations, ensuring that power uncapping only occurs when the aggregated power falls below this level. The algorithm demonstrates flexibility by allowing the configuration of capping and uncapping thresholds on a per-controller basis, facilitating tailored trade-offs between power efficiency and performance across various levels of the power delivery hierarchy.

The system also focuses on ensuring performance-aware power capping, *i.e.*, minimizing its impact on application performance. The leaf power controller is crucial in selecting the appropriate servers to cap and determining their capping targets, considering metadata and power consumption history. Facebook services are categorized into priority groups, each with its own service-level agreement (SLA) regarding the lowest allowable power cap. Higher priority (*e.g.*, cache servers) corresponds to potentially higher performance impact due to power capping. When power capping is necessary, the controller calculates the total power cut and systematically distributes it across servers in ascending priority groups. Within each priority group, a *high-bucket-first* algorithm is employed to allocate the power cut based on current power consumption. This approach punishes servers with higher power consumption, addressing potential issues such as application regressions or unexpected software behavior. The leaf power controller adheres to predefined bucket sizes (set at 20 W) to evenly distribute power cuts among servers within a bucket. After calculating the allocated power cut for each server, the controller determines the power cap by subtracting the power cut from the server's current power value. Ultimately, the leaf power controller issues capping requests, including the designated capping targets, to all affected servers.

Since the upper-level power controllers share the same functions as the leaf controllers and each possesses their own autonomous capping/uncapping logic, it is imperative for them to coordinate effectively, ensuring that their respective capping actions do not conflict or result in negative interactions. To address this challenge, *Dynamo* employs a *punish-offender-first* algorithm. When the upper-level power controller surpasses its power limit, initiating the capping process, it evaluates its subordinate controllers and prioritizes addressing the offenders. In this context, an offender is identified based on the power quota (the child consuming more power). When choosing between multiple offenders, the upper-level controller would distribute the power cut among all downstream offenders using a *high-bucket-first* algorithm. Ultimately, each controller selects the minimum value between its individual capping decision and the decision propagated from its parent controller.

As the algorithm focuses on ensuring safe power distribution and increasing power capacity optimization, centering its design on physical power devices (and their limits), it does not solve the problem of limiting energy consumption. With *Dynamo*, the capping actions are only triggered when the energy consumption surpasses the breaker limit. Envisioning, for instance, a data center powered by diverse energy sources, including both renewable and non-renewable options, limitations arise within *Dynamo*. For instance, if an administrator aims to curtail the energy consumption of the data center on a day when renewable energy is not available, *Dynamo* lacks the

capability to facilitate this restriction. Similarly, if the intention is to constrain the consumption of a specific rack hosting less critical services, *Dynamo* does not provide the functionality for such fine-grained control.

On the other hand, *Patel* and *Tiwari* [73] focused on fairness and efficiency of power management of large-scale computing systems by implementing *PERQ*. *PERQ* was designed to solve the problem of unfairness among multiple concurrent applications that run on current state-of-the-art data center power management systems and targets two objectives: remain fair to all concurrently running jobs and achieve higher system throughput to make up for the cost of over-provisioning (large-scale computing systems employ hardware over-provisioning because enables the system to concurrently execute a higher number of jobs). To achieve the desired results, *PERQ* applies a feedback-based approach and allocates additional power resources to jobs with a substantial influence on system throughput without affecting the performance of other jobs. This is possible as applications exhibit varying levels of sensitivity to power-capping. Some jobs perform equally well at lower power caps, allowing for a decrease, while others do not. In the latter case, *PERQ* increases the power allocation to maximize the system throughput [73].

As *Zhang et al.*, this approach is job-oriented, not considering the energy expenditure of idle CPUs, and actuates at the server level, not considering the division of power across the entire data center hierarchy. Moreover, it is important to notice that since *PERQ* focuses on improving system throughput, sometimes increases the power allocation, instead of focusing on achieving a more sustainable data center.

Guliani and *Swift* [46] report that current power-limiting mechanisms do not address the difference in power usage and frequency limits across cores. Attempting to solve this problem involved focusing on how a system should allocate different amounts of power to different applications to effect prioritization and isolation. This problem is essential, especially for servers running heterogeneous jobs, since a job may be throttled by another job hitting a power limit. Interference may arise from the shared power constraints among multiple jobs or applications running concurrently on the same server or core within the data center. This interference occurs because when one job consumes more power than allocated or hits its power limit, the system initiates throttling mechanisms, reducing voltage and frequency levels to stay within the power constraints. As a result, this throttling negatively impacts the performance of the throttled job and, in turn, affects the performance of other jobs running on the same server or core due to shared power limitations. Differentiating the power provided to different jobs prevents interference.

The aim was to deliver differential power to applications running on a single server socket by proposing and evaluating both priority-based and share-based policies. While priority policies ensure maximum priority applications run at maximum performance and use residual energy for lower priority applications, which can lead to starvation, proportional-share policies divide power between applications according to configured shares (in a more flexible manner). Within the proportional share policies, the work includes three resources for distribution: *Power*, *Frequency* (the frequency at which applications run should be proportional to the shares they hold), and *Performance loss* (applications with more shares should suffer less performance loss). This work is

essential since it proposes power shares, frequency shares, and performance shares as three alternatives to the current policy of restricting power consumption via an upper-frequency limit. An important conclusion is that although all the alternatives can deliver per-application power, frequency shares were the most stable and provided the best performance isolation. Although the solution actuates at a very sharp granularity level (per core), it is noted that the implementation of the policy should be in hardware, providing a low sampling overhead and a fast response to changing workloads and workload characteristics. Furthermore, *Guliani* and *Swift* note that the proposed policies do not account for applications with performance or power saturation, situations where there is insufficient power to run all applications, and the issue of starvation under space sharing, even when a subset of applications could still run.

2.2.3 Summary

Existing energy control systems typically manage energy expenditure at either a high granularity level, focusing on the entire data center or individual servers without considering the applications running on them, or at a fine granularity level, specifically at the CPU level, controlling individual applications but lacking scalability for the entire data center. Several solutions face challenges in providing a comprehensive abstraction for controlling energy consumption across different large-scale infrastructures and granularity levels. Specifically, few solutions can effectively guarantee a power threshold for the entire data center, specific clusters, racks, servers, or CPUs, or for a particular application simultaneously. Therefore, there is a need for a novel approach that addresses these limitations, enabling the application of diverse control algorithms across all these granularity levels.

Additionally, the monolithic nature of many existing systems, which operate primarily at the infrastructure level, often limits their flexibility in implementing QoS rules for controlling energy consumption in specific case studies. Examples include minimizing energy consumption in disaggregated storage or optimizing energy usage in deep learning training.

In summary, the existing solutions face the following challenges:

- **Focus on increasing the efficiency of using data center power**, which considers the limitations of the physical power devices. The existing solutions do not address the problem of limiting energy consumption but focus on optimizing it to increase the number of tasks, applications, or workloads running in a data center. This focus is important and beneficial for maximizing resource utilization and improving overall performance. However, these solutions do not tolerate scenarios where controlling and limiting energy usage is crucial. Considering a scenario where a data center draws energy from diverse sources, including both renewable and non-renewable, it would be interesting to allow administrators to proactively regulate energy usage, mainly when sourced from non-renewable outlets. In such cases, the problem consists of limiting energy consumption instead of using the energy efficiently.

- **Actuate at very specific granularity levels** and do not have the granularity needed to control the energy consumption of specific applications and, simultaneously, the entire data center. Most solutions actuate (at most) at the server level or at the data center level and are not generic enough to comprise both cases, lacking the flexibility and adaptability required to holistically manage the diverse energy needs and priorities inherent in large-scale computing infrastructures.

- **Apply a specific energy control algorithm** that is predefined. Considering the same scenario referred to in the first problem, there should be the possibility to apply various algorithms for different circumstances and types of services data centers provide. The energy control algorithm applied in an application-specific data center, where all the servers could be limited to the same energy threshold, should not be the same as in a cloud, where some resource pools may have a higher priority.

Chapter 3

FINER: A fine-grained, holistic energy controller

As data centers continue to expand in response to the growing demand for digital services, effective energy management is critical to reduce operational costs and environmental footprint. Controlling and limiting the energy consumption of large-scale computing infrastructures is paramount, and there is a need to tackle challenges not mitigated by existing solutions. These challenges include coarse-grained energy control and partial visibility, which hinder efficient energy optimization. In response, this work introduces FINER, a novel system designed to dynamically manage energy usage across different levels of granularity within the infrastructure. FINER enables precise control over energy allocation, ensuring QoS by adapting to server workloads effectively. For instance, administrators can set energy consumption limits for specific racks and dynamically redistribute unused energy to optimize performance without exceeding hardware constraints. This chapter presents the design and implementation of FINER with a thorough depiction of its architecture and control algorithms.

3.1 Design Principles

FINER 's design is built under four core design principles: decoupled design, fine-grained energy control, coordinated control, and adaptable energy tuning.

- **Decoupled design** — FINER follows a decoupled design which draws inspiration from Software-Defined Networking (SDN) [93, 54] and Software-Defined Storage (SDS) [86, 66] paradigms. This approach separates the control logic (control plane) from the mechanisms that implement it (data plane). The control plane centralizes decision-making processes such as energy consumption policies, resource allocation strategies, and QoS enforcement, while the data plane focuses on executing these policies independently of specific hardware or infrastructure details. This architecture simplifies maintenance, testing, and development by decoupling control from data, allowing for modular updates and enhancements to the control logic without affecting underlying infrastructure components. This design enhances system

flexibility, scalability, and adaptability to evolving workload demands and technological advancements in large-scale computing infrastructures.

- **Fine-grained energy control** — By following a fine-grained energy control, FINER enables the management of energy consumption at multiple levels of granularity within large-scale computing infrastructures. This control includes regulating energy usage at various levels, such as data centers, specific clusters, racks, servers, CPUs, or individual applications. FINER promotes fine-grained energy allocation strategies to match specific requirements and priorities from both applications and hardware. For instance, administrators can enforce energy consumption limits on particular servers or allocate excess energy from underutilized resources to more demanding tasks within the same rack or cluster. This granularity optimizes energy efficiency and enhances performance by ensuring critical applications receive sufficient resources while minimizing waste. Fine-grained energy control thus facilitates efficient resource utilization across diverse workload scenarios, contributing to overall system reliability and sustainability.

- **Coordinated control** — The third principle of FINER is global and coordinated control, emphasizing synchronized management of energy consumption across the entire infrastructure. This approach ensures that energy policies and allocation strategies are globally coordinated, considering diverse applications and services' holistic energy demands and priorities. By maintaining global visibility and control, the system can dynamically adjust energy allocations in real time based on workload fluctuations and current system conditions. This coordinated approach optimizes energy efficiency and enhances overall system performance by preventing resource contention and balancing energy usage effectively. Additionally, coordinated control facilitates proactive decision-making at the cluster or rack level, allowing administrators to implement cohesive energy management strategies that align with organizational goals and sustainability objectives.

- **Adaptable energy tuning** — FINER is built to be adaptable, ensuring a prompt and dynamic response to fluctuations of the workloads or operational scenarios. The system is designed to dynamically adjust energy allocations across various components and levels of the infrastructure, continuously optimizing resource utilization while adhering to predefined energy constraints. By incorporating adaptive algorithms and real-time monitoring mechanisms, the system can promptly respond to changes in workload demands, environmental conditions, or user priorities. This adaptability enhances energy efficiency and maintains overall system performance by allocating resources efficiently where and when they are most needed.

3.2 Architecture

The FINER's architecture is composed of two main functional components: the *control plane* and the *data plane*. The control plane is responsible for all energy control logic, including monitoring,

decision-making, and enforcement of energy policies (identified as P_1, P_2, and P_y in Figure 3.1). It collects real-time metrics, computes energy allocation strategies, and enforces these strategies to ensure efficient energy usage across the infrastructure. Within the control plane, there are several distributed subcomponents that operate throughout the infrastructure. These include *node controllers* at the server level, rack-level controllers, and a global orchestrator that oversees the entire system. Each controller plays a specific role in the energy management process, ensuring that energy policies are applied at the appropriate granularity. On the other hand, the data plane comprises the mechanisms responsible for energy management, such as DVFS. It receives the energy rules from the control plane and applies the necessary changes to alter the energy consumption of the hardware components.

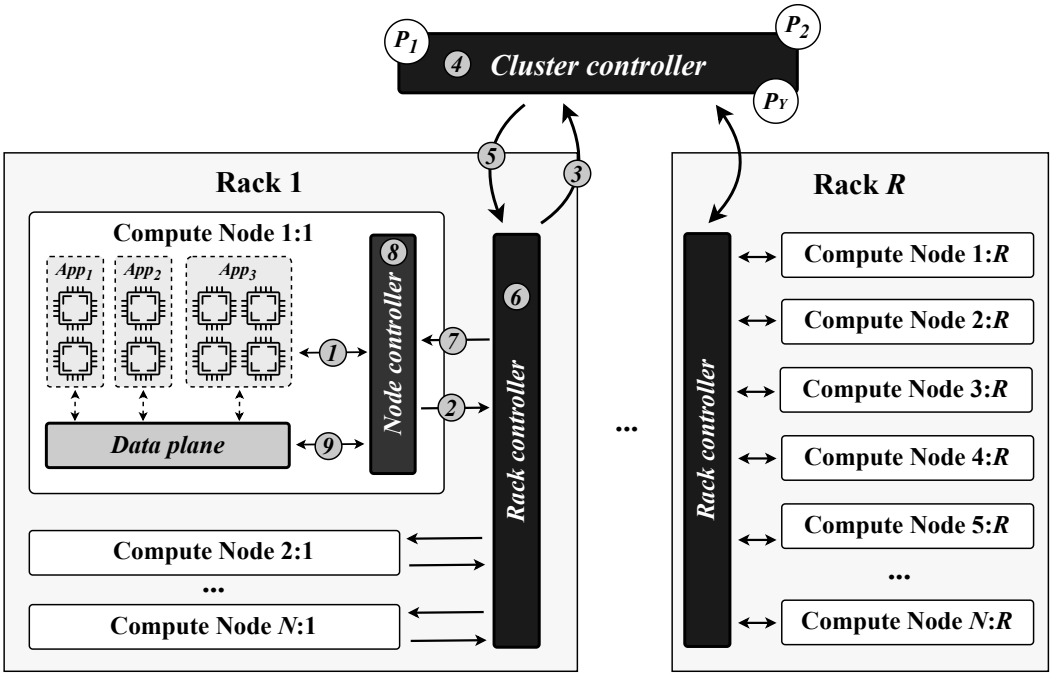


Figure 3.1: FINER high-level architecture.

The system's overall functionality can be visualized through a high-level diagram (Figure 3.1), which illustrates the interplay between the control plane and data plane, along with their respective subcomponents. This figure shows the described architecture with the *cluster controller* managing multiple *rack controllers*, each overseeing several compute nodes within their respective racks. Each compute node contains several CPU cores in which multiple applications are running. These applications are managed by a *node controller*, highlighting the hierarchical control structure from the global to the local level. Along with a *node controller*, each compute node comprises a *data plane stage*, responsible for enforcing the controller's rules to the applications.

The control logic is provisioned through control algorithms that follow a feedback control loop scheme composed of three phases: *collect*, *compute*, and *enforce*. During the *collect* phase (identified by the numbers *1*, *2*, and *3* in Figure 3.1), metrics are gathered by the controllers from

various infrastructure components. In the *compute* phase (identified by the numbers *4*, *6*, and *8* in Figure 3.1), these metrics are analyzed, and optimal energy allocation decisions are made at the control plane level. Finally, in the *enforce* phase (identified by the numbers *5*, *7*, and *9* in Figure 3.1), these decisions are implemented, adjusting the energy consumption of different components to achieve the desired balance between performance and efficiency. This phase involves the control plane and the data plane. Although the control logic adheres to the flow shown in Figure3.1, each controller's *collect*, *compute*, and *enforce* phases operate independently. Consequently, only one cycle occurs at the node-level controller, identified by numbers *1*, *8*, and *9*. At the rack-level controller, the cycle identified by *2*, *6*, and *7* takes place, and at the cluster-level controller, the cycle identified by *3*, *4*, and *5* occurs.

3.2.1 Data Plane

The data plane is a critical system component responsible for managing various hardware components' energy consumption. Its primary functions include enforcing energy policies and interacting with the control plane to receive and execute control commands. The data plane operates directly on hardware components, such as CPUs and GPUs, implementing specific energy management mechanisms such as DVFS. The data plane stages manage hardware components based on predefined rules and real-time inputs from the control plane. Each stage is tailored to a specific type of hardware (*e.g.*, CPU or GPU) and can enforce energy policies by adjusting parameters like frequency and voltage.

Upon initialization, the data plane connects with the control plane to receive control operations and enforcement rules. These operations are processed and executed to ensure that the system adheres to the defined energy efficiency goals. Such interactions involve:

1. *Handshake and Initialization*: The data plane stage initiates a handshake with the control plane, sharing essential information about its environment and capabilities.

2. *Control Operations Reception*: It continuously receives control operations, which include metadata information on the operations to be performed, such as the number of enforcement rules to be received.

3. *Rule Enforcement*: The data plane enforces these rules on the hardware components, adjusting parameters to match the control plane's requirements.

4. *Acknowledgment and Feedback*: After executing a command, the data plane sends acknowledgment messages back to the control plane, indicating the success or failure of the operations.

The data plane exposes an interface to the control plane for effective communication and control. Table 3.1 outlines this interface.

A `stage_info()` call (Table 3.1) provides the information about the data plane stage, including the stage identifier, its type (the hardware components it will control), PID, hostname, and

Table 3.1: Data plane interface.

Interface	Description
`stage_info()`	Provides information about the data plane stage.
`enf_rule(hardware_id, freq)`	Details rules to be enforced on hardware components.

username. It is used at the initialization of the data plane. Such information is essential for the functionality of the control plane. On the other hand, a `enf_rule()` call is used to enforce a specific frequency (`freq`) in a particular hardware component (identified by the `hardware_id`). This call allows the data plane to receive the enforcement rules defined by the control plane.

3.2.2 Control Plane

The control plane is a critical component of the FINER architecture, responsible for all energy control logic, including monitoring, decision-making, and enforcement of energy policies. The control plane operates hierarchically to provide global and local infrastructure control. This hierarchical structure ensures a comprehensive and scalable energy management approach, providing global oversight and local control. Such hierarchy follows the same logic as the data centers' organization described in Section 2.1.1.1. The control plane consists of three levels of controllers (as illustrated in Figure 3.2).

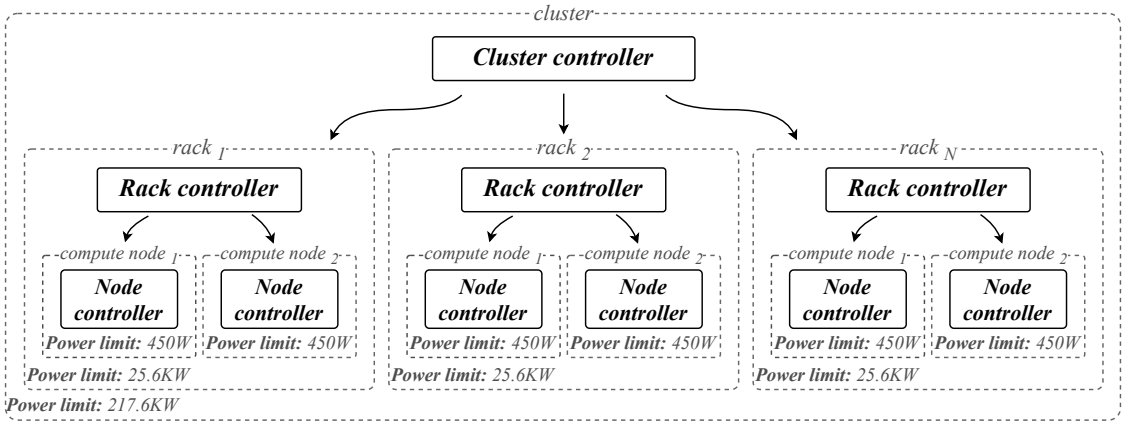


Figure 3.2: FINER control hierarchy.

- **Node Controllers** — At the lowest level, the *node controller* manages individual compute nodes (servers). It collects real-time metrics from the CPUs and other components using a monitoring module, applies local energy policies, and enforces power limits to optimize energy usage at the node level.

- **Rack Controllers** — The intermediate level, the *rack rontroller*, oversees multiple *node controllers* within a single rack. It aggregates metrics from all nodes in the rack and ensures the combined power consumption stays within the rack's power target. This level of control

allows for balancing energy usage across nodes and optimizing the overall energy efficiency of the rack.

- **Cluster Controller** — At the top level, the *cluster controller* provides a global view of the entire data center, managing multiple *rack controllers*. It aggregates metrics from all racks, formulates energy policies for the whole cluster, and ensures that the total power consumption of the data center does not exceed predefined thresholds. This controller allows for strategic decisions that optimize energy usage across the entire infrastructure, ensuring a balance between performance and energy efficiency.

This three-level hierarchy is justified by the need for scalable and efficient energy management in large data centers. The infrastructure's architecture, abstracted as shown in Figure 2.1, aligns with FINER's hierarchical distribution. By distributing the control tasks across these levels, the system can handle the complexity of modern data center operations, providing both detailed control at the node level and broad oversight at the cluster level. Throughout this hierarchical distribution, each level of the controller hierarchy only communicates with the level exactly above and the level exactly below, abstracting from the remaining components of the system as shown in Figure 3.2. Each level of the hierarchy comprises its corresponding physical power limit, which is the maximum allowable power consumption that can be allocated to the components under its control.

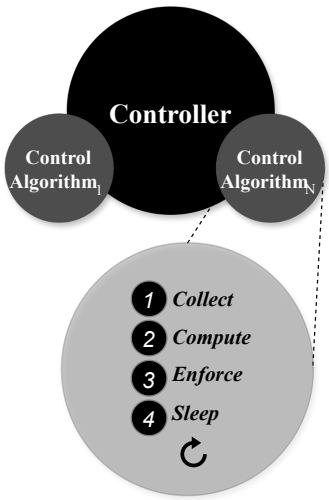


Figure 3.3: Controller Abstraction.

The control plane logic can be abstracted as represented in Figure 3.3, with the applicability of different control algorithms composed of 4 phases. Each algorithm consists of *collect*, *compute*, *enforce*, and *sleep* phases, which operate in a constant *feedback control loop*. Such abstraction enables the definition of multiple control algorithms while ensuring the holistic logic of the core

tasks. The *collect* phase of the *feedback control loop* is used for gathering metrics (such as CPU usage and energy consumption) exploited in the *compute* phase. Consequently, this phase involves communicating with the descendant nodes, retrieving the CPU usage percentage of each core, retrieving the energy consumption data using RAPL, interpolating the energy consumed by each CPU core, and mapping the energy consumption to the corresponding applications. On the other hand, the *compute* phase defines how the energy target should be distributed among the descendent components. Such distribution can be static (*i.e.*, enable defining for each component fixed energy consumption targets) or dynamic (*i.e.*, enable assigning, for each component, energy consumption targets that change over time, being adaptable to workload or system variations). The dynamic energy distribution is accomplished through two different algorithms, *Priority Algorithm* and *Fairness Algorithm*, described in detail Section 3.4. The *Priority Algorithm* assigns each node energy target considering previously defined priorities for each component. This means that components (racks, servers, CPU cores, applications) with higher priority have access to a larger energy share. Conversely, components with medium or low priority get smaller portions of energy. The *Fairness Algorithm* assigns each node energy target considering their proportional usage of CPU. Therefore, components with higher CPU usage percentages can access a larger energy share. The *enforcement* phase of the loop involves communicating with the descendent nodes to transmit the enforcement rules obtained in the *compute* phase. At the end of the *enforcement* phase, the system waits for acknowledgment (ACK) messages from the descendent nodes to confirm that the enforcement rules have been successfully received and implemented. The *sleep* phase defines the periodicity of these control cycles (*e.g.*, perform the aforementioned control tasks at 1-second intervals).

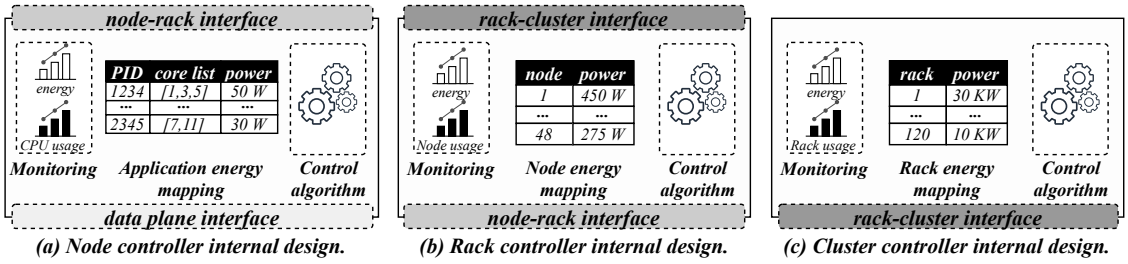


Figure 3.4: FINER detailed control architecture.

3.2.2.1 Node Controller

Each *node controller* can be seen as an independent component that (even without communicating with *rack controllers* throughout the hierarchy) can control the energy consumption of a server by applying energy control algorithms. This controller periodically *collects* metrics (such as the CPU usage, or the actual energy consumption of the server regarding the CPU cores, GPU, or a specific application), *computes* the energy control algorithm (verifies if all policies are being met by correlating the energy targets defined by the system administrator and the collected values), and *enforces* the generated rules (frequencies to apply) by submitting them to the *data plane*.

To perform the mentioned activities, the *node controller* internal design (Figure 3.4) comprises three essential subcomponents. The energy mapping subcomponent maps the collected metrics to the corresponding hardware components. It translates the raw data into actionable insights, determining which energy-saving measures to apply based on current conditions. The core of the *node controller*, the control algorithm, processes the mapped metrics and decides the optimal energy policies to enforce. It continuously adjusts the server's operational parameters to balance performance and energy efficiency. Finally, the monitoring subcomponent oversees the node's status in real-time, ensuring that all changes and adjustments made by the control algorithm are effective. It provides feedback to the energy mapping and control algorithm subcomponents, enabling a closed-loop control system.

Furthermore, as seen in Figure 3.4, the *node controller* comprises two interfaces: one to establish the communication with *rack controller* and another to establish the communication with data plane stages. The node-rack interface facilitates communication between the *node controller* and the *rack controller*, allowing the *node controller* to send aggregated metrics and receive higher-level energy policies from the *rack controller*. This interface ensures that the node's energy management aligns with the overall rack-level strategy. Table 3.2 outlines the interface exposed by the *node controller* to the *rack controller*. On the other hand, the data plane interface (Table 3.1) connects the *node controller* with the server's hardware components, such as CPU or GPU. It enables interaction with the data plane, ensuring the implementation of control actions.

Table 3.2: Node-Rack interface.

Interface	**Description**
`collect_usage()`	Collects resource usage statistics from the *node controller*.
`collect_energy()`	Collects energy consumption statistics from the *node controller*.
`enf_rule(node_id, energy)`	Details rules to be enforced on the *node controller*.

A `collect_usage()` call (Table 3.2) gathers the resource usage metrics (such as CPU usage) about the *node controller*. This information is essential for the computing phase of the *rack controller*. Furthermore, a `collect_energy()` call collects the *node controller*'s aggregated energy consumption. It monitors the energy over time and is used to adjust the *rack controller* control algorithms. Finally, a `enf_rule()` call is used to enforce a specific energy target (`energy`) in a particular *node controller* (identified by the `node_id`). This call allows the *node controller* to receive the enforcement rules dictated by the *rack controller* and adjust its energy target.

3.2.2.2 Rack Controller

Establishing a hierarchy of controllers, even by adding a single *rack controller*, enables the application of more complex energy control algorithms. Each *rack controller* is responsible for periodically *collecting* the aggregated energy consumption of each compute node in the lower

level of the hierarchy, *computing* the energy control algorithm (verify if all policies are being met by correlating the energy targets defined by the system administrator and the collected values), and *enforcing* the generated rules (energy targets) by submitting them to the *node controller*s in the lower level of the hierarchy. This interaction is illustrated in Figure 3.2.

To perform these activities, the *rack controller* internal design (Figure 3.4) comprises the same subcomponents as the *node controller*. The energy mapping subcomponent, with the same functionality as in the *node controller*, maps the aggregated collected metrics to the corresponding computing node. The control algorithm processes the mapped metrics and decides the optimal energy policies to enforce in each *node controller*. Finally, the monitoring subcomponent oversees the aggregated computing node's status in real-time, ensuring that all changes and adjustments made by the control algorithm are effective and providing feedback to the energy mapping and control algorithm subcomponents.

The *rack controller* also comprises two interfaces: one to establish the communication with *cluster controllers* and another to establish the communication with *node controllers*. The rack-rluster interface establishes communication between the *rack controller* and the *cluster controller*, allowing the *rack controller* to send aggregated metrics and receive higher-level energy policies from the *cluster controller*. This interface ensures that the racks' energy management aligns with the overall cluster-level strategy. Unlike the interface exposed by the *node controller* to the *rack controller* (shown in Table 3.2), in this case, metrics are collected from the *rack controller*, and rules are enforced in the *rack controller*. Apart from these differences, the interfaces are otherwise identical. On the other hand, the node-rack interface connects the *rack controller* with the *node controller*, ensuring the implementation of control actions throughout the hierarchy.

3.2.2.3 Cluster Controller

The responsibilities and functionalities of the *cluster controller* are very similar to the *rack controller*'s, with the difference being that, instead of *collecting* and *enforcing* in *node controllers*, the *cluster controller* interacts with *rack controllers* (Figure 3.2).

The *cluster controller* subcomponents are also very similar to the *rack controller*'s subcomponents. The energy mapping subcomponent maps the aggregated collected metrics to the corresponding rack. The control algorithm processes the mapped metrics and decides the optimal energy policies to enforce in each *rack controller*, and the monitoring subcomponent aggregates each rack's status in real-time to allow adjustments made by the control algorithm.

Unlike the node and *rack controllers*, the *cluster controller* comprises one interface to establish communication with *rack controllers*. The rack-cluster interface connects the *cluster controller* with the rack, ensuring the implementation of control actions throughout the hierarchy.

At this hierarchy level, the administrator can apply holistic energy control rules which will be routed throughout the hierarchy. Such rules can be static by defining fixed energy consumption targets for each descendent node or dynamic by employing energy control algorithms. The control algorithms assign energy consumption targets that change over time for each descendent node. Although the administrator can apply energy control rules at each level of the hierarchy (*node*

controller, *rack controller*, or *cluster controller*), it is the *cluster controller* that ensures the global optimization of energy consumption across the entire data center.

3.3 Implementation

The implementation of FINER consisted of developing the two components that make up the system architecture: the data plane and the control plane. Each component actuates independently, following the decoupled design principle. This section explains and describes the implementation of each component.

3.3.1 Data Plane

The data plane was implemented with approximately 520 lines of C++ code. To complement the logic described in the architecture (Section 3.2.1), this section particularly describes how the data plane applies the different frequencies and how it communicates with the control plane.

DVFS For the purposes of this work, the data plane stages enforced the energy limits by applying **CPU frequency scaling** (DVFS). Such a technique consisted of applying the *userspace* governor in all the CPU cores to ensure that specific frequencies could be enforced. Throughout the execution of the system, as the data plane stages receive the enforcement rules, they change the CPU frequency by overwriting the

`/sys/devices/system/cpu/cpuX/cpufreq/scaling_setspeed`

file (*i.e.*, replacing the *X* with the corresponding CPU core number) with the received frequency.

Communication Communication with the *node controller* is established through UNIX Domain Sockets. This approach allows for high-performance, low-latency communication within the same host.

3.3.2 Control Plane

As mentioned in Section 3.1, the FINER implementation draws inspiration from SDN/SDS paradigms. The proposed requirements and architecture align with some systems that apply the *software-defined* paradigm, which ensures quality of service, maintainability, resource fairness, performance, and scalability [66], and align with the design principles of this work's solution. This paradigm has been applied in different contexts and domains [50, 65, 70], which proves its efficacy and robustness. One can find a system that follows this paradigm by analyzing the work produced by *Macedo et al.* [65]. The authors implemented an application- and file system-agnostic storage middleware [1] that enables QoS control of data and metadata workflows in HPC storage

[1] https://github.com/dsrhaslab/cheferd

systems. Although this work applies the *software-defined* paradigm in the context of storage systems, the implementation was leveraged and adapted to control energy consumption.

The control plane's current implementation consists of approximately 5500 lines of C++ code and includes the *rack controller* and *node controller*. The *cluster controller* was not developed due to the exploratory nature of this work, the difficulty of validation, and the limited time available.

3.3.2.1 Node Controller

This section describes the *node controller* implementation by specifying how the monitor subcomponent was developed using a CPU usage monitoring module and an energy monitoring module and how the energy mapping occurred by translating energy to frequency and calibrating the frequency. The energy control algorithms were described in Section 3.4.

CPU usage monitoring module In the *node controller*, the *collect* phase includes obtaining the CPU usage percentage of each CPU core by accessing the file:

```
/proc/stat
```

and retrieving the information related to the number of CPU busy cycles and total cycles of each CPU core. To determine the number of CPU busy cycles for a core, the values from user (normal processes in user mode), nice (niced processes in user mode), and system (processes in kernel mode) are combined. The total cycles for a CPU core are obtained by adding the idle (twiddling thumbs) values to the busy cycles. Using the busy and total cycles of each CPU core, the core's CPU usage percentage is computed by dividing its busy cycles by the sum of the total cycles across all cores.

Energy monitoring module The energy monitor phase in FINER is done through an independent module. This is designed to measure and monitor energy consumption across various levels of granularity in different software and hardware components. It achieves this by leveraging Intel's Running Average Power Limit (RAPL) interface, which provides detailed energy consumption data. To enhance the accuracy of its measurements, FINER monitoring module employs interpolation techniques, allowing it to estimate the energy usage of components even when direct measurements are not possible. The *node controller collect* phase includes communicating with the energy monitor module to retrieve the energy consumption of each component.

Energy to frequency The *compute* phase of the *feedback control loop* generates the energy targets for each descendent node, while the *enforcement* phase imposes the energy targets (if the node is a *rack controller*) or the CPU frequencies to apply (if the node is a *node controller*). In the *node controller*, an additional step between these phases converts the energy into frequency. This conversion process begins with data gathered at the program's start, specifically using a table read from a file defined in the configuration settings. This table (which will be called the frequency-energy table for the rest of the document) contains the available frequencies for the machine and

the energy obtained for each CPU core running in all the frequencies. Such a table must be obtained through profiling done as explained in Section 4.2.2.

This table enables the program to access the available frequencies and the maximum and minimum energy consumption values corresponding to the number of active components in the machine. Using this information, after the *compute* phase, an interpolation is performed. This interpolation utilizes the lowest and highest available frequencies and the minimum and maximum energy consumption values for the specific number of active components as well as the target energy to determine the ideal frequency. Then, the ideal frequency is mapped to the closest available frequency. This conversion is done for all the components.

Calibrating frequency When some iterations of the *feedback control loop* pass, and the collected energy consumed is too low or too high compared to the target, the energy value to distribute is adjusted in the *node controller*. For the purpose of this work, the program waits until 5 iterations have elapsed. After 5 iterations, if the consumed energy is lower or higher than the target, the energy allocated for each component is adjusted by increasing or decreasing 1 W, respectively, to fine-tune energy mapping to frequency. This adjustment only happens until the energy allocated for each component does not surpass the maximum or falls below the minimum described in the frequency-energy table.

Communication Communication between node and *rack controllers* is facilitated through Remote Procedure Calls (RPC), utilizing the gRPC framework [4]. This method ensures efficient and reliable interactions across different controllers. On the other hand, communication between the *node controller* and the energy monitor module is established through UNIX Domain Sockets, similar to the communication with data plane stages.

Energy control rules and configurations The energy control rules defined by the administrator are specified in the configuration files that the FINER receives as input. Depending on the selected rules, 5 different configuration files can be used: the main configuration file, the frequency-energy table, the static rules file, the components to control file for the fairness algorithm, and the components to control file for the priority algorithm. The main configuration file is composed of the following parameters:

- **controller** — Defines the type of controller of the current node (can be - local - for the *node controller*, or - global - for the *rack controller*).
- **core_address** — Specifies the IP address of the *rack controller* (to establish the communication).
- **local_address** — Configures the IP address of the *node controller*.
- **control_type** — Defines the control type (energy control rules) to be applied (can be - 0 - for static rules, - 1 - for the fairness algorithm -, and - 2 - for the priority algorithm).

- **max_limit** — Sets the energy target for the current node.
- **components_to_control_file** — Specifies the path to another configuration file that describes the hardware components to monitor and control.
- **table_file** — Configures the path to another configuration file that constitutes the frequency-energy table.
- **rules_file** — Sets the path to another configuration file describing the static energy rules for each hardware component.

These parameters are not all mandatory and can be arranged depending on the control type. For instance, when trying to execute the *node controller* independently, there is no need to specify the *core_address*. When using the fairness algorithm, the user needs to define the *controller*, *local_address*, *control_type*, *max_limit*, *components_to_control_file*, and *table_file* parameters. The *components_to_control_file* for the fairness algorithm describes each hardware component (through its identifier) and its respective maximum energy limit. For instance, a line of such a file could be:

```
cpu 1 3 85
```

defining that the system should control the CPU core 3 of package 1 and that this component has a maximum physical power limit of 85 W. The *table_file* is obtained as described in Section 4.2.2, and each line represents a CPU frequency and the energy obtained by a set of active CPU cores running a CPU-intensive application. For instance, a line of such a file could be:

```
800000 14 0 0 0
```

describing that, in a computing node with 4 CPU cores, when applying the frequency of 800000 KHz to all the CPU cores and running a CPU-intensive application in core 0, it obtains a maximum energy of 14 W.

When using the priority algorithm, the user needs to define exactly the same parameters. However, the *components_to_control_file* should be different. In this case, besides describing each hardware component (through its identifier) and its maximum energy limit, the file needs to detail each component's priority. Each priority is defined with a number: 0 for low priority, 1 for medium priority, and 2 for high priority components. For the provided example, a line of such a file could be:

```
cpu 1 3 85 0
```

meaning that CPU core 3 of package 0 should have low priority.

Lastly, when setting static energy control rules, the administrator should include in the main configuration file the parameters: *controller*, *local_address*, *control_type*, *rules_file*, and *table_file*. All the parameters are the same as in the previous control types, except the *rules_file*. The *rules_file* specifies static energy control rules by describing, in each line, the rule's identifier,

the hardware component identifier, and the energy target for that component. The energy target is assumed to not surpass the hardware component's physical maximum limit. A line of such a file could be:

```
1 cpu 1 3 10
```

specifying that rule 1 defined an energy target of 10 W for the CPU core 3 of package 1.

3.3.2.2 Rack Controller

The *rack controller* constitutes an abstraction from the *node controller*, making its implementation more straightforward. By following the logic described in 3.2.2.2 and using the *node controller* implementation as a baseline, the *rack controller* implementation consisted of altering some functionalities. Thus, this section details specifically the monitoring and energy mapping subcomponents and how the administrator configures the system to apply different energy control rules. The energy control algorithms, which are the same as in the *node controller* are described in Section 3.4.

Monitoring and energy mapping In the *rack controller*, the *collect* phase includes retrieving the aggregated metrics of the descendent nodes. Accordingly, the *rack controller* communicates with *node controllers* to obtain the total CPU usage and the total energy consumption of each computing node.

Unlike the *node controller*, in the *rack controller enforcement* phase the messages to transmit embody the energy target for the descendent nodes (*node controllers*). Upon receiving the new energy target, the *node controller* updates its energy limit variable, and the control algorithms are executed with the latest information.

Energy control rules and configurations Given the *rack controller* abstraction, its configuration is also simpler. Depending on the selected rules, 3 different configuration files can be used: the main configuration file, the static rules file, and the components to control file for the priority algorithm. This controller does not need the frequency-energy table, given that the translation of energy to frequency is a *node controller* functionality, nor does the components to control file for the fairness algorithm, since it relies on the establishment of communication from the *node controllers* to know which nodes to control. The main configuration file is composed of a smaller set of the same parameters as in the *node controller*.

When using the fairness algorithm, the user needs to define the *controller*, *core_address*, *control_type*, and *max_limit* parameters. These parameters have the same functionalities as in the *node controller*, except the *core_adress* which configures the IP address of the *rack controller*.

If the administrator wants to apply the priority algorithm, the same parameters are used, adding the *components_to_control_file*. This file follows the same structure as in the *node controller*. A line of such a file could be:

```
server 38 195 2
```

meaning that server 38 has a maximum physical power limit of 195 W and has high priority.

When applying static energy rules, the administrator needs to set the *controller*, *core_address*, *control_type*, and *rules_file* parameters. The *rules_file* also follows the same structure as in the *node controller*. A line of this file could be:

```
2 server 38 86
```

defining that rule 2 sets the server 38 with the target energy of 86 W. Also, in this case, it is assumed that the energy target does not surpass the hardware component's physical maximum limit.

3.4 Control Algorithms

Existing energy control algorithms (Section 2.2.2) may encounter challenges in adapting to dynamic environments and optimizing energy consumption effectively in large-scale computing infrastructures. One significant concern involves the need for more comprehensive hierarchical control frameworks. Many current approaches primarily focus on optimizing power usage at the server level, which may not fully address the broader management requirements of the entire data center. This approach can potentially result in inefficiencies and performance challenges across different areas of the data center, reducing overall effectiveness and energy efficiency. Furthermore, existing policies sometimes struggle to prioritize tasks adequately and ensure equitable resource allocation. This limitation might lead to performance bottlenecks and unequal treatment among applications competing for resources. Moreover, current algorithms face challenges in adapting to dynamic environments where workloads and energy sources vary, such as those incorporating renewable energy sources. Their rigid nature can limit their ability to effectively optimize energy consumption and maintain stable operations. Two energy control algorithms were proposed to address these challenges: the *Fairness* and the *Priority* Algorithm.

3.4.1 Fairness Algorithm

The fairness algorithm 1 is designed to distribute energy fairly among various system components based on their resource usage. This algorithm aims to ensure that each component receives a fair share of the available energy while considering each component's maximum energy limit and current energy consumption. For the purpose of this work, the resource usage consists of the CPU usage percentage.

The algorithm utilizes predefined variables for its computations, namely the target energy (variable $Target_{energy}$) to be distributed through the hardware components (which can be CPU cores in the *node controller* or servers in the *rack controller*), the number of active hardware components (variable N_{comp}), an energy threshold (variable ε) and each component physical maximum power limitation (variable $physical\_lim_i$). While the target energy and each component's physical

Algorithm 1 Fairness Algorithm.

Initialize: $0 < Target_{energy} < \sum_{i=0}^{N_{comp}} physical\_lim_i$; $N_{comp} > 0$; $0 \leq \varepsilon \leq Target_{energy}$; $physical\_lim_i > 0$

1: $\{m\_energy_0, ..., m\_energy_{Ncomp-1}\} \leftarrow collect\_energy()$
2: $\{m\_usage_0, ..., m\_usage_{Ncomp-1}\} \leftarrow collect\_usage()$
3: $left \leftarrow Target_{energy}$
4: $fair\_share \leftarrow Target_{energy}/N_{comp}$
5: **for** $i = 0$ in $[0, N_{comp} - 1]$ **do**
6: $\quad to\_enforce_i \leftarrow m\_energy_i$
7: $\quad$ **if** $m\_energy_i < fair\_share$ **then**
8: $\quad\quad$ **if** $m\_energy_i + \varepsilon < fair\_share$ **then**
9: $\quad\quad\quad$ **if** $m\_energy_i + \varepsilon \leq physical\_lim_i$ **then**
10: $\quad\quad\quad\quad to\_enforce_i \leftarrow m\_energy_i + \varepsilon$
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ **if** $fair\_share \leq physical\_lim_i$ **then**
13: $\quad\quad\quad\quad to\_enforce_i \leftarrow fair\_share$
14: $\quad$ **else**
15: $\quad\quad to\_enforce_i \leftarrow fair\_share$
16: $\quad left \leftarrow left - to\_enforce_i$
17: $visited\_comp \leftarrow []$
18: **while** $left > 0$ & $length(visited\_comp) < N_{comp}$ **do**
19: $\quad copy\_left \leftarrow left$
20: $\quad$ **for** $i = 0$ in $[0, N_{comp} - 1]$ **do**
21: $\quad\quad$ **if not** $visited\_comp.contains(i)$ **then**
22: $\quad\quad\quad to\_enforce\_left \leftarrow m\_usage_i \times copy\_left$
23: $\quad\quad\quad$ **if** $to\_enforce_i + to\_enforce\_left \leq physical\_lim_i$ **then**
24: $\quad\quad\quad\quad to\_enforce_i \leftarrow to\_enforce_i + to\_enforce\_left$
25: $\quad\quad\quad\quad left \leftarrow left - to\_enforce\_left$
26: $\quad\quad\quad\quad$ **if** $to\_enforce\_left = 0$ **then**
27: $\quad\quad\quad\quad\quad visited\_comp.add(i)$
28: $\quad\quad\quad$ **else**
29: $\quad\quad\quad\quad left \leftarrow left - physical\_lim_i - to\_enforce_i$
30: $\quad\quad\quad\quad to\_enforce_i \leftarrow physical\_lim_i$
31: $\quad\quad\quad\quad visited\_comp.add(i)$
32: $enforce(\{to\_enforce_0, ..., to\_enforce_{Ncomp-1}\})$
33: $sleep(loop\_interval)$

power limitation are obtained through the configurations of the system, the number of active components is calculated based on the collected metrics, and the energy threshold is defined as 1 for the purpose of this work.

After *collecting* the metrics (energy consumption values through variables m_energy_i, and resource usage proportion through variables m_usage_i), the algorithm starts by calculating a fair share of energy for each hardware component (variable $fair\_share$). This is done by dividing the total energy to be distributed by the number of active components. Also, the remaining energy

(variable $left$) is initialized. After this, the algorithm is divided into two phases. Phase one (lines 5 to 16) distributes energy equitably among system components, and phase two (lines 17 to 31) distributes the leftover energy.

As phase one iterates through each component (line 5), it initializes the energy allocation of the current hardware component (variable $to\_enforce_i$) with the current energy consumption of the component (variable m_energy_i). Then, it evaluates whether the current energy consumption is less than the calculated fair share (line 7). The algorithm allocates the corresponding fair share if the current energy consumption exceeds the calculated fair share (line 15). However, if the current energy consumption of the hardware component is less than the calculated fair share, the energy allocation is adjusted considering the energy threshold, the fair share, and the component's physical limitations (lines 8 to 13).

If adding a threshold to the current consumption surpasses the component's fair share, the energy target to enforce in the hardware component is its fair share (line 13), as long as it does not surpass the component's physical maximum limitation. Conversely, if adding a threshold to the current consumption remains below the component's fair share (line 8), that is the energy target to enforce as long as it does not surpass the component's physical maximum limitation (line 10). Throughout this phase, the algorithm updates the variable to track the remaining energy (*left*, line 16).

The second phase of the algorithm (lines 17 to 31) begins by initializing a variable (*visited_comp*) for storing the hardware components that should not increase their energy allocation (because they are in an *idle* state or at their physical power maximum limit). Thus, this variable stores the "visited components". Then, the algorithm continues distributing energy as long as there is leftover energy and not all components have been visited (line 18). In each iteration, for each unvisited component, the algorithm calculates the additional energy portion to allocate (variable $to\_enforce\_left$) based on the component's resource usage proportion (variable m_usage_i) and the leftover energy (variable $copy\_left$, line 22). If adding the additional energy portion (variable $to\_enforce\_left$) to the already allocated energy target (variable $to\_enforce_i$) does not surpass the hardware component's physical power maximum limit, the energy target of the component is updated with that total energy, and the leftover energy is updated by subtracting the additional portion of energy that was allocated (lines 24 and 25).

However, if adding the additional energy portion surpasses the component's physical maximum limit, the energy target to enforce is updated with the physical maximum energy limit (line 30), and the leftover energy is adjusted by subtracting the difference between the physical maximum energy limit and the energy target allocated in the first phase (this difference corresponds to the additional portion that is being added to the $to\_enforce_i$ variable, line 29). Additionally, in this case, the component is added to the visited components variable. In the case where the additional energy portion (variable $to\_enforce\_left$) gets the zero value (due to resource unutilization, where the hardware component is, for instance, in an *idle* state), the component is also added to the visited components variable (line 27). After the two phases, the generated rules are *enforced* in the descendent nodes (line 32), and the algorithm finishes by *sleeping* for *loop_interval* before

beginning a new control cycle (line 33).

The fairness algorithm ensures that energy is distributed across system components in a fair and balanced manner. By initially allocating a fair share and then proportionally distributing any leftover energy based on CPU usage, the algorithm ensures efficient and fair energy management while respecting each component's maximum limits.

3.4.2 Priority Algorithm

The priority algorithm 2 distributes energy among various system components based on their priority levels and energy consumption. This algorithm ensures that higher priority components receive a proportionally larger share of the available energy while considering each component's maximum energy limit and current energy consumption. For the purpose of this work, three priority levels were defined: high, medium, and low priority. However, the priority levels can be modified and even expanded. Currently, the priorities do not change over time, and each hardware component is given a defined priority in the configuration files, as explained in Section 3.3.2.1.

The priority algorithm relies on the auxiliary function *get_priorities_percentage* for determining the energy proportion allocated to each priority level. This function takes as parameters the numbers of hardware components in each priority level (variables $N_{high\_priority}$, $N_{mid\_priority}$, and $N_{low\_priority}$) and returns a vector that stores the calculated energy proportions for each priority level. The *get_priorities_percentage* function utilizes the Analytic Hierarchy Process (AHP) [80] to define these proportions based on the relative importance of high, medium, and low priority components. The Analytical Hierarchy Process (AHP) is a structured technique used for organizing and analyzing complex decisions based on mathematics and psychology. AHP helps decision-makers break down a problem into a hierarchy of more easily comprehensible sub-problems, each of which can be analyzed independently. The process involves defining the problem and structuring the hierarchy from the top (the goal of the decision) through intermediate levels (criteria on which alternatives are evaluated) to the bottom level (the list of alternatives). Decision makers then perform pairwise comparisons of the elements at each level, rating their relative importance on a numerical scale (from 1 to 9). This scale follows the standard codes:

- 1 - Equal importance.
- 3 - Moderate importance.
- 5 - Strong importance.
- 7 - Very strong importance.
- 9 - Extreme importance.
- 2, 4, 6, 8 - Intermediate values.
- 1/3, 1/5, 1/7, 1/9 - Inverse comparison.

These comparisons are used to generate a set of priorities for each element. Finally, the numerical priorities derived for each level are synthesized to determine which decision option has the highest overall priority and should be selected. AHP is widely used in various fields such as business, healthcare, and government for decision-making processes where multiple criteria need to be considered.

In the context of this work, an adaptation of the AHP algorithm was implemented. Instead of following through the entire algorithm, the AHP was applied to reach the criteria relative weights, which, in this context, correspond to the relative importance of each priority level. The algorithm was implemented by attributing the following relative importance for each priority level (taken as the criteria):

- Value 9 to high priority components compared to low priority ones. High priority is extremely important compared to low priority.
- Value 5 to medium priority components compared to low priority ones. Medium priority is strongly important compared to low priority.
- Value 4 to high priority components, compared to medium priority ones. High priority is moderate to strongly important compared to medium priority.

These values were attributed in a way that the consistency ratio obtained was 0.062296, smaller than the standard value of 0.10. Meaning that the metrics are reasonably consistent. These relative importances configured the following pairwise comparison matrix: $\begin{bmatrix} 1 & \frac{1}{5} & \frac{1}{9} \\ 5 & 1 & \frac{1}{4} \\ 9 & 4 & 1 \end{bmatrix}$ which after normalization and applying the criteria weights calculations resulted in the following priority level percentages:

- High Priority Percentage = 0.701308;
- Medium Priority Percentage = 0.236438;
- Low Priority Percentage = 0.062254;

Given that the number of hardware components in each priority level can differ from one, it would not be correct to simply use these percentages to calculate the energy portion for each component (as the sum of the proportions would result in a value higher than 100%). To ensure the correctness of the function, the proportions are normalized considering the number of components in each priority level. Finally, the adjusted percentages are assigned to the returned vector. The order of insertion is low, medium, and high, respectively. It is important to note that these values can be modified taking into account the system administrator preferences.

Algorithm 2 Priority Algorithm.

Initialize: $0 < Target_{energy} < \sum_{i=0}^{N_{comp}} physical\_lim_i$; $N_{comp} > 0$; $0 \leq \varepsilon \leq Target_{energy}$; $physical\_lim_i > 0$; $priority_i \in \{high, mid, low\}$; $0 \leq N_{high\_priority} \leq N_{comp} - N_{mid\_priority} - N_{low\_priority}$; $0 \leq N_{mid\_priority} \leq N_{comp} - N_{high\_priority} - N_{low\_priority}$; $0 \leq N_{low\_priority} \leq N_{comp} - N_{high\_priority} - N_{mid\_priority}$

1: $\{m\_energy_0, ..., m\_energy_{Ncomp-1}\} \leftarrow collect\_energy()$
2: $left \leftarrow Target_{energy}$
3: $priorities\_percentage[] \leftarrow get\_priorities\_percentage(N_{high\_priority}, N_{mid\_priority}, N_{low\_priority})$
4: $hp\_energy\_budget \leftarrow priorities\_percentage[2] \times Target_{energy}$
5: $mp\_energy\_budget \leftarrow priorities\_percentage[1] \times Target_{energy}$
6: $lp\_energy\_budget \leftarrow priorities\_percentage[0] \times Target_{energy}$
7: $visited\_comp \leftarrow []$
8: **for** $i = 0$ in $[0, N_{comp} - 1]$ **do**
9: $comp\_budget \leftarrow 0$
10: **if** $m\_energy_i = 0$ **then**
11: $comp\_budget \leftarrow \varepsilon$
12: $visited\_comp.add(i)$
13: **else if** $priority_i = high$ **then**
14: $comp\_budget \leftarrow hp\_energy\_budget / N_{high\_priority}$
15: **else if** $priority_i = mid$ **then**
16: $comp\_budget \leftarrow mp\_energy\_budget / N_{mid\_priority}$
17: **else**
18: $comp\_budget \leftarrow lp\_energy\_budget / N_{low\_priority}$
19: **if** $comp\_budget \leq physical\_lim_i$ **then**
20: $to\_enforce_i \leftarrow comp\_budget$
21: **else**
22: $to\_enforce_i \leftarrow physical\_lim_i$
23: $left \leftarrow left - to\_enforce_i$
24: **while** $left > 0$ & $length(visited\_comp) < N_{comp}$ **do**
25: $consumption\_proportion \leftarrow 0$
26: $copy\_left \leftarrow left$
27: $N_{high\_priority}, N_{mid\_priority}, N_{low\_priority} \leftarrow 0$
28: $total\_energy\_hp, total\_energy\_mp, total\_energy\_lp \leftarrow 0$
29: **for** $i = 0$ in $[0, N_{comp} - 1]$ **do**
30: **if not** $visited\_comp.contains(i)$ **then**
31: **if** $priority_i = high$ **then**
32: $total\_energy\_hp \leftarrow total\_energy\_hp + m\_energy_i$
33: $N_{high\_priority} \leftarrow N_{high\_priority} + 1$
34: **else if** $priority_i = mid$ **then**
35: $total\_energy\_mp \leftarrow total\_energy\_mp + m\_energy_i$
36: $N_{mid\_priority} \leftarrow N_{mid\_priority} + 1$
37: **else**

```
38:                 total_energy_lp ← total_energy_lp + m_energy_i
39:                 N_low_priority ← N_low_priority + 1
40:     priorities_percentage[] ← get_priorities_percentage(N_high_priority, N_mid_priority, N_low_priority)
41:     hp_share ← priorities_percentage[2] × copy_left
42:     mp_share ← priorities_percentage[1] × copy_left
43:     lp_share ← priorities_percentage[0] × copy_left
44:     for i = 0 in [0, N_comp − 1] do
45:         if not visited_comp.contains(i) then
46:             if priority_i = high then
47:                 consumption_proportion ← m_energy_i / total_energy_hp
48:                 to_enforce_left ← consumption_proportion × hp_share
49:             else if priority_i = mid then
50:                 consumption_proportion ← m_energy_i / total_energy_mp
51:                 to_enforce_left ← consumption_proportion × mp_share
52:             else
53:                 consumption_proportion ← m_energy_i / total_energy_lp
54:                 to_enforce_left ← consumption_proportion × lp_share
55:             if to_enforce_i + to_enforce_left ≤ physical_lim_i then
56:                 to_enforce_i ← to_enforce_i + to_enforce_left
57:                 left ← left − to_enforce_left
58:                 if to_enforce_left = 0 then
59:                     visited_comp.add(i)
60:             else
61:                 left ← left − physical_lim_i − to_enforce_i
62:                 to_enforce_i ← physical_lim_i
63:                 visited_comp.add(i)
64: enforce({to_enforce_0, ..., to_enforce_Ncomp−1})
65: sleep(loop_interval)
```

The priority algorithm utilizes predefined variables for its computations, namely the target energy (variable $Target_{energy}$) to be distributed through the hardware components (which can be CPU cores in the *node controller* or servers in the *rack controller*), the number of active hardware components (variable N_{comp}), an energy threshold (variable ε), and each component's physical maximum power limitation (variable $physical\_lim_i$). Additionally, it uses priority levels for each component (variable $priority_i$) and the number of components in each priority level (variables $N_{high\_priority}$, $N_{mid\_priority}$, and $N_{low\_priority}$). The target energy, each component's physical power limitation, each component's priority, and the number of components in each priority level are obtained through the configurations of the system. The number of active components is calculated based on the collected metrics, and the energy threshold is defined as 1 for the purpose of this

work.

After *collecting* the metrics (energy consumption values through variables m_energy_i), the algorithm starts by obtaining each priority level energy proportion by retrieving the vector *priorities_percentages* from the *get_priorities_percentage* function (line 3). After that, it is calculated the energy budget for each priority level based on their proportions (variables hp_energy_budget, mp_energy_budget, and lp_energy_budget, lines 4 to 6). This is done by multiplying the target energy by the proportion of each priority level. Additionally, the remaining energy (variable $left$) and the visited components (variable *visited_comp*) are initialized. The algorithm is divided into two phases. Phase one (lines 8 to 23) distributes energy based on priority among system components, and phase two (lines 24 to 63) distributes the leftover energy based on each component's priority and energy consumption.

As phase one iterates through each component, it initializes the energy budget of the current hardware component (variable $comp\_budget$, line 9) based on its priority level. High priority components get a budget obtained by dividing the total high priority budget (variable hp_energy_budget) by the number of high priority components (variable $N_{high_priority}$, line 14). The medium and low priority hardware components' budgets are calculated the same way, with the respective variables (lines 16 and 18, respectively). If the component's current energy consumption is zero, it is added to the list of visited components (variable $visited_comp$), and its budget is updated with the energy threshold (lines 11 and 12). The algorithm then allocates energy (variable $to\_enforce_i$) based on the component's energy budget and physical limitations. If the component's energy budget is within its physical limit, it is allocated that budget (line 20). Otherwise, it is allocated up to the physical limit (line 22), and the remaining energy (variable $left$) is adjusted accordingly.

The second phase of the algorithm (lines 24 to 63) continues distributing energy as long as there is leftover energy and not all components have been visited. This phase is also divided into two subphases. Subphase one (lines 25 to 43) is used to update the total energy consumed by each priority level (variables $total\_energy\_hp$, $total\_energy\_mp$, and $total\_energy\_lp$, lines 32, 35, and 38), as well as the number of components in each priority level (variables $N_{high\_priority}$, $N_{mid\_priority}$, and $N_{low\_priority}$, lines 33, 36, and 39) to which portions of energy can still be allocated (hardware components which are not in an *idle* state, nor at the physical maximum power limit), and each level's energy budget (variables *hp_share*, *mp_share*, and lp_share) using again the auxiliary function *get_priorities_percentage* (lines 40 to 43).

The second subphase (lines 44 to 63) distributes the leftover energy of each priority level according to current energy consumption. In each iteration, for each unvisited component, the algorithm calculates its consumption proportion (variable $consumption\_proportion$) based on its priority level and current energy consumption (lines 47, 50, and 53). This proportion is obtained by dividing the energy consumed by the component by the total energy consumed by the components in the same priority level. The additional energy portion to allocate (variable $to\_enforce\_left$) is then determined by multiplying this consumption proportion by the remaining energy budget for the corresponding priority level (lines 48, 51, and 54). If adding the additional energy portion (variable $to\_enforce\_left$) to the already allocated energy target (variable $to\_enforce_i$) does

not surpass the hardware component's physical power maximum limit, the energy target of the component is updated with that total energy, and the leftover energy is updated by subtracting the additional portion of energy that was allocated (lines 56 and 57). However, if adding the additional energy portion surpasses the component's physical maximum limit, the energy target to enforce is updated with the physical maximum energy limit, and the leftover energy is adjusted by subtracting the difference between the physical maximum energy limit and the energy target allocated in the first phase (lines 61 and 62). Additionally, in this case, the component is added to the visited components variable. If the additional energy portion (variable $to\_enforce\_left$) gets the zero value, the component is also added to the visited components variable (line 59). After the two phases, the generated rules are *enforced* in the descendant nodes, and the algorithm finishes by *sleeping* for $loop\_interval$ before beginning a new control cycle (lines 64 and 65).

The priority algorithm ensures that energy is distributed across system components in a prioritized and balanced manner. The algorithm ensures efficient and fair energy management while respecting each component's maximum limits by initially allocating energy budgets according to priority and then proportionally distributing any leftover energy based on energy consumption.

3.5 Summary

The FINER proposed solution for energy management in data centers aims to advance state of the art by introducing a hierarchical, fine-grained, and adaptive control system. This system provides efficient and scalable energy management through real-time monitoring, dynamic adjustment, and granular control across different infrastructure levels. The solution introduces key principles such as a decoupled design, fine-grained energy control, coordinated control, and adaptable energy tunning and provides a structured approach to monitor, compute, and enforce different energy policies. The architecture comprises a three-level hierarchical control plane, consisting of node controllers, rack controllers, and a cluster controller. This structure ensures a global view and control over the entire data center while enabling localized, fine-grained management. Node controllers manage individual compute nodes, rack controllers oversee nodes within a rack, and the cluster controller coordinates the overall energy strategy across the data center.

The proposed control algorithms are designed to dynamically adjust energy usage based on real-time metrics. These algorithms operate in three phases: collect (gathering metrics from the descendent nodes of the hierarchy), compute (analyzing metrics to determine optimal energy allocation), and enforce (implementing strategies to balance performance and energy efficiency). The fairness algorithm was designed to distribute energy fairly among various system components based on their resource usage, while the priority algorithm distributes the energy among the descendent nodes based on their priority levels and energy consumption.

FINER is particularly useful in large-scale data center infrastructures where energy efficiency is paramount. It is ideal for environments with high computational demands, such as high-performance computing clusters, and supercomputers. The current focus of the solution is on CPU energy management. This decision is crucial because many clusters exclusively use CPUs. This

trend is also relevant given the emergence of supercomputers with ARM chips, such as RIKEN's Fugaku [3] and TACC's VISTA (Grace/GraceHopper) [9]. The solution aligns with the ongoing shift towards ARM-based supercomputing by targeting CPUs, ensuring its applicability to future high-performance computing systems.

In summary, the proposed solution offers a fine-grained, holistic, hierarchical energy controller for managing the energy consumption in data centers, advancing the state of the art with real-time adaptive control and granular management. It is particularly suited for large-scale, CPU-centric infrastructures, addressing the growing trend of ARM-based supercomputing.

Chapter 4

Experimental Evaluation

The evaluation of the proposed solution sets out to answer the following questions:

1. Can FINER employ different energy control algorithms over the infrastructure?
2. Can FINER dynamically adapt to system changes?
3. Can FINER control the infrastructure's energy consumption at different granularity levels?
4. What is the performance impact in FINER when the number of components to manage increases?

4.1 Experimental Setup

The experiments were performed with the following experimental setup.

Hardware and OS configurations The experiments were conducted under three different hardware configurations. ***Compute Node 1 (CN_1)*** respects to a server with two 24-core Intel Xeon Gold 6342 2.8GHz processors, 192 GiB of memory, a 480 GiB SATA SSD, and connected with a 10 Gbps network card. ***Compute Node 2 (CN_2)*** respects to a commodity server, configured with an Intel Core i3 4170 3.7GHz CPU, 16 GiB of memory, a 120GB SATA SSD, and a 1 Gbps network card. ***Compute Node 3 (CN_3)*** respects to a server with two 18-core Intel Xeon Gold 6240 2.6 GHz CPU, 192 GiB of memory, a 480 GiB SATA SSD, connected with a 10 Gbps network card. Software-wise, all servers were configured with Ubuntu Server 20.04 LTS with Linux kernel 5.4.0 and an ext4 file system.

Applications and workloads To demonstrate the applicability and effectiveness of FINER in a realistic testing environment, three different applications that provide representative workloads of those executed in large-scale infrastructures were selected.

The first application is **PyTorch** [60], a popular machine learning framework commonly used for training deep learning and large language models. The SqueezeNet neural network [49, 8]

with the CIFAR-100 dataset [59] was used for all testing scenarios. PyTorch was configured with 4 workers with dataset shuffling enabled. Each training iteration handled a batch size of 1024 samples. PyTorch was selected because it exhibits a compute-intensive workload, with periodic phases for model validation and checkpointing. Although training is primarily conducted over GPU devices in a production environment, this configuration is used to demonstrate the feasibility of using FINER. Support for GPU frequency tuning is planned for future work.

The second application is **RocksDB** [6], a popular log-structured merge-tree key-value store from Meta, used as a storage engine for databases, machine learning engines, and file systems. For all testing scenarios, the YCSB benchmark was executed over RocksDB with a read-write mixture workload (namely, workload A, with 50% reads and 50% writes). Experiments were conducted with 100M key-value pairs in a uniform distribution over a preloaded dataset of 50M key-value pairs, with 16B-sized keys and 100B-sized values. RocksDB was set with a single client generating the main load and configured with 4 background threads for handling internal operations of the key-value store, specifically 3 compaction threads and 1 flushing thread. RocksDB was selected because it exhibits an I/O-intensive workload, with periodic phases of medium compute load due to compactions being triggered.

The third application is the **High-Performance Linpack** (HPL) benchmark [15], a software package used for solving random dense linear systems with double-precision arithmetic on distributed-memory computers. HPL is widely used to assess compute node performance in HPC supercomputers. For all experiments, the number of problem sizes (N) was set to 8, with the following sizes: 900, 8000, 10000, 8000, 900, 3000, 6000, and 10000. These sizes represent the dimensions of the processed matrices, which impact the computational load and memory usage. HPL was selected because it exhibits a compute-bound workload, making it suitable for evaluating the computational performance and efficiency of the system.

All the applications evaluated in this study were running locally. Although analyzing the performance and impact of energy control on distributed applications would be interesting, this is left for future work.

Methodology For most tests, all applications were run simultaneously on the same server to ensure that FINER's energy management capabilities were evaluated under high-load conditions. The node controller, responsible for managing the energy consumption of individual compute nodes, was executed directly on the server hosting the applications, specifically in $\boldsymbol{CN}_1$.

Core affinity was defined for each application to ensure proper control of the applications and prevent their processes from being scheduled across different CPU cores. This setup simulated cloud environments where containers or virtual machines are used, as well as HPC environments that employ operating system primitives like *cgroups* for isolation. Additionally, applications were run on the same NUMA (Non-Uniform Memory Access) node, while the FINER processes, primarily the node controller, were run on a different NUMA node. This separation aimed to minimize the energy impact associated with managing and monitoring tasks, ensuring that the

evaluation focused on the energy consumption of the applications themselves rather than the overhead introduced by FINER.

4.2 Profiling

A workload characterization of each application was conducted to gain a deeper understanding of application functionality and draw conclusions about their energy consumption, analyzing the CPU, memory, and disk utilization.

Furthermore, comprehensive profiling over $\boldsymbol{CN}_1$ CPU cores was conducted to understand (i) the energy consumption range (*i.e.*, minimum and maximum) of the server and (ii) the impact on the energy consumption of CPU cores when set at different operation frequencies. The profiling utilized a CPU-intensive application, which continuously calculates prime numbers. This profiling was crucial for understanding how different operating frequencies impact the energy consumption of the CPU cores, which was used for implementing an effective energy-to-frequency mapping strategy.

4.2.1 Workload characterization

To understand how the three applications behave in terms of resource occupation, a set of experiments was conducted to retrieve the CPU, memory, and disk usage. All reported resource utilization metrics were collected using the *dstat*[1] monitoring tool.

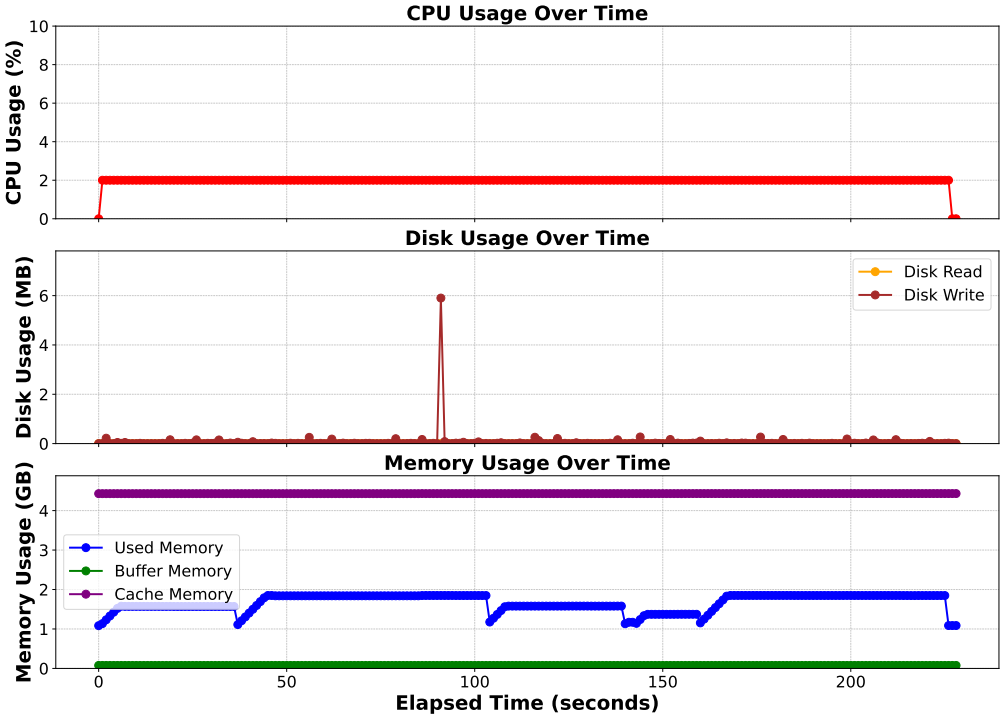


Figure 4.1: *HPL* profiling with CPU, memory and disk usage.

[1] https://linux.die.net/man/1/dstat

HPL The *HPL* profiling is depicted in Figure 4.1. While *HPL* is considered a CPU-intensive application, it presents a total CPU usage of 2% due to using only one CPU core (out of 48). Nevertheless, upon further inspection, it was found that the utilization of the CPU core responsible for executing *HPL* was actually at 100% at all times. *HPL* presents minimal disk activity, with occasional spikes in write activity up to 0.2 MB/s and a bigger spike of 6 MB/s at 95 s (as *HPL* transitions from one matrix size to another, it involves temporary storage or checkpointing by writing intermediate results to disk, particularly in this case where the problem sizes vary significantly, with the largest matrix being more than 10 times larger than the smallest one). The buffer and cache memories are both stable at 0.07 GB and 4.4 GB, respectively, showing consistent caching behavior. Nevertheless, the used memory varies between 1.1 GB and 1.8 GB, with periods of constant values.

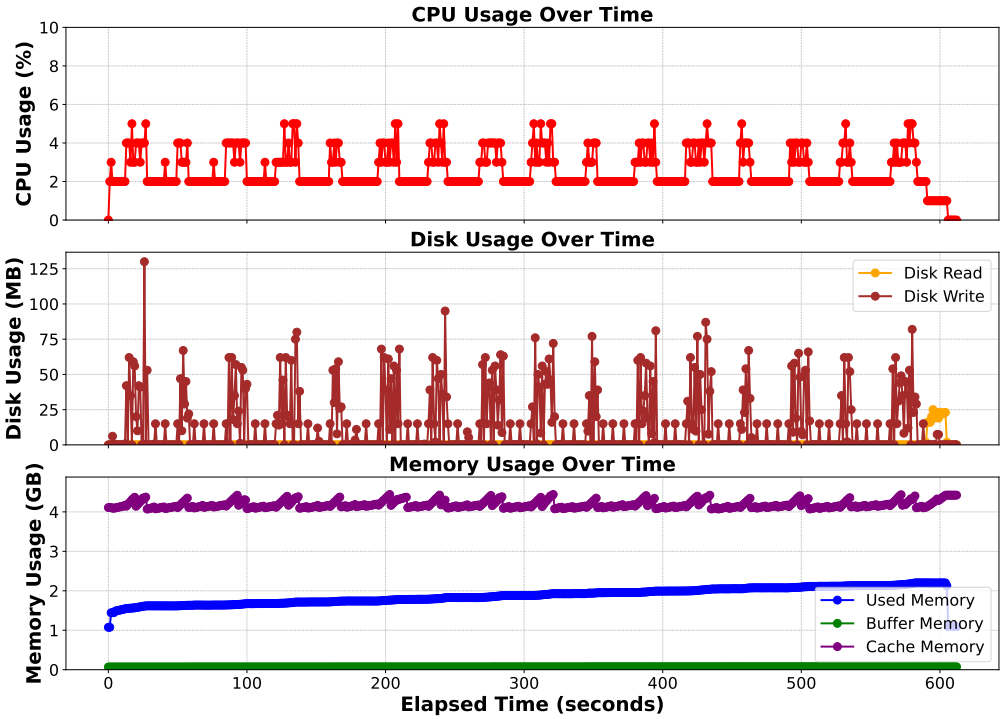


Figure 4.2: *RocksDB* profiling with CPU, memory and disk usage.

RocksDB Contrarely to *HPL* (which shows a constant usage of CPU), *RocksDB*'s CPU usage oscilates between 2 and 5% (Figure 4.2). The periodic peaks are caused by the background threads of *RocksDB*, specifically flushes and compactions. Such peaks can also be observed in disk utilization (writes). This application uses much more disk, with write operations going from 12 KB/s up to 130 MB/s. While the write activity managed by *RocksDB* initially interacts with the operating system's page cache before potentially persisting data to long-term storage, the read operations benefit from direct memory access hence, little read activity at disk level is observed. The buffer memory is stable at around 0.07 GB, while the cache oscillates between 4.07 and 4.4 MB due to compactions being triggered by *RocksDB* (*i.e.*, compactions read several files, merge them in

memory, and flush them to persistent storage), and the used memory continuously increases from 1 to 2 GB.

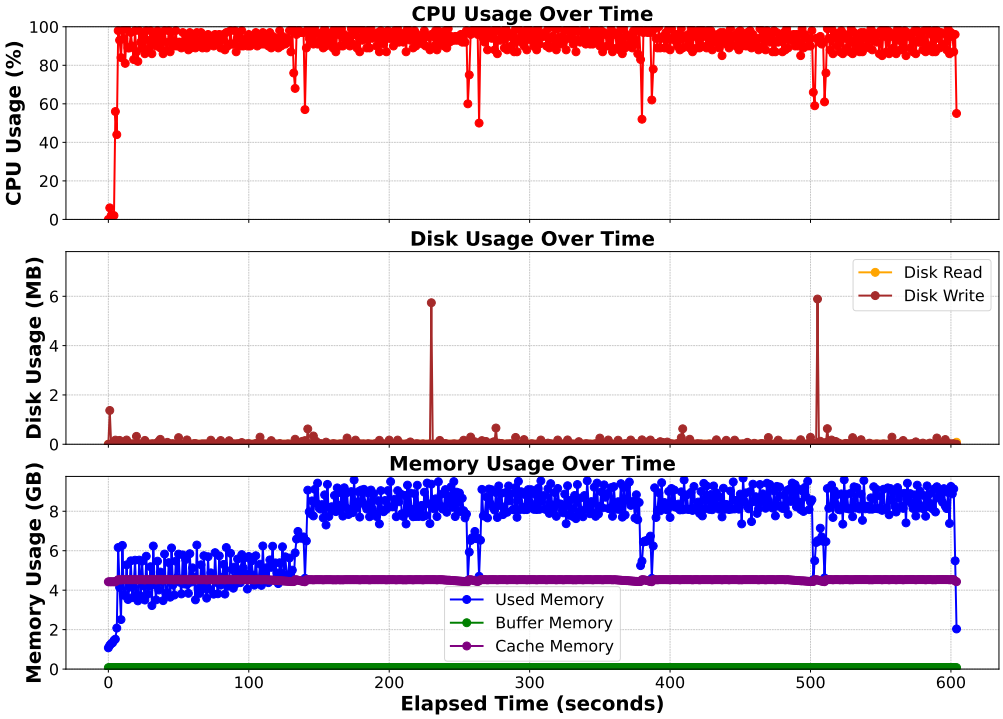


Figure 4.3: *PyTorch* profiling with CPU, memory and disk usage.

PyTorch *PyTorch* is the application that uses more CPU, with around 100% CPU utilization (Figure 4.3). It presents periodic phases that mark the end of training epochs, during which model validation is conducted. The disk activity is minimal (similar to *HPL*), with two spikes of the write operation around 6 MB/s. Regarding memory, *PyTorch* used far more memory than the other applications, with initial values around 5 GB and transitioning to values around 9 GB thereafter, with occasional troughs of around 5 GB. The buffer and cache memories remained stable at around 0.07 and 4.5 GB, respectively.

To summarize, the profiling of *HPL*, *RocksDB*, and *PyTorch* enabled several conclusions to be drawn. The three applications are very different in terms of behavior and resource utilization. *PyTorch* is by far the application that uses more CPU and memory, *RocksDB* is the application that uses more disk, with some read operations but mostly write activity, and *HPL* has the lowest resource consumption and is the most stable application.

4.2.2 Energy – core frequency analysis

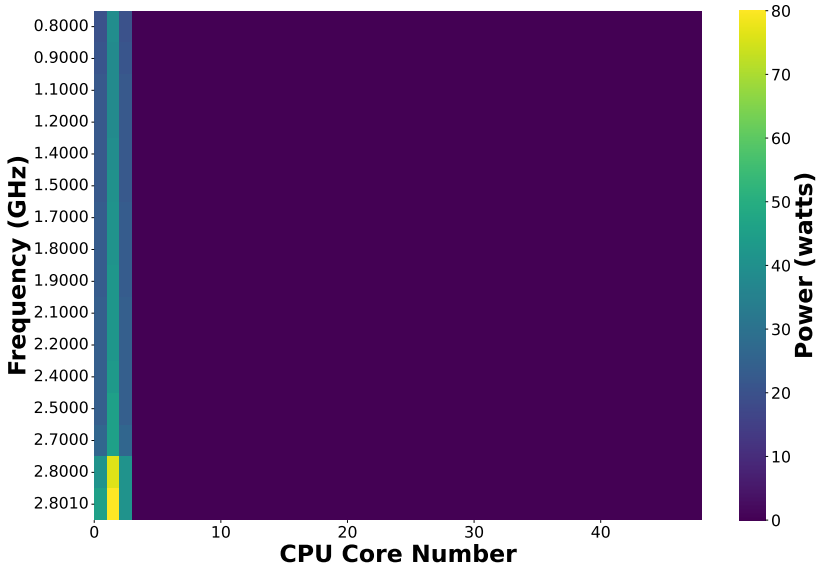
A comprehensive profiling of the $\boldsymbol{CN}_1$ server was conducted to further understand the impact of energy consumption across cores operating at different frequencies.

Using a specially designed CPU-intensive application that continuously calculates prime numbers, the evaluation consisted of running the application for 1 minute for each set of CPU cores (between 1 and 48) for each of the available frequencies, namely 0.8 GHz, 0.9 GHz, 1.1 GHz, 1.2 GHz, 1.4 GHz, 1.5 GHz, 1.7 GHz, 1.8GHz, 1.9 GHz, 2.1 GHz, 2.2 GHz, 2.4 GHz, 2.5 GHz, 2.7 GHz, 2.8 GHz, and 2.801 GHz. For instance, for the lowest frequency (0.8 GHz), the primes application was executed with affinity in CPU core 0 for 1 minute, then executed with affinity in CPU cores 0 and 1, then in CPU cores 0, 1, and 2, and so on, until the 48 CPU cores were used. Thus, given that a NUMA node of the machine has the CPU cores identified with odd numbers and the other has the cores identified with even numbers (between 0 and 47), this experiment evaluated the usage of 1 core of one NUMA node, then 1 core of the other NUMA node, then 2 cores of the same NUMA node and 1 core of the other NUMA node, then 2 cores of each NUMA node, so on and so forth.

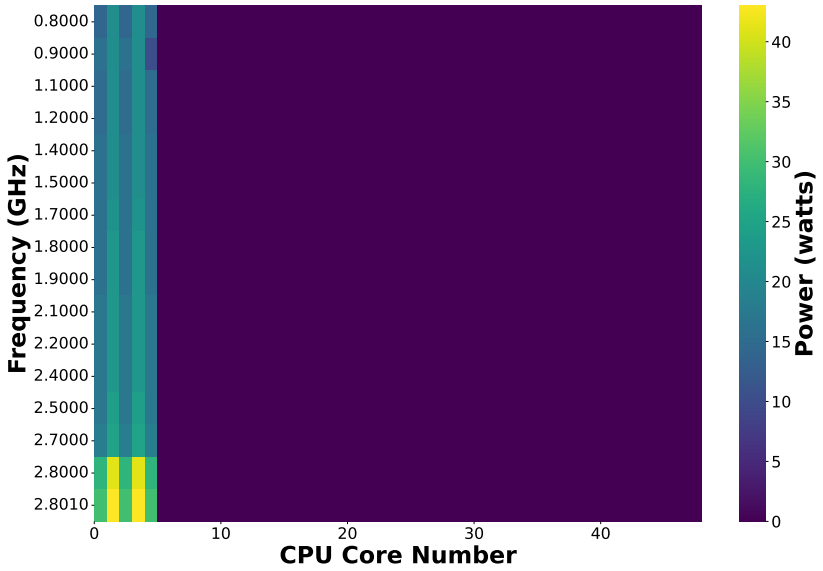
Given that the application was CPU-intensive, the CPU usage of the corresponding cores was at 100%. Thus, this evaluation showed the maximum energy consumption value for each frequency and each set of cores. Apart from Figures 4.4a and 4.4b, the results of these experiments can be found in Appendix A. Due to space constraints, the results were separated into matrices, each matrix representing a set of CPU cores to draw conclusions about the effects of applying different frequencies to the same sets of CPU cores, presented as heatmaps. Throughout this experiment, it was found that applying the same frequency to the same number of CPU cores in both NUMA nodes resulted in the same energy consumption values, thus, the figures presented in appendix A correspond to increasing one CPU core in the same NUMA node. The missing results (corresponding to even numbers of CPU cores) would show the same energy values for the other NUMA node. For instance, Figure 4.4b shows the results of using 3 CPU cores in one NUMA node and 2 in the other, resulting in a total of 5 CPU cores running. However, showing a figure with 6 active CPU cores would show the same conclusions as the values (and colors) of the 3 CPU cores running in the same NUMA node.

As depicted in the plots, there is a big difference in the energy consumption values of a CPU core when the number of active cores in the same NUMA node increases. Figure 4.4a, for instance, depicts this very well, as for the highest frequency (2.801 GHz), the energy values of CPU cores 0 and 3 (in the same NUMA node) go up to 45 and 41 W, while the energy value of core 2 (in the other NUMA node) goes up to 81 W. This insight is confirmed in different frequencies and sets of cores, which means that the number of active CPU cores affects the maximum energy of each CPU core.

Furthermore, it appears that the energy consumption is not linearly proportional to the applied frequencies. For instance, in Figure A.1a, the energy value for a single CPU core remains around 40 W until the highest two frequencies, where it rises to 77 and 81 W. This is supported by the remaining figures, which display contrasting jumps between energy values rather than a gradient. Even though applying higher frequencies increases energy consumption, a different frequency does not necessarily mean a different energy consumption value. Figure A.8a, for instance, depicts that from frequency 0.8 to 1.2 GHz, the energy spent by each CPU core is 2 W (with a total of 94



(a) 3 CPU cores.



(b) 5 CPU cores.

Figure 4.4: Experiments measuring energy consumption per set of CPU cores in CN_1.

W). When transitioning to 1.4 GHz, the energy spent by each CPU core rises to 3 W (with a total of 141 W) and remains that way until 1.9 GHz.

In summary, this profiling allowed some conclusions to be drawn. In $\boldsymbol{CN}_1$ there are 16 different frequencies available to apply (ranging from 0.8 to 2.801 GHz), and the maximum energy consumption is 384 W, obtained with all CPU cores (48) running at maximum frequency. The energy spent by a CPU core is influenced by the other active cores, and applying different frequencies does not necessarily mean different energy consumption values. The energy consumption is not linearly proportional to the frequencies.

4.3 Node-level energy control

This section evaluates the system's effectiveness in controlling energy consumption over the single compute node (*i.e.*, node-level energy control). $\boldsymbol{CN}_1$ is used for accommodating the *node controller* and executing the applications.

For these experiments, the energy and performance of the three selected applications were compared. Each application ran on specific CPU cores as detailed in Table 4.1, across three distinct scenarios: (i) Linux CPUFreq governors, representing the current state-of-practice in modern operating systems (the baseline), (ii) FINER with the priority energy control algorithm, and (iii) FINER with the fairness energy control algorithm. In all experiments, the applications were executed simultaneously. FINER was configured with a single node controller responsible for managing the energy consumption of each application.

Table 4.1: Applications running and corresponding CPU core affinity.

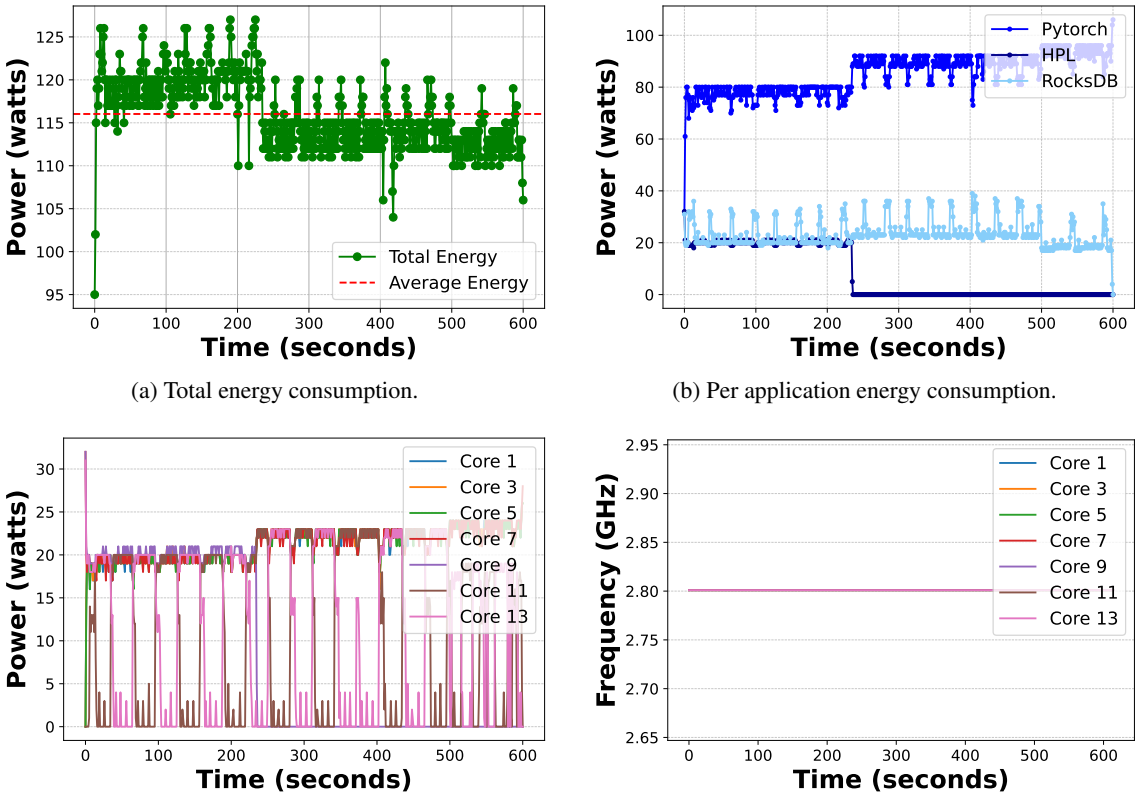
Application	**CPU core affinity**
PyTorch	1, 3, 5, 7
HPL	9
RocksDB	11, 13

4.3.1 Linux CPUFreq governors

This section explores the foundational assessment of the system, setting a benchmark for the evaluations of the *node controller*. This baseline evaluation serves as a critical reference point, providing an initial snapshot of the system's performance to understand its current capabilities and limitations.

To obtain the baseline evaluation, different experiments were conducted to represent various CPU frequencies at which applications were executed, applying the *performance*, *powersave*, and *ondemand* governors. Additionally, a fourth testing scenario considered setting the *performance* and *powersave* governors differently for each application, simulating the concept of energy priorities.

Performance governor Figures 4.5a, 4.5b, and 4.5c depict the overall energy consumption of $\boldsymbol{CN}_1$, each application's energy consumption, and each CPU core's energy consumption, respectively, under the performance governor. Running *PyTorch*, *HPL* and *RocksDB*, and applying the governor *performance* in all CPU cores (which sets the frequency to its highest value, as shown in Figure 4.5d) results in average power consumption of more than 115 W, with around 120 W up to 235 s and dropping to around 115 W after that time. This power consumption was distributed between the applications as shown in Figure 4.5b, with *PyTorch* spending around 80 W, *HPL* spending around 20 W and *RocksDB* spending between 20 and 32 W until 235 s. When 235 s elapsed since the test started, *HPL* terminated. After that period, *PyTorch* spent around 92 W,



(a) Total energy consumption.

(b) Per application energy consumption.

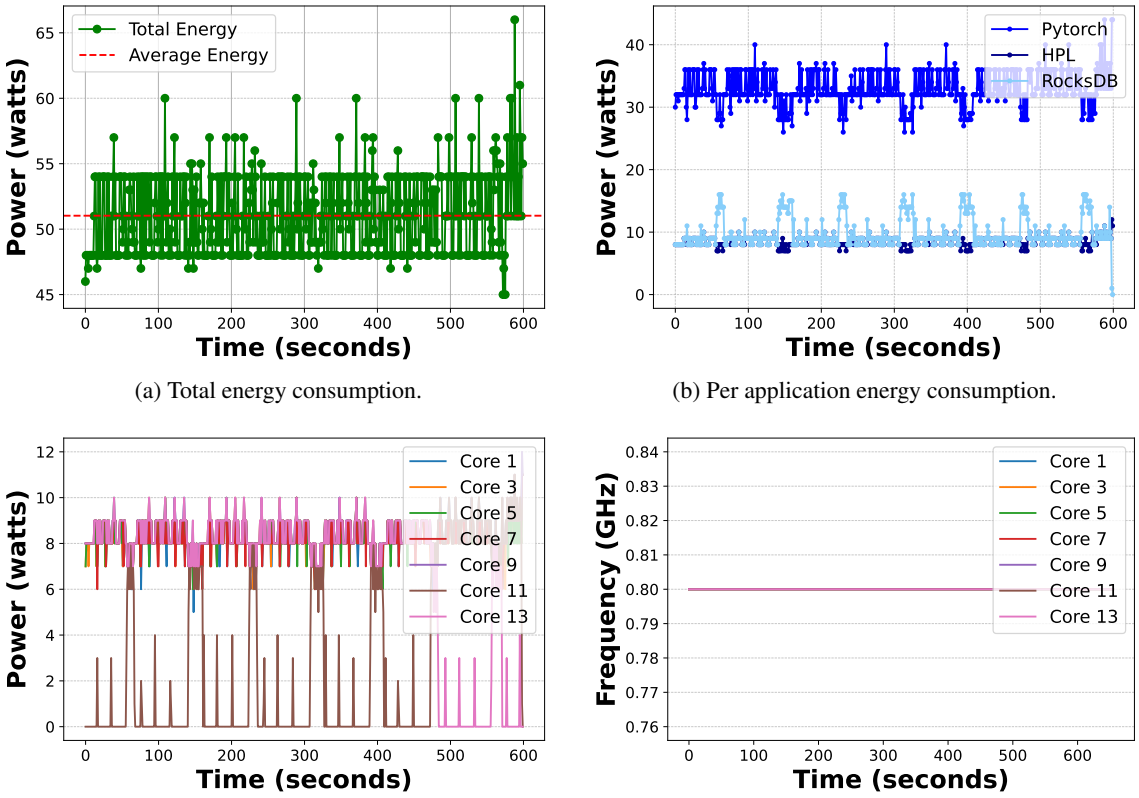
(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

Figure 4.5: Experiments for the performance governor under the node-level energy control setting.

and *RocksDB* spent between 23 and 39 W, until 500 s. The findings in Section 4.2.2 justify this increase in both applications' energy consumption. As explained, when the number of active CPU cores in the same NUMA node decreases, the energy consumption of each core reaches higher values. With the termination of *HPL*, the number of active cores decreases, wich leads to an increase in each active core energy consumption. After this period and until the end of the test, *PyTorch* consumption rose a little to around 96 W, and *RocksDB* dropped a little to between 17 and 35 W.

It is not surprising that *PyTorch* consumes much more energy than *HPL* and *RocksDB* since it uses 4 CPU cores and is a CPU-intensive application. Although *HPL* is also a CPU-intensive application, it only uses 1 CPU core, thus it consumes much less energy. *RocksDB* uses 2 CPU cores, but since it is more I/O-intensive, it presents spikes when the workers use the CPU and drops when some workers use the CPU and others use the disk. These conclusions are corroborated by Figure 4.5c. Although *PyTorch* consumes more energy than *HPL* when analyzing the energy consumption per CPU core, one can see that core 9 (running *HPL*) consumes more or less the same as cores 1, 3, 5, and 7 (running *PyTorch*). All these cores spend around 20 W each, with core 9 spending a little more than the rest. The *RocksDB* conclusions are also corroborated since it is shown that cores 11 and 13 (running *RocksDB*) take turns spending 0 W, which means they are not used at certain times due to few background tasks or some bottleneck in the storage.



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

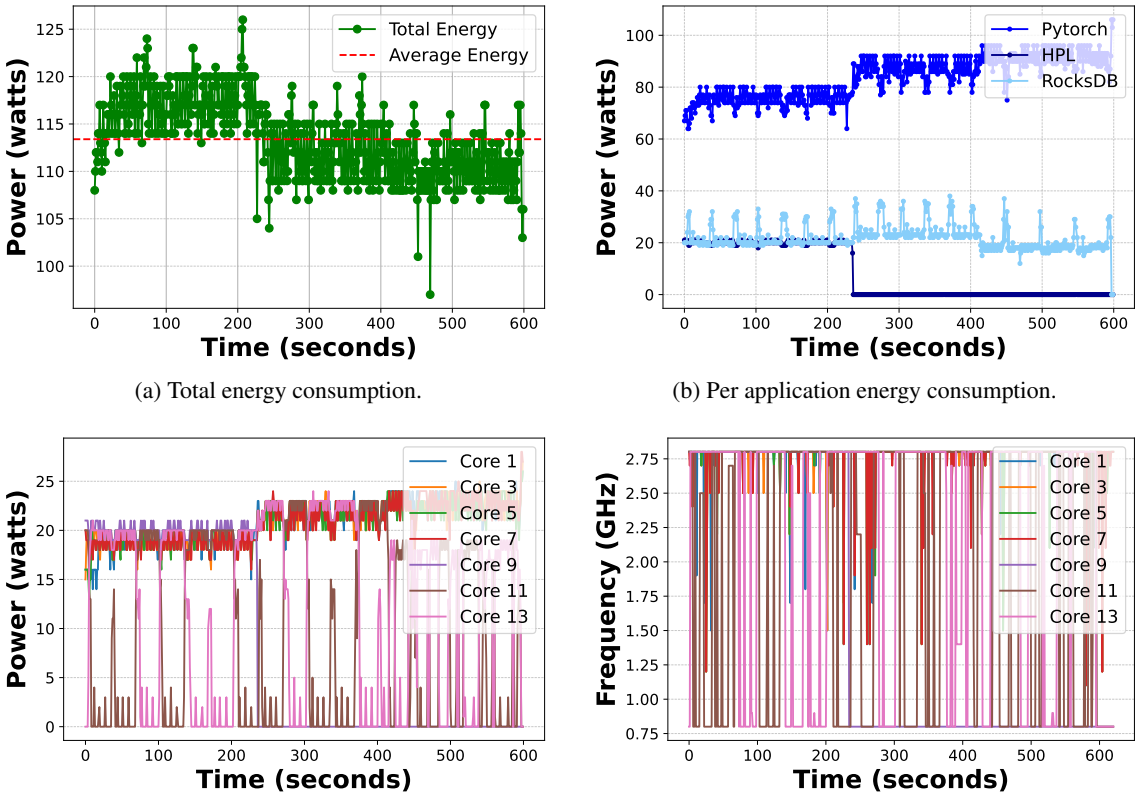
(d) Per CPU core applied frequency.

Figure 4.6: Experiments for the powersave governor under the node-level energy control setting.

Powersave governor Figures 4.6a, 4.6b, and 4.6c depict the overall energy consumption of $\boldsymbol{CN}_1$, the energy consumption of each application, and the energy consumption of each CPU core, respectively, under the *powersave* governor (which sets the frequency to its lowest value as seen in Figure 4.6d) results in an average power consumption of around 51 W (Figure 4.6a). This power consumption was distributed between the applications as shown in Figure 4.6b, with *PyTorch* spending around 35 W, *HPL* spending around 8 W, and *RocksDB* spending between 8 and 16 W. The conclusions about the applications' behavior and energy consumption are the same as in the last test, also confirmed by analyzing the energy per CPU core (Figure 4.6c).

These results show that applying the *powersave* governor significantly reduces overall energy consumption (of approximately 65 W) when compared to using the *performance* governor. *PyTorch*, *HPL*, and *RocksDB* decreased their energy consumption by approximately 45, 12, and 12 W, respectively. By running at the lowest frequency, *HPL* did not even terminate during the test execution.

Ondemand governor Figures 4.7a, 4.7b, and 4.7c depict the overall energy consumption of $\boldsymbol{CN}_1$, the energy consumption of each application, and the energy consumption of each CPU core, respectively, under the *ondemand* governor. By applying this governor (which adjusts the frequency dynamically based on the current load, initially ramping up to the highest frequency, and



(a) Total energy consumption.

(b) Per application energy consumption.

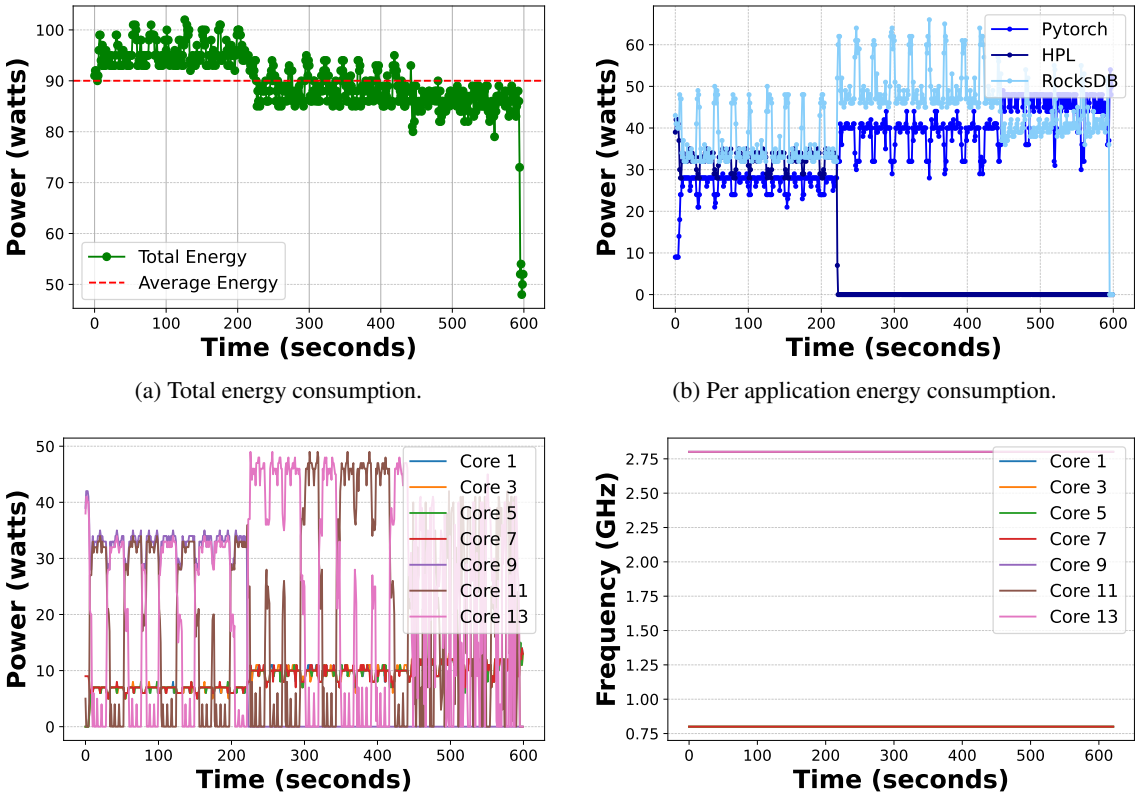
(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

Figure 4.7: Experiments for the ondemand governor under the node-level energy control setting.

then reducing it if idle time increases), one obtains the average consumption of around 113 W as shown in 4.7a. The values obtained in this evaluation were very similar (almost identical) to the ones obtained in Section 4.3.1. Figures 4.7b and 4.7c enabled the same conclusions to be drawn: *PyTorch* spends a higher amount of energy because uses more CPU cores, and the energy per CPU core is logical.

Mixed governors Figures 4.8a, 4.8b, and 4.8c depict the overall energy consumption of $\boldsymbol{CN}_1$, the energy consumption of each application, and the energy consumption of each CPU core, respectively, under a mixed use of the *performance* and *powersaving* governor. For this experiment, a scenario where *HPL* and *RocksDB* applications have high priority, and *PyTorch* has low priority was simulated. As such, the CPU cores of all high priority applications are configured with the *performance* governor, while *PyTorch*'s cores are set with the *powersave* governor. This evaluation resulted in an average power consumption of 90 W (Figure 4.8a). As in Section 4.3.1, the total energy consumption drops from around 95 W to around 85 at 220 s due to the termination of *HPL*. However, the energy spent by each application (Figure 4.8b) was very different from the last evaluations. Since *HPL* and *RocksDB* were running at the highest frequency, their energy consumption was higher than *PyTorch*'s (which was running at the lowest frequency). While *HPL* is running, it spends around 34 W, and *RocksDB* spends between 32 and 50 W, both application's



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

Figure 4.8: Experiments for the powersave and performance governors under the node-level energy control setting.

highest values when comparing the usage of all the governors. During the same period, *PyTorch* spends around 28 W, its lowest value. After *HPL* terminated, until 447 s, *PyTorch* spent between 28 and 44 W, and *RocksDB* spent between 45 and 66 W, the increase in both applications' energy consumption follows the same explanation as in the last tests. Afterward, *PyTorch* spent around 48 W, and *RocksDB* between 37 and 54 W.

When observing the energy consumption per CPU core (Figure 4.8c), it is obtained, as expected, higher power consumption values for the cores running *RocksDB* and *HPL*. CPU core 9 (running *HPL*) shows the highest values, above 30 W, while CPU cores 1, 3, 5, and 7 show values below 10 W. *RocksDB* (CPU cores 11 and 13) presents the same behavior as in the rest of the evaluations, alternating between the highest energy values and 0.

This evaluation clearly illustrates the importance of applying different frequencies when considering applications' varying priorities. For instance, when using the *performance* governor holistically, the *HPL* application, did not reach higher energy consumption values or terminated its execution earlier. Such results enhance the impact of coarse-grained energy control in applications' performance and energy management since even with these values, the average energy consumption was lower than with the *performance* and *demand* governors.

The performance and average energy efficiency of each application were also analyzed to

Table 4.2: Application efficiency running baseline evaluation.

Governor	Application	Application Performance	Avg Energy Efficiency
Performance	RocksDB	67788511 operations, 112708 ops/sec	204.828 W/MOps
	HPL	Time to solve the equations (seconds): 0.04, 26.75, 52.09, 26.94, 0.04, 1.52, 11.51, 52.58	589.875 W/Eq
	PyTorch	Trained until training epoch 4, iteration 21504 of 50000.	297.439 W/KIter
Powersave	RocksDB	26777402 operations, 44164 ops/sec	216.936 W/MOps
	HPL	Time to solve the equations (seconds): 0.19, 111.05, 216.27, 111.04, 0.19, 6.07, 47.05, did not finish the last one	unknown
	PyTorch	Trained until training epoch 2, iteration 7168 of 50000.	342.779 W/KIter
Ondemand	RocksDB	59354758 operations, 98685 ops/sec	219.746 W/MOps
	HPL	Time to solve the equations (seconds): 0.05, 27.14, 52.86, 27.35, 0.05, 1.54, 11.68, 53.07	589 W/Eq
	PyTorch	Trained until training epoch 4, iteration 8192 of 50000.	315.724 W/KIter
Powersave and Performance	RocksDB	82473758 operations, 137125 ops/sec	306.352 W/MOps
	HPL	Time to solve the equations (seconds): 0.04, 25.76, 50.18, 25.93, 0.04, 1.43, 11.05, 50.26	902.875 W/Eq
	PyTorch	Trained until training epoch 2, iteration 13312 of 50000.	335.592 W/KIter

evaluate FINER's impact. This analysis involved retrieving the total number of operations executed by *RocksDB*, as well as the number of operations per second, the time taken by *HPL* to solve each equation, and the number of training epochs and iterations completed by *PyTorch*. The average energy efficiency of the applications is reported in Watts per million operations (W/MOps) for *RocksDB*, Watts per equation (W/Eq) for *HPL*, and Watts per thousand iterations (W/KIter) for *PyTorch*.

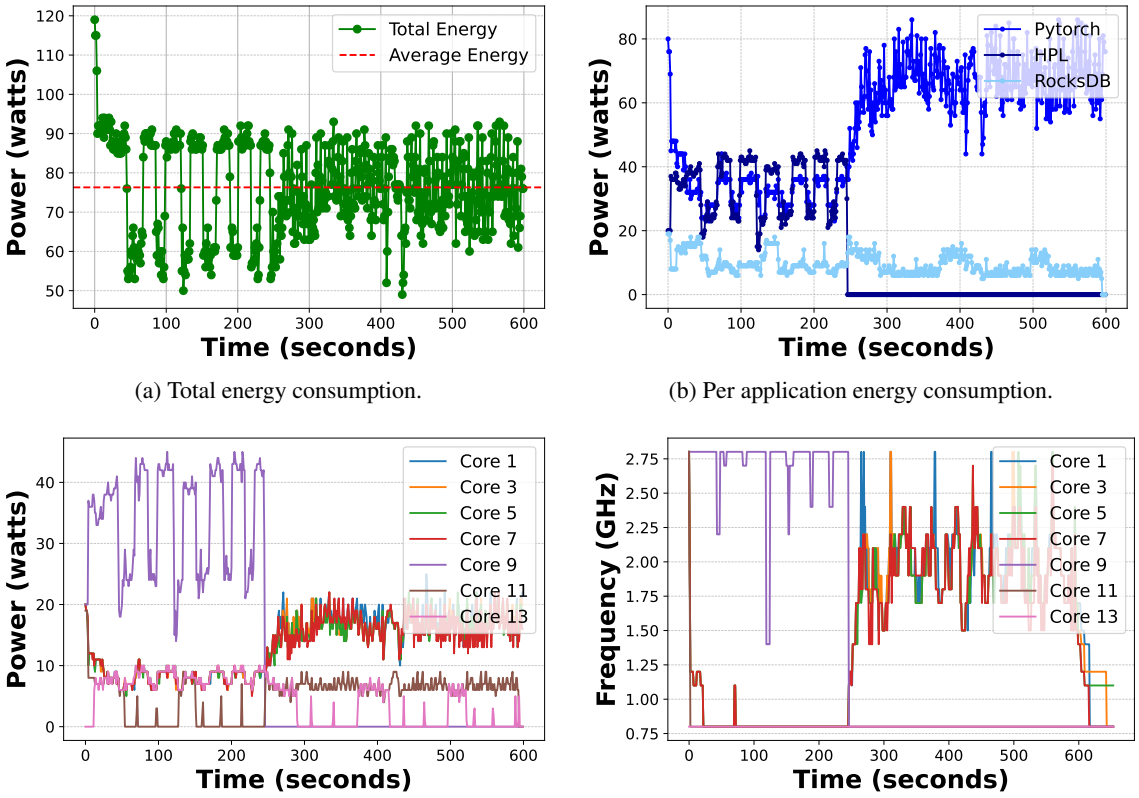
Table 4.2 shows each application's performance and average energy efficiency when performing the baseline evaluation. Applying the *performance* governor consisted of all applications running at maximum frequency, resulting in the best performance and highest energy consumption. When comparing the results with the *powersave* governor, one can notice the overall worse efficiency due to the reduced *performance*. *PyTorch*'s efficiency worsens (from 297.439 W/KIter on *performance* to 342.779 W/KIter on *powersave*), indicating its reliance on high CPU frequencies for optimal performance. *RocksDB*'s efficiency (216.936 W/MOps) and *HPL*'s inability to finish the last equation demonstrates the impact of reduced CPU frequency on I/O-bound and compute-bound applications, respectively. The results of applying the *ondemand* governor show that *PyTorch*'s efficiency (315.724 W/KIter) and *RocksDB*'s efficiency (219.746 W/MOps) are

comparable to the *performance* setting, showing minimal impact from dynamic frequency scaling. *HPL* maintains a similar efficiency (589 W/Eq), reinforcing that dynamic scaling effectively matches performance with demand. For the scenario with mixed governors, *PyTorch*'s efficiency (335.592 W/KIter) is worse than the *performance* setting (297.439 W/KIter) but better than the *powersave* setting (342.779 W/KIter), indicating a trade-off between energy savings and performance. *RocksDB* energy efficiency worsens to 306.352 W/MOps (when compared to the other scenarios where it was around 210 W/MOps) and *HPL* (902.875 W/Eq) improves the energy efficiency values compared to when the *powersave* governor is applied uniformly across all cores but worsens compared to the application of the *performance* governor (589.875 W/Eq).

4.3.2 Priority Control Algorithm

The performance and effectiveness of FINER for the node controller under the priority control algorithm are assessed in this section. For these experiments, the energy goal is set at 75 W (which is the maximum energy consumed by all applications). The previously collected profiling data (Section 4.2.2) for 7 CPU cores was analyzed to evaluate the *node controller*'s priority control algorithm. Running *PyTorch* on 4 cores, *RocksDB* on 2 cores, and *HPL* on 1 core within the same NUMA node resulted in the usage of 7 cores. This configuration's energy consumption ranged from 46 to 112 W, varying from the lowest to the highest frequency. Given this range, 75 W was selected as the target energy consumption value, as it represents an approximate midpoint between the minimum and maximum values. Moreover, for these experiments, the following priorities are set for each application: high priority for *HPL*, medium priority for *PyTorch*, and low priority for *RocksDB*. Appendix B presents additional experiments where applications have different priorities.

Figures 4.9a, 4.9b, and 4.9c depict the overall energy consumption of $\boldsymbol{CN}_1$, the energy consumption of each application, and the energy consumption of each CPU core, respectively, under the energy control priority algorithm with a 75 W goal. The server's total energy consumption varies, with oscillations mostly between 90 and 50 W, but the average (shown by the red line) is around 76 W, very close to the goal (Figure 4.9a). This average shows that FINER successfully manages the energy consumption to reach a predefined target. Since the configuration set *HPL* as having high priority, it is coherent that this application spent the highest amount of energy (up to 45 W) until it terminated at 247 s (Figure 4.9b). After terminating, it is evident that *PyTorch* started to spend a higher amount of energy, while *RocksDB* remained more or less the same, which makes sense due to the defined priorities (*PyTorch* had higher priority). Respecting the imposed limits and after completing the allocation of energy to applications with higher priority, FINER allows applications with lower priorities to take advantage of the energy that has become available. Even if an application with high priority is not actively working (and reports an energy consumption of 0 W) or the allocation of energy reaches the hardware component's physical maximum energy limit, the leftover energy is distributed through the remaining resources. Nevertheless, even after *HPL* ended its execution, the total energy consumption did not suffer major variations (Figure 4.9a), which had little impact on the overall average. Even though in the first 247 s of execution,



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

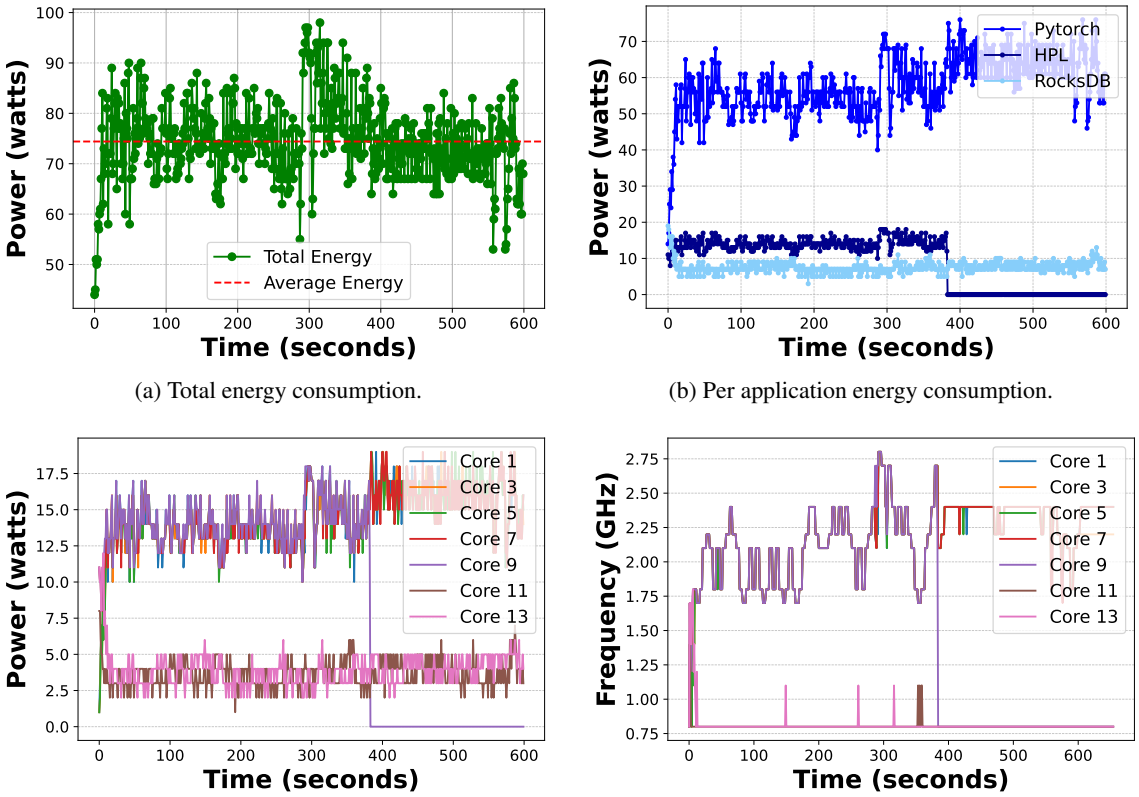
Figure 4.9: Experiments for the priority algorithm under the node-level energy control setting with a 75 W target and high priority in *HPL*.

PyTorch spends high energy consumption values, similar to *HPL* (Figure 4.9b), that is due to the differences between the number of used cores. Figure 4.9c confirms that *PyTorch*'s CPU cores are spending the same energy as the sum of *RocksDB*'s cores, and *HPL* CPU core is spending between 15 and 46 W. This evaluation confirmed that FINER respects the configured priorities in energy distribution.

The evaluation of the priority algorithm for the node controller included additional experiments by changing the configured priorities (presented in Appendix B). Such tests allowed the same conclusions to be drawn regarding FINER's effectiveness in targeting an energy consumption value and distributing energy respecting each component's priority. FINER effectively overcomes the challenge of targeting specific energy consumption limits, a feat unattainable with Linux CPUFreq governors, as demonstrated in the baseline experiments (Section 4.3.1). Additionally, FINER allows for more flexible priority energy distribution by incorporating low, medium, and high priority levels and proportionally allocating energy accordingly, unlike the restrictive approach of merely combining different governors.

4.3.3 Fairness Control Algorithm

The performance and effectiveness of FINER for the node controller under the fairness control algorithm are assessed in this section. For these experiments, within the 46 to 112 W range, three distinct energy goals were defined to evaluate the fairness control algorithm: one near the maximum value, one at the midpoint, and one near the minimum value. To assess the algorithm's behavior with a target at the midpoint of the energy range, the *node controller* was configured with a target of 75 W. Appendix B presents additional experiments where FINER was configured with different energy targets.



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

Figure 4.10: Experiments for the fairness algorithm under the node-level energy control setting with a 75 W target.

Figures 4.10a, 4.10b, and 4.10c depict the overall energy consumption of $\boldsymbol{CN}_1$, the energy consumption of each application, and the energy consumption of each CPU core, respectively, under the energy control fairness algorithm with a 75 W goal.

This configuration led to total energy consumption of up to 97 W, averaging 74 W (Figure 4.10a). Also in this test, the energy target was successfully managed. *PyTorch* got a bigger portion of energy compared to *HPL* and *RocksDB* (Figure 4.10b) with variations mostly between around 40 and 75 W, which makes sense as it uses more cores and uses more CPU (as mentioned in Section 4.2.1). Since *RocksDB*'s CPU cores use less CPU in proportion (because it also uses

other resources), this application was limited more strictly, consuming around 8 W. *HPL* spent around 15 W and took 383 s to terminate. *PyTorch*'s and *HPL*'s cores consume nearly the same energy, approximately 14 W until the end of HPL's execution, while *RocksDB*'s cores consume approximately 4W each. When *HPL* terminates, *PyTorch*'s cores' energy exhibits a minimal energy increase, consuming around 16 W. The energy distribution throughout this test seems to be fair, as the applications using more CPU in proportion to their CPU cores (*HPL* and *PyTorch*) were given higher energy values.

While achieving a similar total average energy consumption, this evaluation produced a markedly different energy distribution per application compared to the priority algorithm (Section 4.3.2). These differences arise from basing the distribution on CPU usage proportion rather than administrator-assigned application priorities. Each method is suited to different contexts, depending on the applications' specific requirements and the infrastructure's overarching goals. Compared to the baseline experiments (Section 4.3.1) using the *performance*, *powersave*, and *ondemand* governors (where the concept of priority is not applicable), the most significant difference is that *HPL* and *RocksDB* consume similar amounts of energy in the baseline experiments. However, *HPL* is assigned higher energy values in this experiment due to its proportionally higher CPU usage. These results also highlight the inability of Linux CPUFreq governors to target specific energy consumption limits.

Experiments targeting energy consumption near the maximum value (100 W) and the minimum value (63 W) are detailed in Appendix B. These evaluations demonstrate that FINER effectively meets the specified energy consumption limits and fairly distributes energy across different applications.

Table 4.3: Application efficiency during node controller evaluation.

Algorithm	Application	Application Performance	Avg Energy Efficiency
Priority	RocksDB	18560999 operations, 30842 ops/sec	300.415 W/MOps
	HPL	Time to solve the equations (seconds): 0.06, 26.90, 58.53, 30.67, 0.04, 1.47, 14.66, 56.67.	1040 W/Eq
	PyTorch	Trained until training epoch 2, iteration 26624 of 50000.	413.121 W/KIter
Fairness	RocksDB	15198210 operations, 25118 ops/sec	297.666 W/MOps
	HPL	Time to solve the equations (seconds): 0.19, 46.97, 94.44, 41.23, 0.07, 2.09, 19.86, 78.43.	667 W/Eq
	PyTorch	Trained until training epoch 3, iteration 26624 of 50000.	272.942 W/KIter

Table 4.3 presents the *node controller*'s application performance and average energy efficiency reported while evaluating the priority and fairness algorithms. With low priority, *RocksDB*'s energy efficiency worsens slightly to 300.415 W/MOps (compared with the *performance*, *powersave* and *ondemand* governors, with 204.828 W/MOps, 216.936 W/MOps, and 219.746 W/MOps,

respectively) indicating increased energy consumption per operation. As a high priority application, *HPL* shows significantly higher energy consumption (when compared to 589.875 W/Eq, unknown, and 589 W/Eq, obtained with *performance*, *powersave* and *ondemand* governors, respectively) with an efficiency of 1040 W/Eq. Medium priority *PyTorch*, also worsens in efficiency (from 297.439, 342.779, and 315.724 W/KIter with *performance*, *powersave* and *ondemand* governors, respectively) to 413.121 W/KIter, the worst energy efficiency so far for this application. The energy efficiency of this scenario is similar to the one obtained in the baseline evaluation with the mixed governors, which resulted in 306.352 W/MOps for *RocksDB*, 902.875 W/Eq for *HPL*, and 335.592 W/KIter for *PyTorch*. For the fairness scenario, the energy efficiency of the three applications improved (when compared to the priority one). *PyTorch*'s average energy efficiency was 272.942 W/KIter, the best so far, while *RocksDB*'s, and *HPL*'s average energy efficiencies were medium range values (297.666 W/MOps and 667 W/Eq, respectively).

4.4 Rack-level energy control

FINER's effectiveness in controlling energy consumption at different granularities, specifically at the rack-level, is evaluated in this section. For these experiments, a testing environment is simulated where the *rack controller* manages the energy consumption of four servers, each executing with its own *node controller*. Due to the limitations of the testing infrastructure, $\boldsymbol{CN}_2$ is used for accommodating the *rack controller*, while $\boldsymbol{CN}_1$ is partitioned into four parts, representing each of the targeted servers. The energy and performance of the four applications (each monitored by an individual *node controller*) were compared, across distinct scenarios: (i) FINER with the priority energy control algorithm, and (ii) FINER with the fairness energy control algorithm. In all experiments, the applications were executed simultaneously. This evaluation does not include a baseline, as the servers are simulated and the applications are running on the same compute node ($\boldsymbol{CN}_1$). A baseline experiment for this evaluation would be similar to the one conducted for the node controller (Section 4.3.1). Table 4.4 presents the logical servers, the applications executed on each of them, as well as their CPU core affinity and NUMA node.

Table 4.4: Applications running, corresponding CPU core affinity, and server identifier.

Server	**Application**	**CPU core affinity**	NUMA node
1	PyTorch	1, 3, 5, 7	1
2	HPL_1	14	0
3	HPL_2	16	0
4	RocksDB	10, 12	0

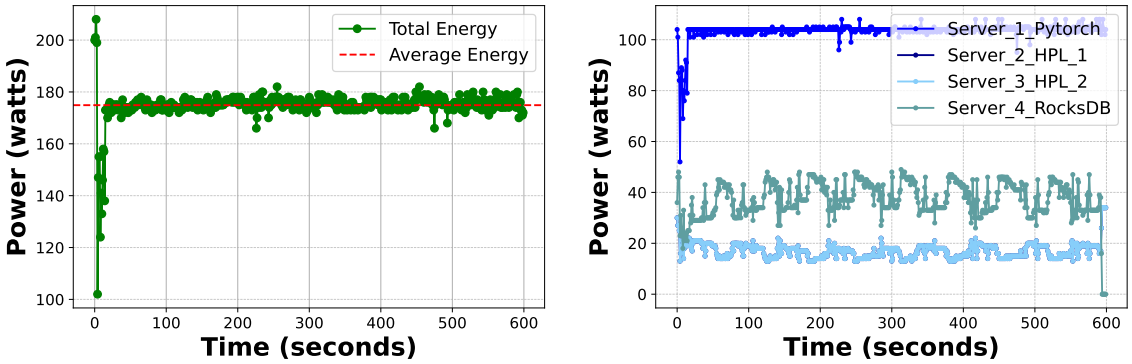
Although using more CPU cores per server would enrich the evaluation and better simulate real servers, this configuration was used in an attempt to balance the number of active CPU cores per NUMA node, taking into account the findings in Section 4.2.2. Since the number of active cores per NUMA node influences the energy consumption of those cores and applications, this setup

was planned to use 4 CPU cores in one NUMA node (cores 1, 3, 5, and 7 - running *PyTorch*), and 4 CPU cores in the other NUMA node (cores 10, 12, 14, and 16 - running *RocksDB*, HPL_1 and HPL_2).

In this evaluation, the number of problem sizes (N) in *HPL* was expanded to 16, with the problem sizes being repeated to extend the runtime. This adjustment was made to prevent premature termination, as this evaluation simulates a workload across 4 independent servers and incurs higher energy consumption due to utilizing multiple NUMA nodes. Consequently, *HPL*'s performance showed improvement compared to the evaluation with the *node controller*.

4.4.1 Priority Control Algorithm

As in the *node controller* evaluation, the evaluation of the priority control algorithm with the *rack controller* consisted of applying a middle-range energy target for different priority scenarios. Given that the energy spent by 4 CPU cores in the same NUMA node obtained in profilling 4.2.2 was between 44 and 96 W, the machine's energy was expected to be between 88 and 192 W. Thus, these experiments were conducted with an energy target of 170 W. For this experiment, the following priorities are considered: Server 1 (*PyTorch*) has high priority, Server 4 (*RocksDB*) has medium priority, and the remaining servers (running *HPL* applications) have low priority.



(a) Total energy consumption.

(b) Per server energy consumption.

Figure 4.11: Experiments for the priority algorithm under the rack-level energy control setting with a 170 W target and high priority in *PyTorch*.

Figures 4.11a and 4.11b depict the overall energy consumption of the rack and each of the logical servers, respectively, under FINER's priority control algorithm. As observed, the total energy consumption is stable throughout the overall execution, averaging 174 W. Such a scenario reveals a clear difference between the defined priorities, with *PyTorch* spending above 100 W, *RocksDB* spending around 40 W, and both *HPL*s spending around 19 W (Figure 4.11b). The line corresponding to HPL_1 is at the same level as HPL_2, thus it can not be seen in the figure.

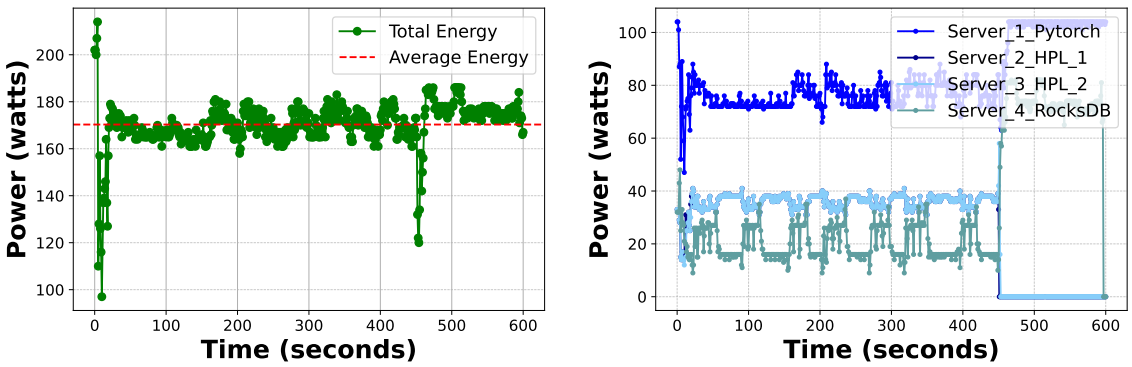
This outcome highlights the hierarchical effectiveness of FINER. The rack-level controller successfully aggregates and manages the energy consumption of multiple compute nodes, ensuring that the total rack energy consumption remains within the targeted limit. The system adheres

to the defined priorities across different levels of the hierarchy, demonstrating a clear and consistent energy distribution aligned with priority configurations. *PyTorch*, being the highest priority application and using more CPU cores, consumes the most energy, followed by *RocksDB* and then the *HPL* applications. The stability in total energy consumption, averaging 174 W, indicates that the *rack controller* is effectively coordinating with node-level controllers to maintain the desired energy target. This coordination ensures that the overall system remains within the predefined energy budget even as individual application loads fluctuate. Also, as in the *node controller*, if a high priority server is not actively working (and reports an energy consumption of 0 W) or the allocation of energy reaches the server's physical maximum energy limit, the leftover energy is distributed through the remaining nodes. This level of hierarchical control and stability is unattainable with the traditional methods that present partial visibility.

Appendix C presents the results for different priority combinations. These tests confirm the effectiveness of FINER's *rack controller* in implementing the priority algorithm and successfully targeting energy consumption limits while adhering to the predefined priorities.

4.4.2 Fairness Control Algorithm

The effectiveness of FINER under the fairness control algorithm for rack-level energy management is assessed in this section. For this experiment, the energy goal is set near the maximum value, namely 180 W. Annex C shows further experiments with other energy goals, specifically 130 W and 100 W.



(a) Total energy consumption.

(b) Per server energy consumption.

Figure 4.12: Experiments for the fairness algorithm under the rack-level energy control setting with a 180 W target.

Figures 4.12a and 4.12b depict the overall energy consumption of the rack and each of the logical servers, respectively, under FINER's fairness control algorithm. With a high energy target, the total energy consumption oscillates mostly between 158 and 186 W, averaging 170 W (Figure 4.12a). This stability in total energy consumption indicates effective hierarchical management by FINER, where the rack-level controller coordinates with node-level controllers to maintain the overall energy target while fairly distributing energy among the servers based on their aggregate

CPU usage. Specific observations about each logical server reveal the nuanced energy distribution managed by the rack-level controller (similar to the results shown in Section 4.3.3). The server running *PyTorch*, consistently consumes the most energy, averaging around 76 W, due to its intensive CPU usage and larger number of active cores. Servers running *HPL* share similar energy consumption (middle-range energy values), around 37 W each, reflecting their balanced CPU usage. The server running *RocksDB* shows the lowest energy consumption, fluctuating mostly between 12 and 37 W (Figure 4.12b) highlighting the proportional energy allocation based on its mixed resource profile. These results align with the application's resource profiles (Section 4.2.1), where *PyTorch* demands more CPU power, *HPL* utilizes the CPU intensively but on fewer cores, and *RocksDB* uses both CPU and disk resources.

Experiments targeting different energy consumption limits, including near the minimum value (100 W) and at the middle point (130 W), are detailed in Appendix C. These evaluations further demonstrate FINER's effectiveness in meeting specified energy consumption limits and fairly distributing energy across servers.

Table 4.5: Application efficiency during rack controller evaluation.

Algorithm	**Application**	**Application Performance**	**Avg Energy Efficiency**
Priority	RocksDB	33218213 operations, 55228 ops/sec	671.198 W/MOps
	HPL_1	Time to solve the equations (seconds): 0.05, 82.19, 214.65, 110.21, 0.18, 6.00, 43.07, did not finish the rest.	unknown
	HPL_2	Time to solve the equations (seconds): 0.05, 86.30, 214.51, 110.17, 0.18, 6.00, 39.40, did not finish the rest.	unknown
	PyTorch	Trained until training epoch 3, iteration 27648 of 50000.	483.917 W/KIter
Fairness	RocksDB	40030864 operations, 66517 ops/sec	493.544 W/MOps
	HPL_1	Time to solve the equations (seconds): 0.05, 31.67, 49.95, 25.70, 0.04, 1.41, 10.94, 50.00, 0.04, 25.69, 49.94, 25.72, 0.04, 1.41, 10.94, 49.93.	995.5 W/Eq
	HPL_2	Time to solve the equations (seconds): 0.05, 32.33, 49.96, 25.68, 0.04, 1.41, 10.94, 50.01, 0.04, 25.67, 49.94, 25.71, 0.04, 1.41, 10.94, 49.94.	999 W/Eq
	PyTorch	Trained until end of training epoch 2, 100000 iterations.	492.78 W/KIter

Table 4.5 presents the *rack controller*'s application performance and average energy efficiency reported while evaluating the priority and fairness algorithms. Similar to the *node controller*'s results, the *PyTorch*'s energy efficiency with the priority control algorithm showed the worst result

of all the evaluations with 483.917 W/KIter, reflecting worse energy utilization. *RocksDB*, with medium priority, shows a relatively high energy efficiency (671.198 W/MOps). *HPL* applications, unable to complete the task, suggest that lower priority significantly hampers their performance. The *HPL* applications demonstrate very similar equation completion times. For the fairness control algorithm, both *HPL*s resulted in very similar performance and energy efficiency (of 995.5 W/Eq and 999 W/Eq), *RocksDB* and *PyTorch* presented energy efficiencies of 493.554 W/MOps and 492.78 W/KIter, respectively.

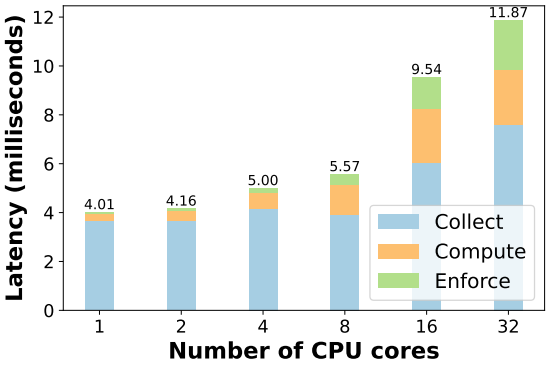
While it is important to analyze the energy efficiency and performance impact of FINER on each application, a slight decrease in these metrics is not particularly concerning. The primary objective of the proposed solution is to limit overall energy consumption.

4.5 Sensitivity Analysis

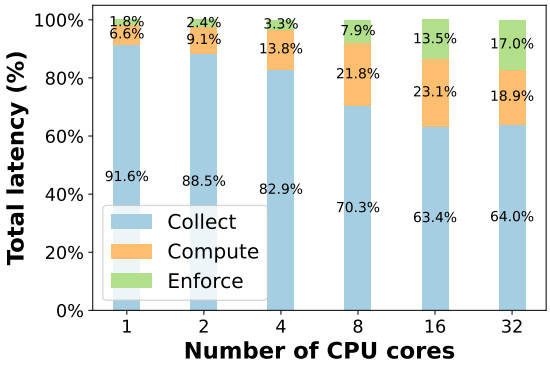
To understand the impact of the control overhead of adding more components to manage, this section provides a latency analysis of the control logic of node and rack controllers.

4.5.1 Overhead of managing more CPU cores

First, the impact of managing more CPU cores is analyzed. In this experiment, the latency taken by the node controller's control logic (*i.e.*, collect, compute, and enforce) is measured when the number of CPU cores is increased. Experiments were conducted over $\boldsymbol{CN}_1$, with the number of managed CPU cores ranging from 1 up to 32. The node controller was configured with the fairness control algorithm. Similar performance results are reported for the priority control algorithm.



(a) Average latency of each control cycle.



(b) Latency proportion of the collect, compute, and enforce phases.

Figure 4.13: Experiments for measuring the node controller overhead with increasing number of managed CPU cores.

Figures 4.13a and 4.13b depict the latency of each control cycle and the proportion of time spent at each control phase, respectively. The average latency of each control iteration (4.13a) ranges between 4.01 and 11.87 ms, increasing with higher numbers of CPU cores. Each control

phase (*collect*, *compute*, and *enforce*) is also represented in the same figure, with the intervals of each phase increasing with higher sets of cores.

By analyzing Figure 4.13b, one can deduce the latency proportion of each phase. While the *collect* phase spends most of the total latency (between 63.4% and 91.6%), *computing* varies between 6.6% and 23.1%, and the *enforce* is the fastest phase, going up to 17%. As the number of CPU cores increases, the *collect* phase seems to diminish, while the *compute* and *enforce* phases augment proportionally. From 8 CPU cores, the *compute* phase converges to around 20%.

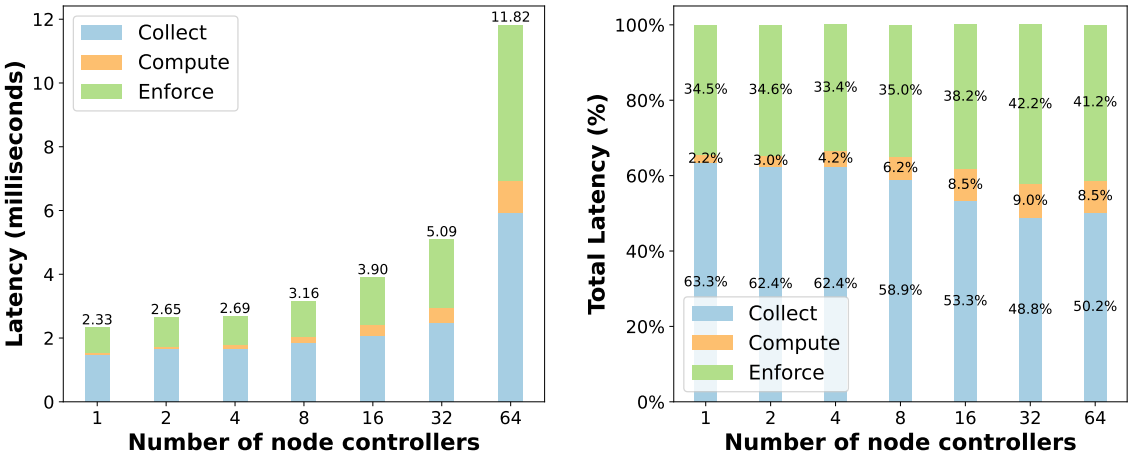
The increases in latency for each phase can be attributed to specific mechanisms within the *node controller*'s control logic. During the *collect* phase, the rise in latency is primarily due to the increased workload associated with retrieving power usage data from multiple CPU cores. This involves the RAPL interface accessing a higher number of files to gather the necessary information for each core, resulting in higher messaging payloads. Consequently, as the number of CPU cores grows, the time required to collect this data correspondingly increases. In the *enforce* phase, the latency surge is linked to the necessity of sending multiple rules to the data plane. With more CPU cores to manage, the control logic must dispatch a greater number of enforcement policies, leading to a larger payload and, consequently, longer processing times. On the other hand, the *compute* phase's latency increase is due to the more complex calculations needed to balance the energy fairly or prioritize specific applications across a larger number of cores. Thus, the growing complexity and data handling requirements in each phase collectively contribute to the overall rise in control latency as the number of managed CPU cores escalates.

4.5.2 Overhead of managing more node controllers

Lastly, the impact of managing more node controllers is analyzed. In this experiment, the latency taken by the rack controller's control logic (*i.e.*, collect, compute, and enforce) is measured when the number of node controllers is increased. Experiments were conducted by running the node controllers over $\boldsymbol{CN}_1$, and the rack controller over $\boldsymbol{CN}_3$, with the number of managed node controllers ranging from 1 up to 64. The rack and node controllers were configured with the fairness control algorithm. Similar performance results are reported for the priority control algorithm. Although the experiment would be enriched with higher numbers of node controllers (for instance, 128), the results would not be statistically correct due to the limitations of the machine that, even with hyper-threading activated, reached a maximum of 96 CPU(s).

As shown in Figures 4.14a and 4.14b, the average latency of each control iteration ranges between 2.33 and 11.82 ms, increasing with the number of *node controllers*. The latency of each control phase (*collect*, *compute*, and *enforce*) is also represented in the same Figure, with the latencies of each phase increasing with higher numbers of *node controllers*.

The proportion of each phase (*collect*, *compute*, and *enforce*) is depicted in Figure 4.14b. While the *collect* phase spends most of the total latency (between 48.8% and 63.3%), *computing* is the fastest phase, going up to 9%, and the *enforce* phase varies between 33.4% and 42.2%. With more extensive sets of node controllers, the *collect*, *compute*, and *enforce* phases diminish,



(a) Average latency of each control cycle.

(b) Latency proportion of the collect, compute, and enforce phases.

Figure 4.14: Experiments for measuring the rack controller overhead with increasing number of managed node controllers.

augment, and increase proportion accordingly. However, the latency proportion of each phase appears to converge starting from 32 node controllers onward.

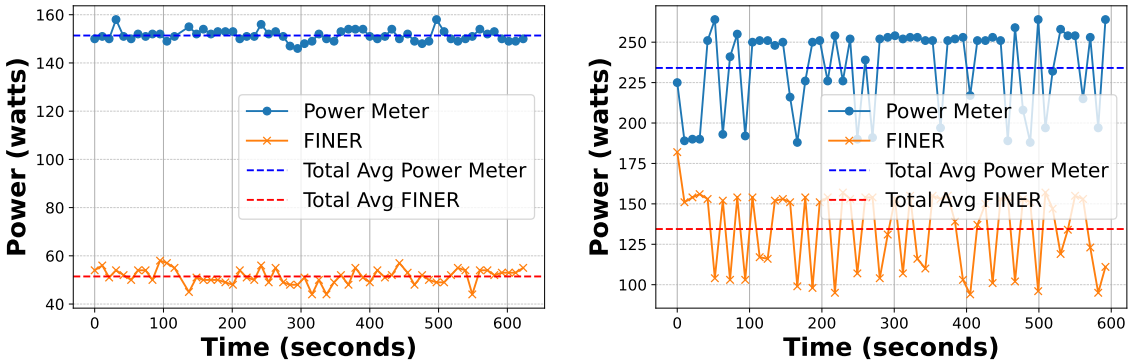
The increases in latency for each phase of the *rack controller*'s control logic can be attributed to specific underlying mechanisms. During the *collect* phase, the observed rise in latency is primarily due to the need to aggregate data from a larger number of *node controllers*. This involves network communication overhead, where more messages need to be sent and received to collect the necessary performance metrics from each *node controller*, leading to higher payload sizes and increased processing times. In the *enforce* phase, the latency surge is linked to the complexity of distributing control policies across multiple *node controllers*. As the number of *node controllers* grows, the rack controller must send out a greater number of enforcement rules, resulting in larger message payloads and longer transmission times. Conversely, the *compute* phase experiences an increase in latency due to the additional computational burden of processing and optimizing the control algorithms across a larger set of *node controllers*. This includes more complex decision-making processes and increased data-handling requirements. Thus, the combined effect of increased network communication overhead and computational complexity in each phase contributes to the overall rise in control latency as the number of managed node controllers increases.

When comparing the results of the *node* and *rack controllers*, one concludes that the range of latency values is the same. The proportion of each phase's latency varies, but it makes sense that in the *rack controller*, the *collect* and *enforce* phases spend more time since they exchange messages with the *node controllers* over the network.

4.5.3 Software-based vs. Hardware-based energy measurements

This section focuses on validating the accuracy and reliability of the results and conclusions regarding energy consumption. A power meter was employed to measure the hardware power spent

to ensure the FINER measurement module functions correctly. To achieve this, the power consumption measured with the power meter was retrieved during the evaluation of Section 4.3.2(Figure 4.15b). This included the test where *RocksDB* was given high priority (as shown in Appendix B). Furthermore, the differences between the tools were analyzed when nothing was running in the machine. Figure 4.15a depicts the results.



(a) When nothing is running.

(b) During *node controller* evaluation with high priority in *RocksDB*.

Figure 4.15: Experiments for comparing power meter and FINER energy measurements in CN_1.

When analyzing the plots, the first thing to notice is the noticeable difference between the values measured with the power meter and FINER's monitoring module, being approximately 100 W. The reasons behind this are threefold. First, the FINER's monitoring module does not account for the energy consumed by CPU cores that are in an *idle* state. As discussed in Section 2.1, CPU cores can consume energy even in *idle* state. Second, FINER's monitoring module only accounts for the energy consumed by CPU cores. However, other resources present in the system also consume energy, such as memory, disk, network, cooling, etc. The power meter, on the other hand, measures the energy consumption of the overall server since it controls the energy provided by the PDU. Finally, FINER's monitoring module is based on RAPL, a software-based energy monitoring tool, which can contain measurement errors.

However, when comparing the averages between Figure 4.15a (where nothing was running in the machine) and 4.15b (where all the applications and the FINER were running), one can notice that the differences between the tools' averages remain consistent. This consistency indicates that the monitoring module can reliably track relative changes in energy consumption, providing a useful tool for assessing power usage trends.

4.6 Summary

The experimental evaluation demonstrates significant reductions in energy consumption across various scenarios, indicating the effectiveness of the FINER energy control system. The fairness control algorithm effectively balances energy distribution across CPU cores and nodes, where

each component receives energy allocation commensurate with its resource usage, while the priority algorithm successfully allocates each component energy target considering their priority and ensuring high priority components take precedence.

The system achieves substantial energy savings, demonstrating specific energy consumption targets can be reached with only minor performance trade-offs. Generally, when reducing the applications' energy consumption the performance decreases, and the average energy efficiency worsens. Such trade-offs are inherent, as achieving significant energy reduction while maintaining peak performance is exceedingly challenging. However, this challenge is orthogonal to this work, as FINER focuses on reducing energy consumption with specific target limits, at different granularity levels. The experiments were validated using a hardware power meter, and FINER is considered to reliably track relative changes in energy consumption.

The FINER latency remains within acceptable bounds. The *node controller* shows an average cycle latency ranging from 4.01 to 11.87 ms when managing up to 32 CPU cores. The *collect* phase accounts for the majority of this latency, followed by the *compute* and *enforce* phases. Similarly, the *rack controller* exhibits average cycle latencies between 2.33 and 11.82 ms for up to 64 *node controllers*, with the *collect*, *compute*, and *enforce* phases contributing varying proportions to the total latency. The FINER demonstrates scalability, effectively managing an increasing number of CPU cores and node controllers while maintaining efficient operation and latency within acceptable ranges. For instance, FINER was able to scale from the management of 8 CPU cores to 32 CPU cores with a minimal increase in control latency, being able to handle larger deployments.

In conclusion, FINER is a viable solution for limiting energy usage in data centers across the infrastructure hierarchical organization. It contributes to sustainable and cost-effective data center operations by effectively balancing energy consumption.

Chapter 5

Conclusion

The concern of controlling energy consumption from a sustainable and cost-saving perspective emerges in the ever-expanding landscape of large-scale computing infrastructures. Despite continuous efforts to optimize hardware and employ energy-saving techniques such as workload consolidation and DVFS, data centers account for a substantial portion of global electricity usage. The existing energy management solutions remain suboptimal due to two main issues. First, current strategies manage energy consumption with coarse granularity, which means that all components (namely, applications, CPU cores, servers, and racks) are treated equally, inhibiting the ability to enforce energy priorities or ensure fairness. Second, current strategies actuate with partial visibility of resources, where energy management is made uncoordinated across the overall infrastructure, which can lead to suboptimal energy efficiency and potentially impact the performance of different parts of the infrastructure. The absence of adaptive, fine-grained energy control mechanisms prevents optimal energy utilization and system performance, highlighting the need for strategies that can dynamically adjust to changing workloads and application priorities.

To overcome these problems, FINER is proposed as a novel energy management system that orchestrates the energy consumption of large-scale computing infrastructures in a hierarchical, fine-grained, and adaptive way. Such a system follows a decoupled architecture composed of a data plane (responsible for managing hardware components and enforcing energy rules) and a control plane (responsible for monitoring and distributing the energy targets). The proposed hierarchy comprises cluster, rack, and node controllers that ensure efficient communication and fine-tuned decision-making while accommodating the specific needs of individual applications, servers, and clusters, overcoming the limitations of partial infrastructure visibility and promoting efficient energy allocation. The control logic is based on a feedback control loop to dynamically adjust the energy consumption to system and workload changes. Accordingly, two new energy control algorithms were proposed. The fairness algorithm ensures equitable energy distribution based on resource usage, while the priority algorithm allocates energy according to node priorities and consumption. FINER is particularly effective in large-scale data centers with high computational demands, focusing on CPU energy management. This focus aligns with the shift towards ARM-based supercomputers, as in Riken's Fugaku and MACC's Deucalion, ensuring its relevance

for future high-performance systems.

To demonstrate the applicability and feasibility of FINER in managing energy limits in large-scale infrastructures, a comprehensive experimental testbed was conducted. The conducted experiments demonstrate that FINER can (1) employ different energy control algorithms over the infrastructure, namely using the priority and fairness control algorithms; (2) dynamically adapt to system changes, distributing leftover energy available in the managed components to those more in need; (3) control energy consumption at different levels of granularity, namely at compute node and rack levels; and (4) control different components (*i.e.*, CPU cores, compute nodes) with an acceptable control cycle latency.

In conclusion, FINER is a viable solution for hierarchical energy management in data centers, being a first step towards a more sustainable and cost-effective management of large-scale infrastructures.

5.1 Future Work

In this section, direct research paths opened by FINER, as well as future improvements to be made over the current work are discussed.

Expand the domain of controlled resources Currently, FINER primarily focuses on CPU energy management. Future developments could aim to incorporate control mechanisms for additional hardware resources such as GPUs, memory, and even other critical components, such as cooling. By extending the scope of resources managed, the system can provide more comprehensive energy optimization and improve overall efficiency. Such improvements would be implemented by extending the data plane to integrate the enforcement of energy rules in different hardware components, the monitoring module to collect the resources' usage proportions, and the control algorithms to consider these additional metrics.

Improve the scalability of the control hierarchy To handle larger and more complex data center infrastructures (*e.g.*, made of thousands of compute nodes), it would be interesting to expand the design (and current implementation) of FINER to support additional control levels. This includes increasing the number of control layers and enhancing the coordination among node controllers, rack controllers, and cluster controllers. A more scalable implementation would enable the system to manage significantly larger deployments, maintaining low latency and efficient operation across all hierarchy levels.

High-level policy specification Future iterations of FINER could incorporate high-level metrics and objectives, such as reducing the data center's carbon footprint by a specific percentage. The system would leverage its fine-grained control and adaptive algorithms to meet these broader environmental and sustainability goals. By setting and achieving such targets, FINER can contribute to greener and more sustainable data center operations.

References

[1] amd-pstate cpu performance scaling driver, June 2024. Available at https://docs.kernel.org/admin-guide/pm/amd-pstate.html.

[2] Cpu performance scaling, June 2024. Available at https://docs.kernel.org/admin-guide/pm/cpufreq.html.

[3] Fugaku | riken center for computational science, June 2024. Available at https://www.r-ccs.riken.jp/en/fugaku/.

[4] grpc: A high performance, open source universal rpc framework, June 2024. Available at https://grpc.io/.

[5] Nvidia:nvml api reference guide vr550, June 2024. Available at https://docs.nvidia.com/deploy/nvml-api/.

[6] Rocksdb: A persistent kay-value store for fast storage environments, June 2024. Available at https://rocksdb.org/.

[7] Running average power limit energy reporting / cve-2020-8694 , cve-2020-8695 / intel-sa-00389, June 2024. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[8] Squeezenet, June 2024. Available at https://github.com/forresti/SqueezeNet.

[9] Vista: A new ai-focused supercomputer for the open science community, June 2024. Available at https://tacc.utexas.edu/news/latest-news/2023/11/13/vista-a-new-ai-focused-supercomputer-for-the-open-science-community/.

[10] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab. Hamid, Muhammad Shiraz, Abdullah Yousafzai, and Feng Xia. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of Network and Computer Applications*, 52:11–25, 2015.

[11] Kazi Main Uddin Ahmed, Math H. J. Bollen, and Manuel Alvarez. A review of data centers energy consumption and reliability modeling. *IEEE Access*, 9:152536–152563, 2021.

[12] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[14] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. Treehouse: A case for carbon-aware datacenter software, 2022.

[15] Antoine Paul Petitet, Clint Whaley, and Jack Dongarra. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers, June 2024. Available at https://www.netlib.org/benchmark/hpl/.

[16] Mauricio Arregoces and Maurizio Portolani. *Data center fundamentals*. Cisco Press, 2003.

[17] Jason Barkes, Marcelo R Barrios, Francis Cougard, Paul G Crumley, Didac Marin, Hari Reddy, and Theeraphong Thitayanun. Gpfs: a parallel file system. *IBM International Technical Support Organization*, 1998.

[18] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019.

[19] W. Lloyd Bircher and Lizy K. John. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, page 327–338, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Pat Bohrer, Elmootazbellah N Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case for power management in web servers. In *Power aware computing*, pages 261–289. Springer, 2002.

[21] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 171–182, 2001.

[22] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2012.

[23] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, GA, October 2000. USENIX Association.

[24] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, page 86–97, New York, NY, USA, 2003. Association for Computing Machinery.

[25] Mohak Chadha, Eishi Arima, Amir Raoofy, Michael Gerndt, and Martin Schulz. Sustainability in hpc: Vision and opportunities, 2023.

[26] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 103–116, New York, NY, USA, 2001. Association for Computing Machinery.

[27] Hao Chen, Michael C. Caramanis, and Ayse K. Coskun. The data center as a grid load stabilizer. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 105–112, 2014.

[28] Hao Chen, Yijia Zhang, Michael C. Caramanis, and Ayse K. Coskun. Energyqare: Qos-aware data center participation in smart grid regulation service reserve provision. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 4(1), jan 2019.

[29] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, page 303–314, New York, NY, USA, 2005. Association for Computing Machinery.

[30] Peng Cheng, Yutong Lu, Yunfei Du, and Zhiguang Chen. Tiered data management system: Accelerating data processing on hpc systems. *Future Generation Computer Systems*, 101:894–908, 2019.

[31] Jinkyun Cho and Yundeok Kim. Improving energy efficiency of dedicated cooling system and its contribution towards meeting an energy-optimized data center. *Applied Energy*, 165:967–982, 2016.

[32] Marcin Copik, Marcin Chrapek, Larissa Schmid, Alexandru Calotoiu, and Torsten Hoefler. Software resource disaggregation for hpc with serverless computing, 2024.

[33] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2016.

[34] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, Santa Clara, CA, July 2015. USENIX Association.

[35] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, page 78–88, USA, 2006. IEEE Computer Society.

[36] Ronald P Doyle. Balance of power: Energy management for server clusters.

[37] Mootaz elnozahy, Michael D. Kistler, and RAMAKRISHNAN RAJAMONY. Energy-efficient server clusters. In *Power-Aware Computer Systems - 2nd International Workshop, PACS 2002, Revised Papers*, volume 2325 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 179–197. Springer-Verlag, 2003.

[38] Thomas Erl, Richardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology, & architecture*. Prentice Hall, 2013.

[39] Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, and Ramón Doallo. Performance analysis of hpc applications in the cloud. *Future Generation Computer Systems*, 29(1):218–229, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.

[40] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 13–23, New York, NY, USA, 2007. Association for Computing Machinery.

[41] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29:33–59, 2020.

[42] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, page 293–302, New York, NY, USA, 2005. Association for Computing Machinery.

[43] Mark E Femal and Vincent W Freeh. Safe overprovisioning: Using power limits to increase aggregate throughput. In *Power-Aware Computer Systems: 4th International Workshop, PACS 2004, Portland, OR, USA, December 5, 2004, Revised Selected Papers 4*, pages 150–164. Springer, 2005.

[44] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. Cxl memory as persistent memory for disaggregated hpc: A practical approach. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 983–994, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Mohammad Ghasemazar, Ehsan Pakbaznia, and Massoud Pedram. Minimizing energy consumption of a chip multiprocessor through simultaneous core consolidation and dvfs. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 49–52, 2010.

[46] Akhil Guliani and Michael M. Swift. Per-application power delivery. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Taliver Heath, Bruno Diniz, Enrique V Carrera, Wagner Meira Jr, and Ricardo Bianchini. Self-configuring heterogeneous server clusters. In *Proceedings of the workshop on Compilers and Operating Systems for Low Power*, pages 75–84. Citeseer, 2003.

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

[49] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[50] Nachikethas A. Jagadeesan and Bhaskar Krishnamachari. Software-defined networking paradigms in wireless networks: A survey. *ACM Comput. Surv.*, 47(2), nov 2014.

[51] Yichao Jin, Yonggang Wen, and Qinghua Chen. Energy efficiency and server virtualization in data centers: An empirical investigation. In *2012 Proceedings IEEE INFOCOM Workshops*, pages 133–138, 2012.

[52] Nicola Jones et al. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–166, 2018.

[53] Philo Juang, Qiang Wu, Li-Shiuan Peh, Margaret Martonosi, and Douglas W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, page 127–130, New York, NY, USA, 2005. Association for Computing Machinery.

[54] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[55] Martin Kleppmann. *Designing Data-Intensive Applications: the big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, 2017.

[56] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. GreenCloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012-12-01.

[57] Tejaswini Kolpe, Antonia Zhai, and Sachin S. Sapatnekar. Enabling improved power management in multicore processors through clustered dvfs. In *2011 Design, Automation & Test in Europe*, pages 1–6, 2011.

[58] Markus Kowarschik and Christian Weiß. *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, pages 213–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[59] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[60] Yonghye Kwon. Pytorch-cifar100, June 2024. Available at https://github.com/weiaicunzai/pytorch-cifar100/.

[61] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

[62] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 77–87, 2006.

[63] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[64] Yongpeng Liu and Hong Zhu. A survey of the research on power management techniques for high-performance systems. *Software: Practice and Experience*, 40(11):943–964, 2010.

[65] Ricardo Macedo, Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, Richard Todd Evans, José Pereira, and João Paulo. Taming metadata-intensive hpc jobs through dynamic, application-agnostic qos control. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 47–61, 2023.

[66] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. A survey and classification of software-defined storage systems. *ACM Comput. Surv.*, 53(3), may 2020.

[67] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. PAIO: General, portable I/O optimizations with minor application modifications. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 413–428, Santa Clara, CA, February 2022. USENIX Association.

[68] Enrico Martin. Virtualization and containerization: a new concept for data center management to optimize resources distribution. 2022.

[69] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 205–216, New York, NY, USA, 2009. Association for Computing Machinery.

[70] G. Merlino, D. Bruneo, F. Longo, A. Puliafito, and S. Distefano. Software defined cities: A novel paradigm for smart cities through iot clouds. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 909–916, 2015.

[71] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh, Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf. A case for intra-rack resource disaggregation in hpc. *ACM Trans. Archit. Code Optim.*, 19(2), mar 2022.

[72] Ripal Nathuji and Karsten Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 265–278, New York, NY, USA, 2007. Association for Computing Machinery.

[73] Tirthak Patel and Devesh Tiwari. Perq: Fair and efficient power management of power-constrained large-scale computing systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 171–182, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. Power routing: Dynamic power provisioning in the data center. *SIGARCH Comput. Archit. News*, 38(1):231–242, mar 2010.

[75] Eduardo Pinheiro, Ricardo Bianchini, Enrique V Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. Technical report, Rutgers University, 2001.

[76] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 342–355, New York, NY, USA, 2015. Association for Computing Machinery.

[77] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: Coordinated multi-level power management for the data center.

In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 48–59, New York, NY, USA, 2008. Association for Computing Machinery.

[78] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: Coordinated multi-level power management for the data center. *SIGOPS Oper. Syst. Rev.*, 42(2):48–59, mar 2008.

[79] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, page 66–77, USA, 2006. IEEE Computer Society.

[80] Thomas L Saaty. *Fundamentals of decision making and priority theory with the analytic hierarchy process*. RWS publications, 1994.

[81] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Zhijian Lu. Power-aware qos management in web servers. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 63–72, 2003.

[82] Woong Shin, Christopher D. Brumgard, Bing Xie, Sudharshan S. Vazhkudai, Devarshi Ghoshal, Sarp Oral, and Lavanya Ramakrishnan. Data jockey: Automatic data management for hpc multi-tiered storage systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 511–522, 2019.

[83] Junaid Shuja, Kashif Bilal, Sajjad A. Madani, Mazliza Othman, Rajiv Ranjan, Pavan Balaji, and Samee U. Khan. Survey of techniques and architectures for designing energy-efficient data centers. *IEEE Systems Journal*, 10(2):507–519, 2016.

[84] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[85] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500 list - november 2023, November 2023. Available at https://www.top500.org/lists/top500/list/2023/11/.

[86] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 182–196, New York, NY, USA, 2013. Association for Computing Machinery.

[87] Amir Varasteh and Maziar Goudarzi. Server consolidation techniques in virtualized data centers: A survey. *IEEE Systems Journal*, 11(2):772–783, 2017.

[88] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W Keller. Hierarchical power control for large-scale data centers, 2009.

[89] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. Ship: Scalable hierarchical power control for large-scale data centers. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–100, 2009.

[90] Md. Wasi-Ur-Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. Panda. Mr-advisor: A comprehensive tuning tool for advising hpc users to accelerate mapreduce applications on supercomputers. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 198–205, 2016.

[91] Q. Wu, P. Juang, M. Martonosi, L.-S. Peh, and D.W. Clark. Formal control techniques for power-performance management. *IEEE Micro*, 25(5):52–62, 2005.

[92] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. *SIGARCH Comput. Archit. News*, 44(3):469–480, jun 2016.

[93] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51, 2015.

[94] Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. End-to-end i/o monitoring on leading supercomputers. *ACM Trans. Storage*, 19(1), jan 2023.

[95] Bin Yang, Yanliang Zou, Weiguo Liu, and Wei Xue. An end-to-end and adaptive i/o optimization tool for modern hpc storage systems. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1294–1304, 2022.

[96] Yijia Zhang, Daniel C. Wilson, Ioannis Ch. Paschalidis, and Ayse K. Coskun. A data center demand response policy for real-world workload scenarios in hpc. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 282–287, 2021.

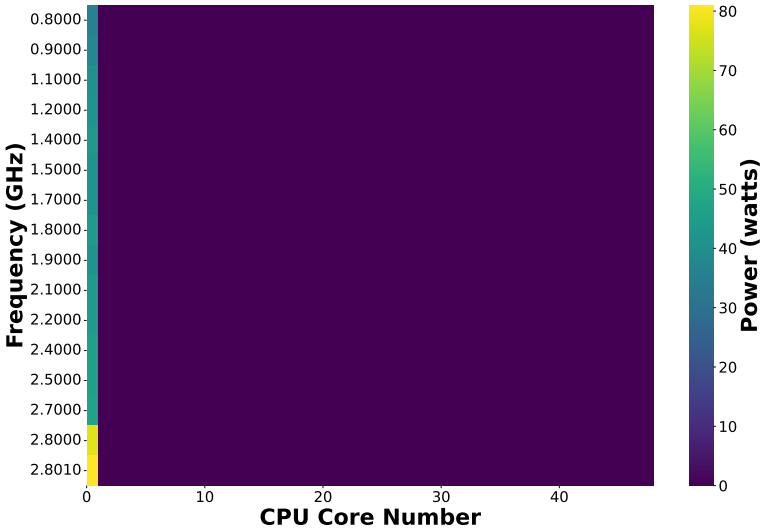
[97] Yin Zhang, Zhiyuan Wei, and Mingshan Zhang. Free cooling technologies for data centers: energy saving mechanism and applications. *Energy Procedia*, 143:410–415, 2017.

[98] Xiaomin Zhu, Chuan He, Kenli Li, and Xiao Qin. Adaptive energy-efficient scheduling for real-time tasks on DVS-enabled heterogeneous clusters. *Journal of Parallel and Distributed Computing*, 72(6):751–763, 2012.

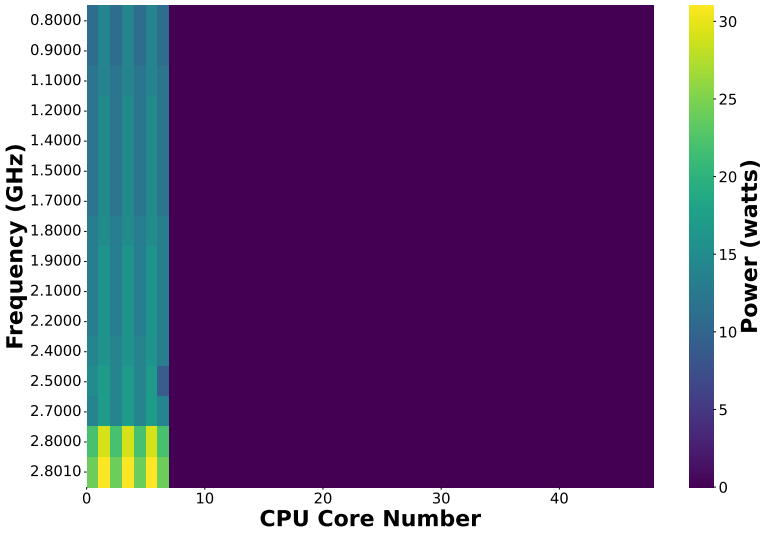
Appendix A

Energy – core frequency analysis

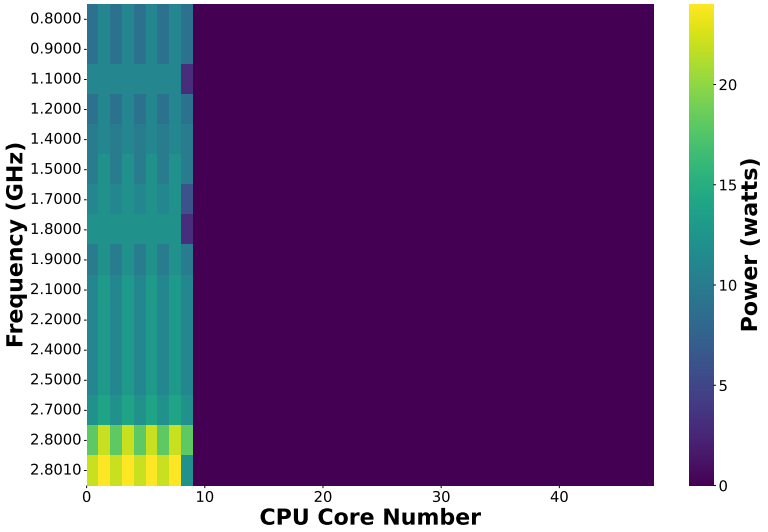
This section presents the energy to core frequency analysis explained in Section 4.2.2. Throughout this experiment, it was found that applying the same frequency to the same number of CPU cores in both NUMA nodes resulted in the same energy consumption values. Thus, these figures correspond to increasing one CPU core in the same NUMA node. The missing results (corresponding to even numbers of CPU cores) would show the same energy values for the other NUMA node. For instance, Figue 4.4b shows the results of using 3 CPU cores in one NUMA node and 2 in the other, resulting in a total of 5 CPU cores running. However, showing a figure with 6 active CPU cores would show the same conclusions as the values (and colors) of the 3 CPU cores running in the same NUMA node. Figures with 3 and 5 CPU cores are represented in Section 4.2.2.



(a) 1 CPU cores.

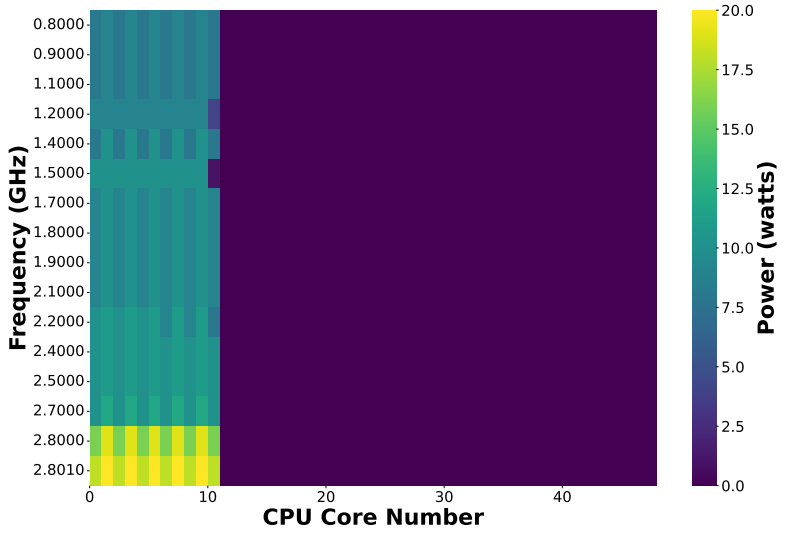


(b) 7 CPU cores.

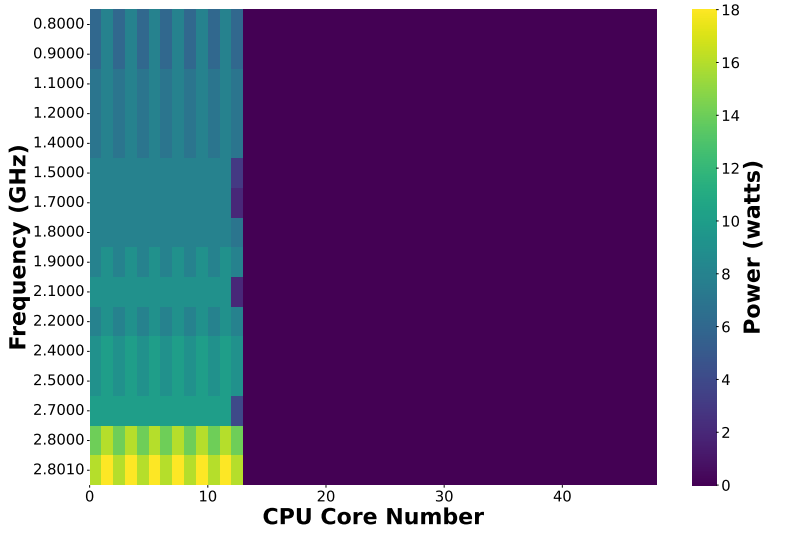


(c) 9 CPU cores.

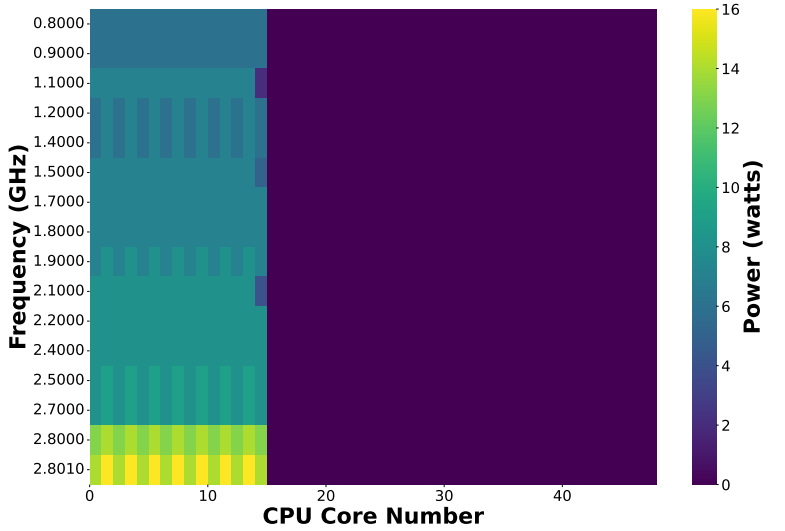
Figure A.1: Experiments measuring energy consumption per set of CPU cores in CN_1 with 1, 7, and 9 cores.



(a) 11 CPU cores.

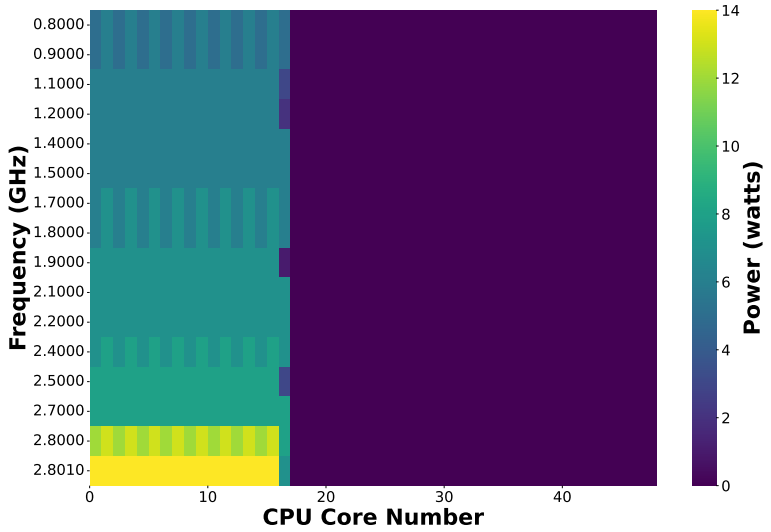


(b) 13 CPU cores.

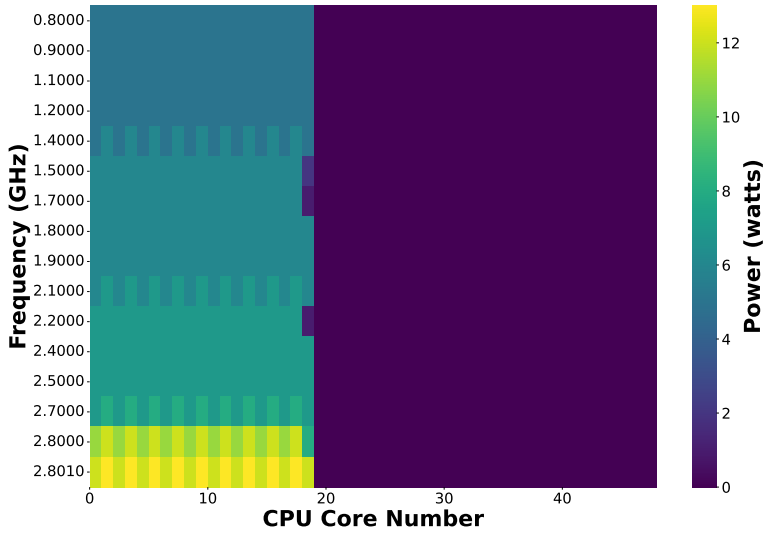


(c) 15 CPU cores.

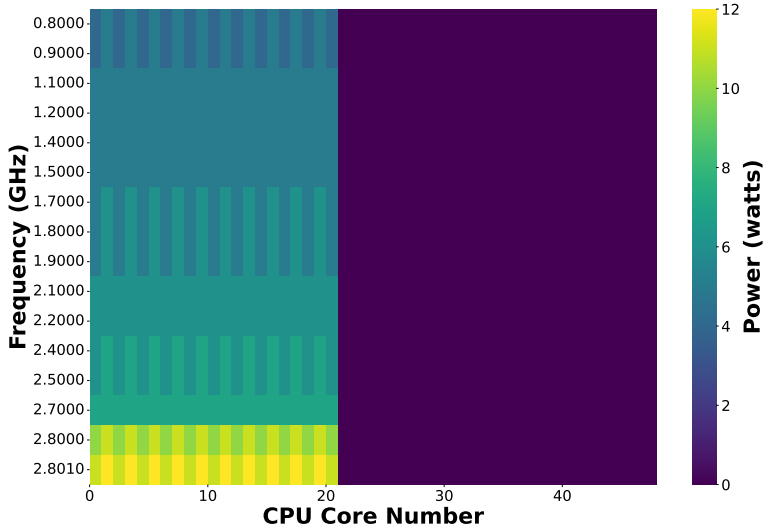
Figure A.2: Experiments measuring energy consumption per set of CPU cores in CN_1 with 11, 13, and 15 cores.



(a) 17 CPU cores.

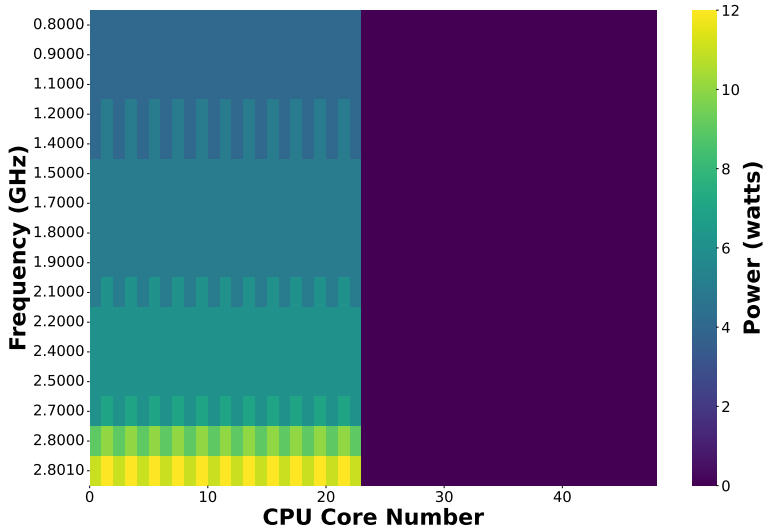


(b) 19 CPU cores.

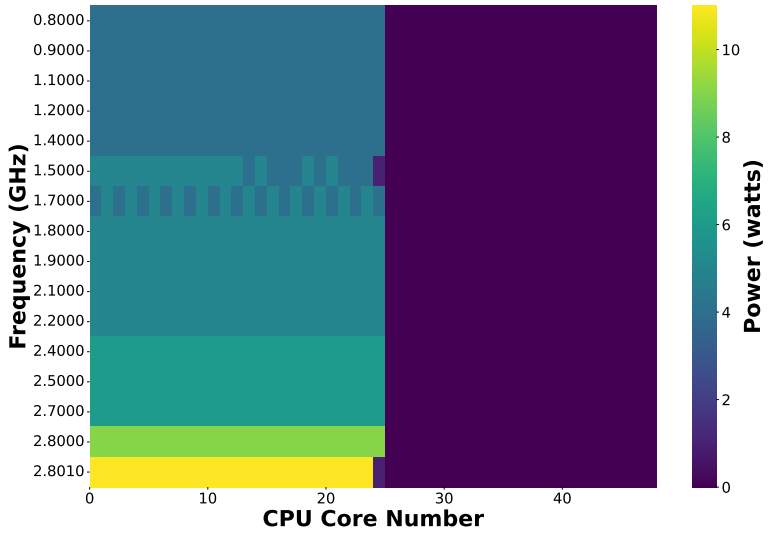


(c) 21 CPU cores.

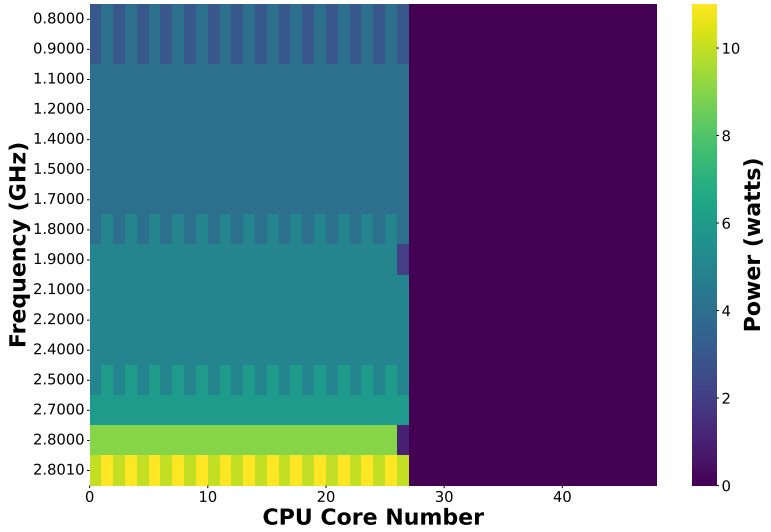
Figure A.3: Experiments measuring energy consumption per set of CPU cores in CN_1 with 17, 19, and 21 cores.



(a) 23 CPU cores.

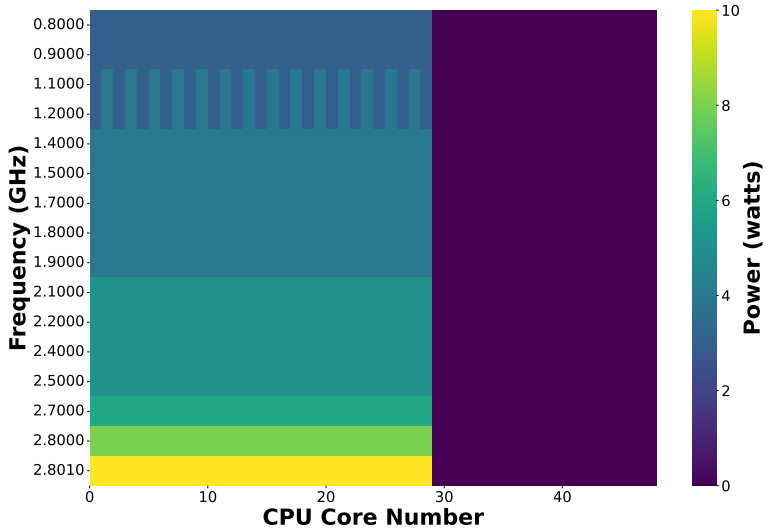


(b) 25 CPU cores.

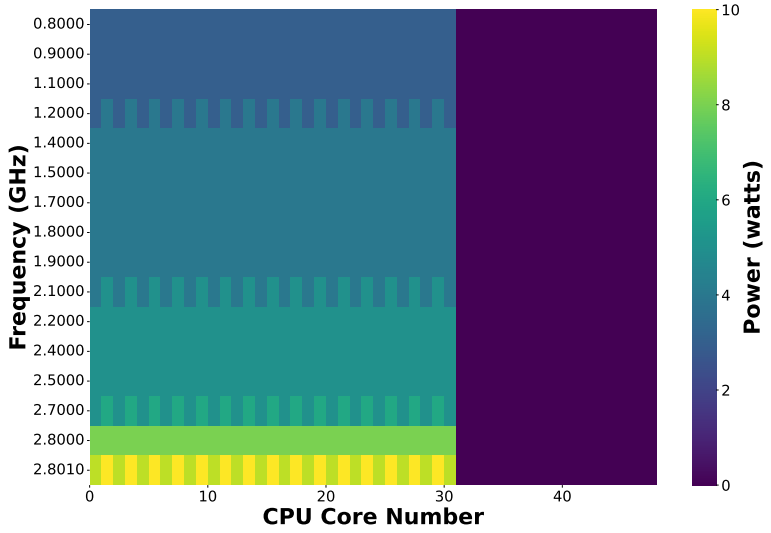


(c) 27 CPU cores.

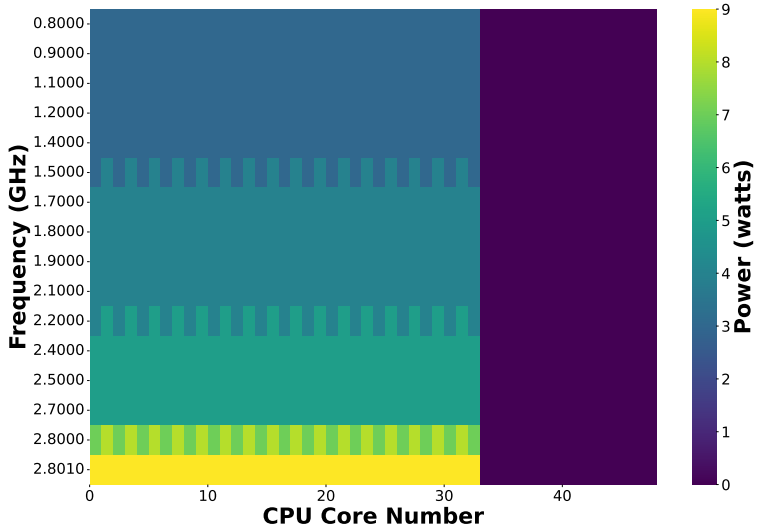
Figure A.4: Experiments measuring energy consumption per set of CPU cores in CN_1 with 23, 25, and 27 cores.



(a) 29 CPU cores.

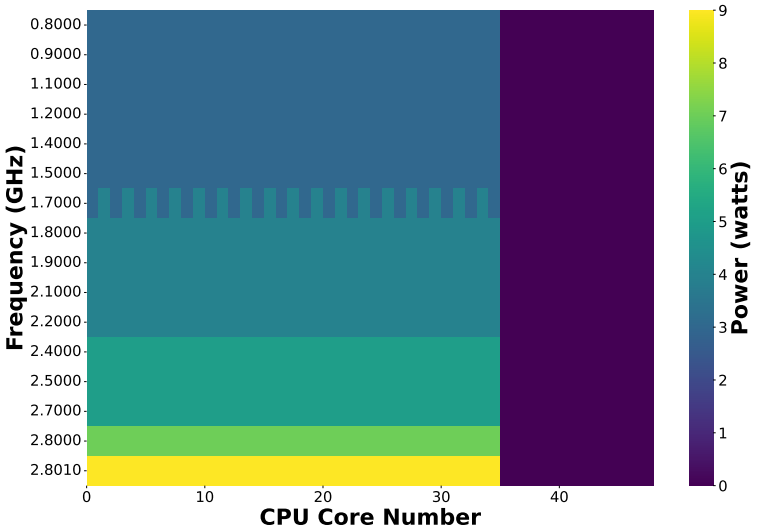


(b) 31 CPU cores.

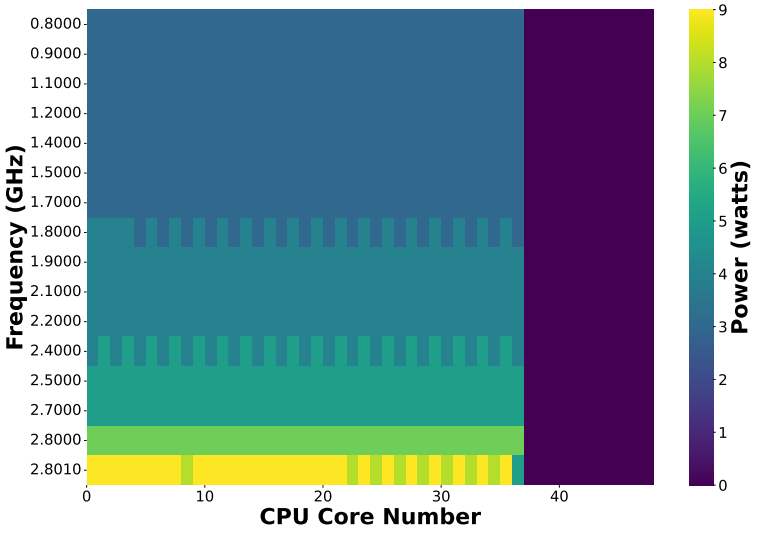


(c) 33 CPU cores.

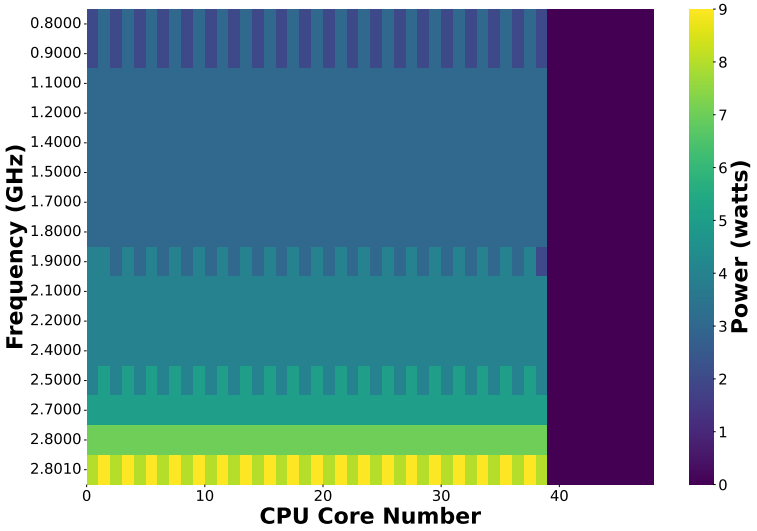
Figure A.5: Experiments measuring energy consumption per set of CPU cores in CN_1 with 29, 31, and 33 cores.



(a) 35 CPU cores.

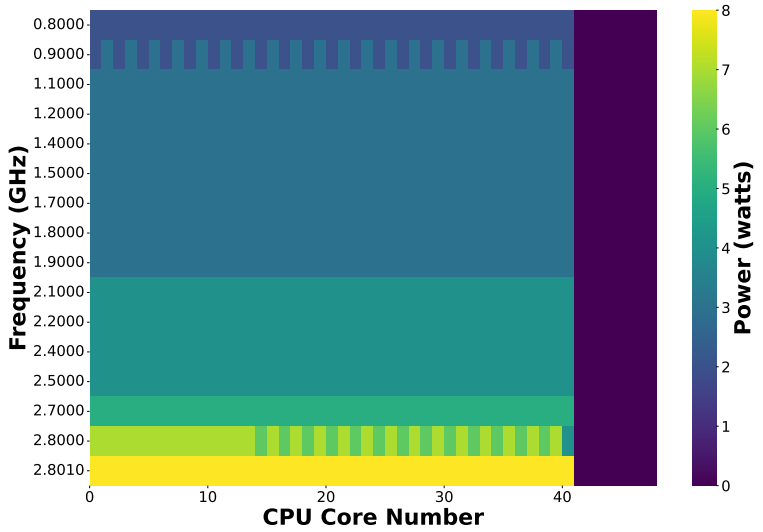


(b) 37 CPU cores.

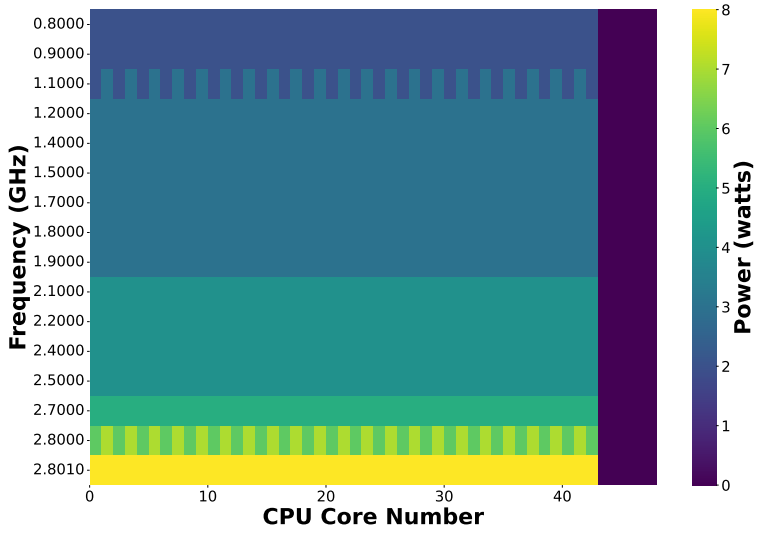


(c) 39 CPU cores.

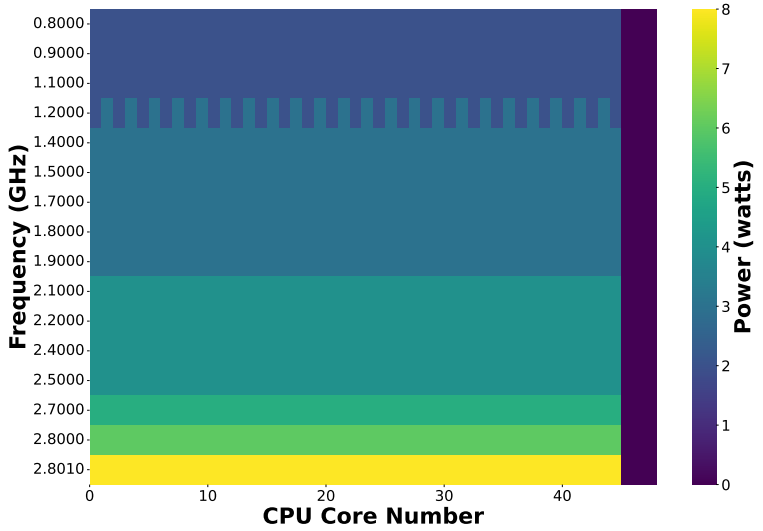
Figure A.6: Experiments measuring energy consumption per set of CPU cores in CN_1 with 35, 37, and 39 cores.



(a) 41 CPU cores.

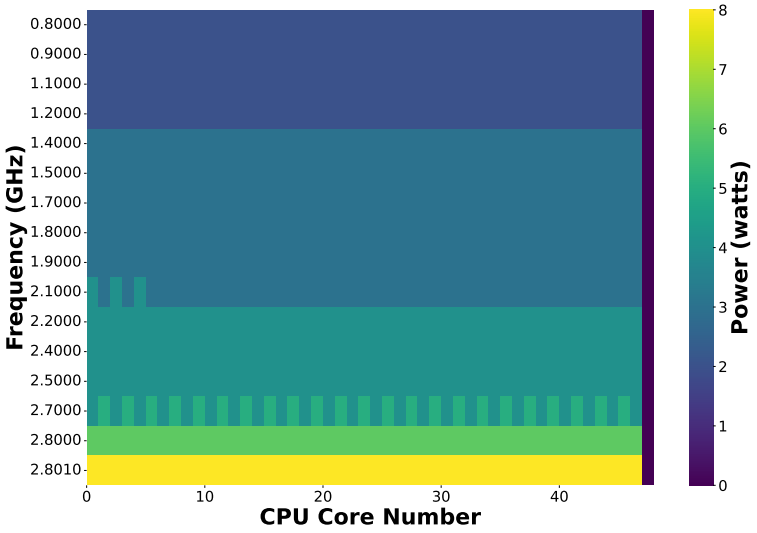


(b) 43 CPU cores.



(c) 45 CPU cores.

Figure A.7: Experiments measuring energy consumption per set of CPU cores in CN_1 with 41, 43, and 45 cores.



(a) 47 CPU cores.

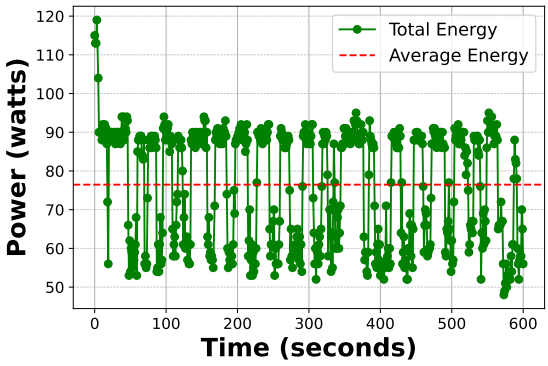
Figure A.8: Experiments measuring energy consumption per set of CPU cores in CN_1 with 47 cores.

Appendix B

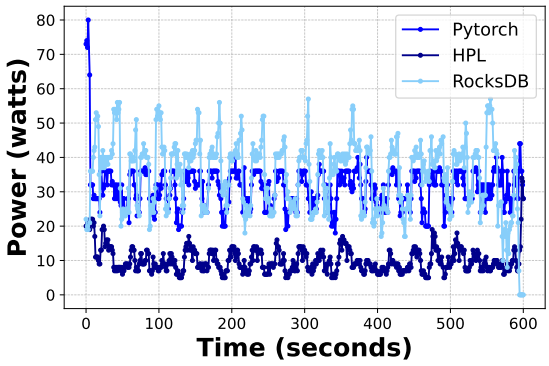
Node controller evaluation

This section presents the additional tests conducted to evaluate the node-level controller, following the methodology explained in Section 4.3.

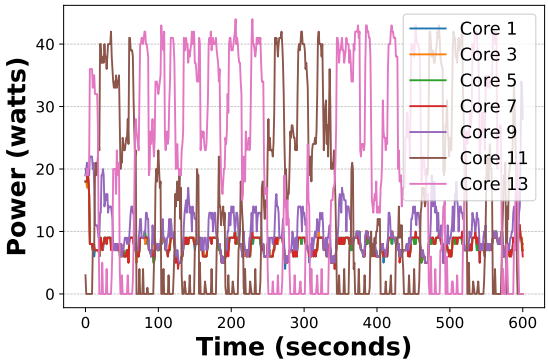
B.1 Priority Control Algorithm



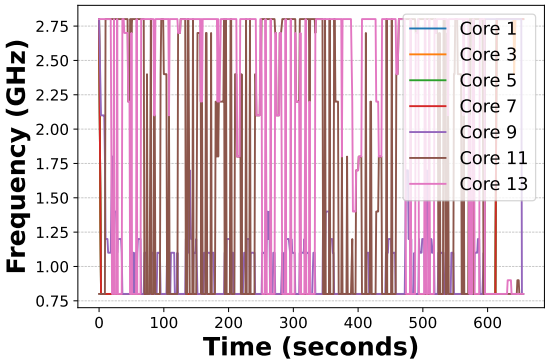
(a) Total energy consumption.



(b) Per application energy consumption.



(c) Per CPU core energy consumption.



(d) Per CPU core applied frequency.

Figure B.1: Experiments for the priority algorithm under the node-level energy control setting with a 75 W target and high priority in *RocksDB*.

When setting the energy consumption limit to 75 W and applying the priority control algorithm with *RocksDB* having high priority, *HPL* having medium priority, and *PyTorch* having low priority in a *node controller*, one obtains the results shown in Figures B.1a, B.1b, and B.1c.

The server's total energy consumption varies, with oscillations mostly between 90 and 55 W, but the average (shown by the red line) is around 76 W, very close to the goal (Figure B.1a). Since the configuration set *RocksDB* as having high priority, it is coherent that this application spent the highest amount of energy (Figure B.1b). Even though *HPL* has higher priority than *PyTorch*, *PyTorch* uses 4 CPU cores while *HPL* uses 1; thus, is it not surprising that *PyTorch* spends more energy than *HPL*, and even be similar to *RocksDB* (uses 2 cores). However, when analyzing the energy consumed by each CPU core (Figure B.1c), one can see that cores 11 and 13 (pink and brown lines - *RocksDB*) spent high values of energy (up to around 40 W), core 9 (purple line - *HPL*) spends between 7 and 20W, and cores 1, 3, 5 and 7 (blue, orange, green and red lines - *PyTorch*) spent low values of energy (mostly below 10 W).

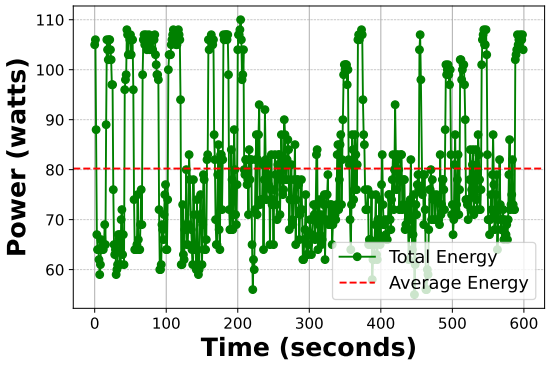
Table B.1 shows each application's average energy efficiency for the current scenario. *RocksDB*, with high priority, achieves 291.445 W/MOps, showing efficient energy usage for its operations under the given constraints. *HPL*, as a medium priority application, exhibits an average energy efficiency of 747.625 W/Eq. This value falls between the one obtained when applying the *performance* and the *powersave and performance* governors. Despite being low priority, *PyTorch*'s energy efficiency of 264.302 W/KIter indicates relatively efficient energy usage.

Table B.1: Application efficiency with 75 W limit and high priority in *RocksDB*.

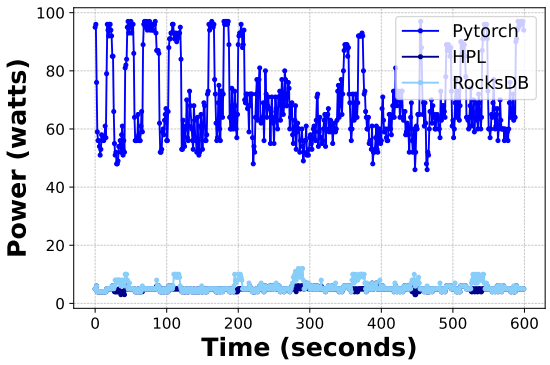
Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	72192046 operations, 119957 ops/sec	291.445 W/MOps
HPL	Time to solve the equations (seconds): 0.05, 65.11, 181.09, 99.59, 0.19, 4.05, 35.82, 77.11.	747.625 W/Eq
PyTorch	Trained until training epoch 2, iteration 20480 of 50000.	264.302 W/KIter

Given the success of this test, it was decided to test the same case with different priorities. Figures B.2a, B.2b, and B.2c show the results of applying the priority control algorithm with *RocksDB* having medium priority, *HPL* having low priority, and *PyTorch* having high priority.

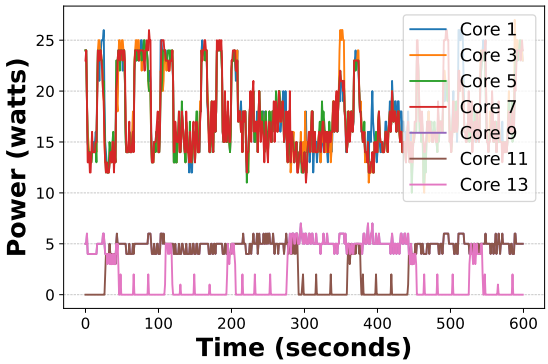
This time, the average total energy consumption was around 80 W, still very close to the 75 W specified in the configurations (Figure B.2a). The results show that contrary to the last test, *PyTorch* is the application that spends most of the total energy, with oscillations between 48 and 97 W (Figure B.2b). It is pretty normal to observe such high oscillations, given the observations stated in Section 4.2.2, in which the energy consumption is not linearly proportional to the applied frequencies. Thus, it is expected that even though the algorithm is increasing the applied frequency to try to achieve a specific energy value, the energy remains more or less the same until the frequency reaches a particular value in which the energy will augment. When that happens, the energy value increases substantially and probably surpasses the energy goal for that CPU core. In that case, the frequency lowers again, and so on.



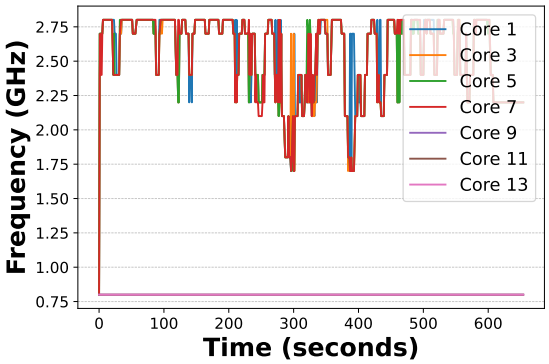
(a) Total energy consumption.



(b) Per application energy consumption.



(c) Per CPU core energy consumption.



(d) Per CPU core applied frequency.

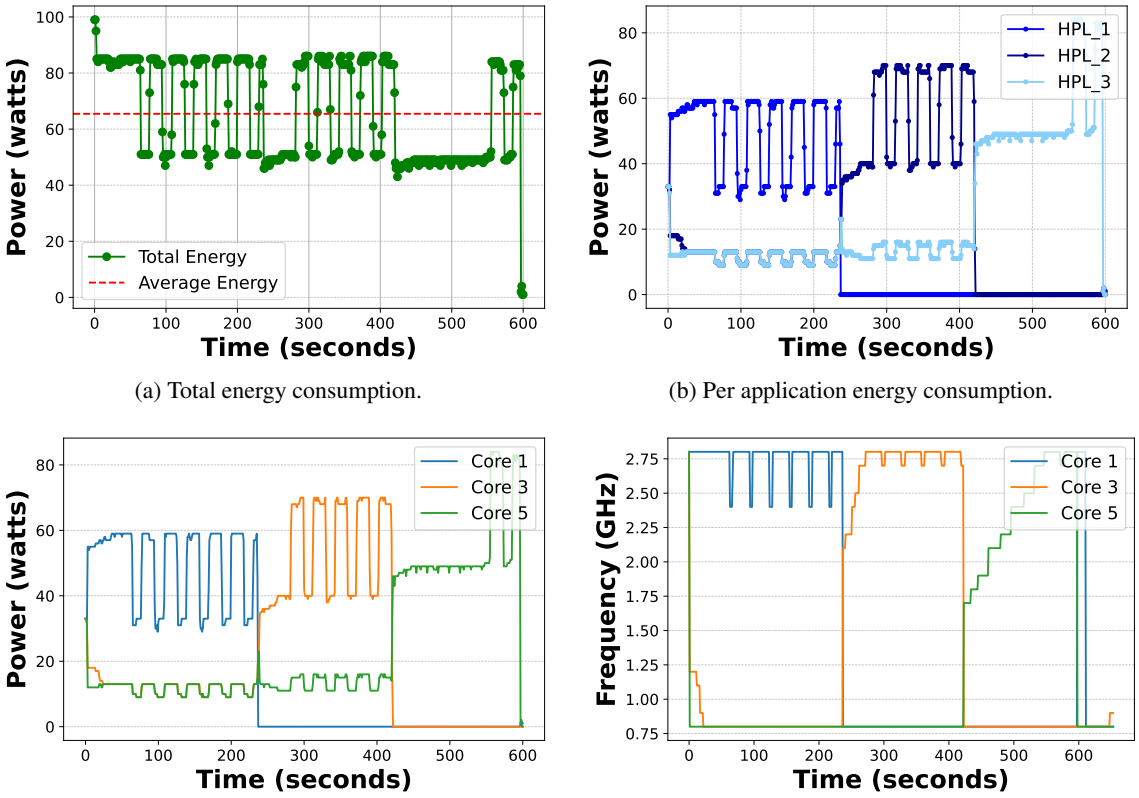
Figure B.2: Experiments for the priority algorithm under the node-level energy control setting with a 75 W target and high priority in *PyTorch*.

Given that *PyTorch* uses 4 cores, when giving it high priority the energy amount distributed to its cores is significant (between 12 and 26 W, as shown in Figure B.2c). Thus, *RocksDB* and *HPL* get very little portions of energy and run at the lowest frequencies, spending around 5 W.

As seen in Table B.2 *RocksDB*, with medium priority, worsens its performance but improves the average energy efficiency (to an average of 128.162 W/MOps). *HPL* running at low priority, does not even finish the calculations, as when applying the *powersave* governor. Lastly, *PyTorch*, as the high priority application, obtains an average efficiency of 277.186 W/KIter, a similar value to the one obtained with the *performance* governors.

Table B.2: Application efficiency with 75 W limit and high priority in *PyTorch*.

Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	26466502 operations, 43651 ops/sec	128.162 W/MOps
HPL	Time to solve the equations (seconds): 0.19, 111.22, 216.52, 111.18, 0.19, 6.06, 47.19, did not finish the last one.	unknown
PyTorch	Trained until training epoch 3, iteration 50000 of 50000.	277.186 W/KIter



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

Figure B.3: Experiments for the priority algorithm under the node-level energy control setting with a 75 W target and running 3 *HPL*s.

Lastly, the aim was to observe the algorithm's response to applications utilizing the same number of CPU cores. Consequently, the priority control algorithm was applied with an energy limit of 75 W and 3 *HPL* applications running, one with high priority, another with medium priority, and the last with low priority.

Having 3 *HPL* dropped the average energy consumption to around 65 W (Figure B.3a). However, this average was influenced by the period between 421 and 555 s, in which the average dropped to around 46 W because 2 *HPL* terminated, and the algorithm is adjusting the frequency of the last running *HPL*, as one can see in Figure B.3d. The energy consumed by each application was as expected, with the priorities being respected (Figure B.3b). The *HPL* with high priority spent higher energy values (between 13 and 59 W), running faster than the others and terminating at 238 s. *HPL* with medium priority spent around 13 W of energy, and when the first *HPL* terminated, *HPL* with medium priority took its place in terms of energy consumption and distribution and started spending between 38 and 70 W. It also terminated at 422 s. Lastly, *HPL* with low priority spent around 13 W all the time, until *HPL* with medium priority terminated. After that period, the last *HPL* spent up to 84 W.

Figure B.3b also corroborates what was stated in Section 4.2.2 because it is clear that the maximum energy spent by the *HPL* running at the highest frequency increases when the number of

used cores diminishes (when the first *HPL* terminates, and then when the second *HPL* terminates). One can take the same conclusions in Figure B.3c. Given that each *HPL* runs in a single CPU core, the energy spent by each application is the same as the energy spent by each core.

Table B.3 shows that HPL_1, as the high priority application, achieves an energy efficiency of 1459.5 W/Eq. The high priority results in substantial energy consumption but efficient processing times. On the other hand, HPL_2, with medium priority, has an energy efficiency of 1586.75 W/Eq. This reflects a balanced energy distribution, though slightly less efficient than HPL_1. Finally, with low priority, HPL_3 exhibits the worse energy efficiency at 1838.375 W/Eq, indicating the highest energy consumption per equation.

Table B.3: Application efficiency with 75 W limit and 3 *HPL*s.

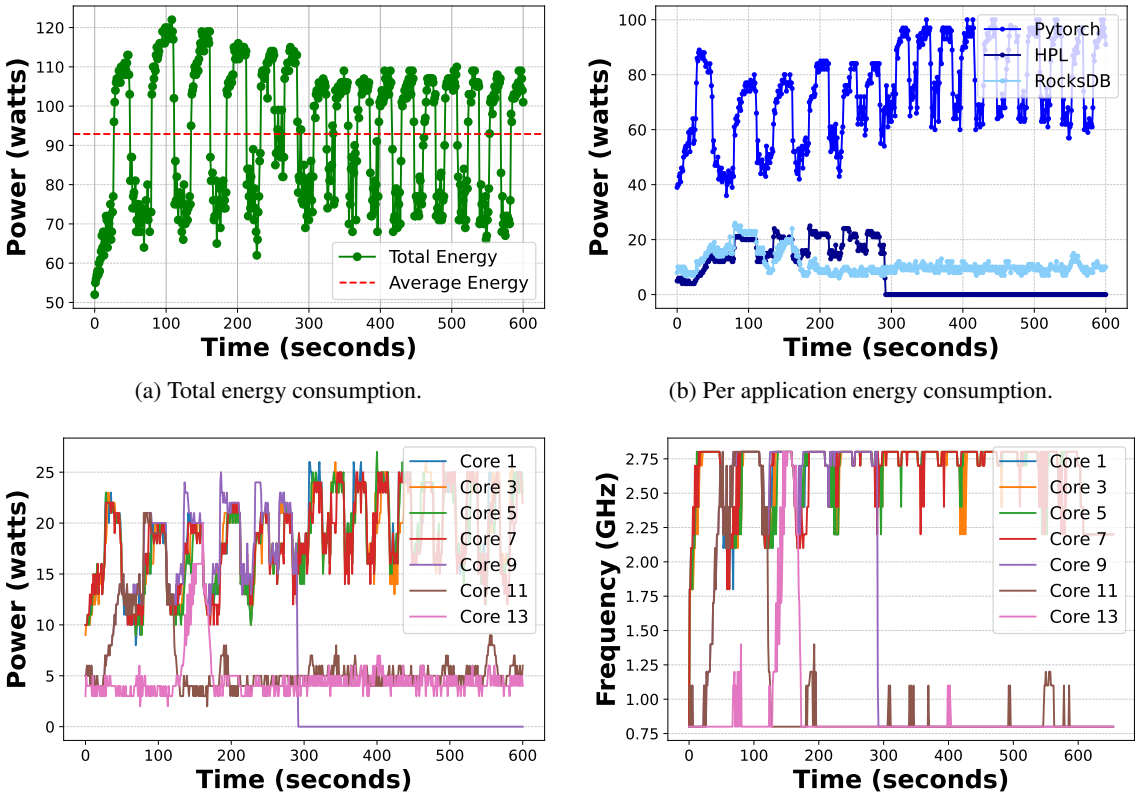
Application	**Application Performance**	**Avg Energy Efficiency**
HPL_1	Time to solve the equations (seconds): 0.06, 25.67, 52.92, 28.30, 0.04, 1.39, 11.10, 55.42	1459.5 W/Eq
HPL_2	Time to solve the equations (seconds): 0.05, 86.46, 135.60, 28.29, 0.04, 1.40, 12.60, 55.16	1586.75 W/Eq
HPL_3	Time to solve the equations (seconds): 0.05, 99.74, 215.26, 46.87, 0.07, 2.23, 16.16, 60.48	1838.375 W/Eq

B.2 Fairness Control Algorithm

To assess the algorithm's behavior with a target near the maximum energy value, the *node controller* was configured with a target of 100 W. The results are presented in Figures B.4a, B.4b, and B.4c. Figure B.4a depicts that the total energy consumption oscillated mostly between around 70 and 120 W, with an average of near 95 W, very close to the goal of 100 W. *PyTorch* got a bigger portion of energy compared to *HPL* and *RocksDB* (Figure B.4b) with variations between around 40 and 100 W, which makes sense as it uses more cores and uses more CPU (as mentioned in Section 4.2.1). *RocksDB* and *HPL* spent similar amounts of energy, up to around 20 W, with *HPL* terminating at 292 s. When analyzing the energy consumption by each CPU core, one can notice that cores running *PyTorch* and *HPL* spent nearly the same energy values until *HPL* finishes. Even for *RocksDB*, it is clear that, in some periods, at least one of its CPU cores also spends nearly the same energy as the rest of the cores. However, given that *RocksDB* also uses the disk (as stated in Section 4.2.1), one of its cores, or even both, is expected to spend less energy most of the time (Figure B.4c).

As Table B.4 depicts, with a 100 W power target, *RocksDB*'s, *HPL*'s and *PyTorch*'s average energy efficiencies were of 213.872 W/MOps, 580.75 W/Eq, and 298.316 W/KIter, respectively. These values were very similar to the ones obtained when applying the *performance* governor, which makes sense given the defined high energy limit.

Lastly, as an example of a scenario with an energy target close to the minimum, the *node controller* was configured with a target energy of 63 W. Also in this test, the total energy consumption



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

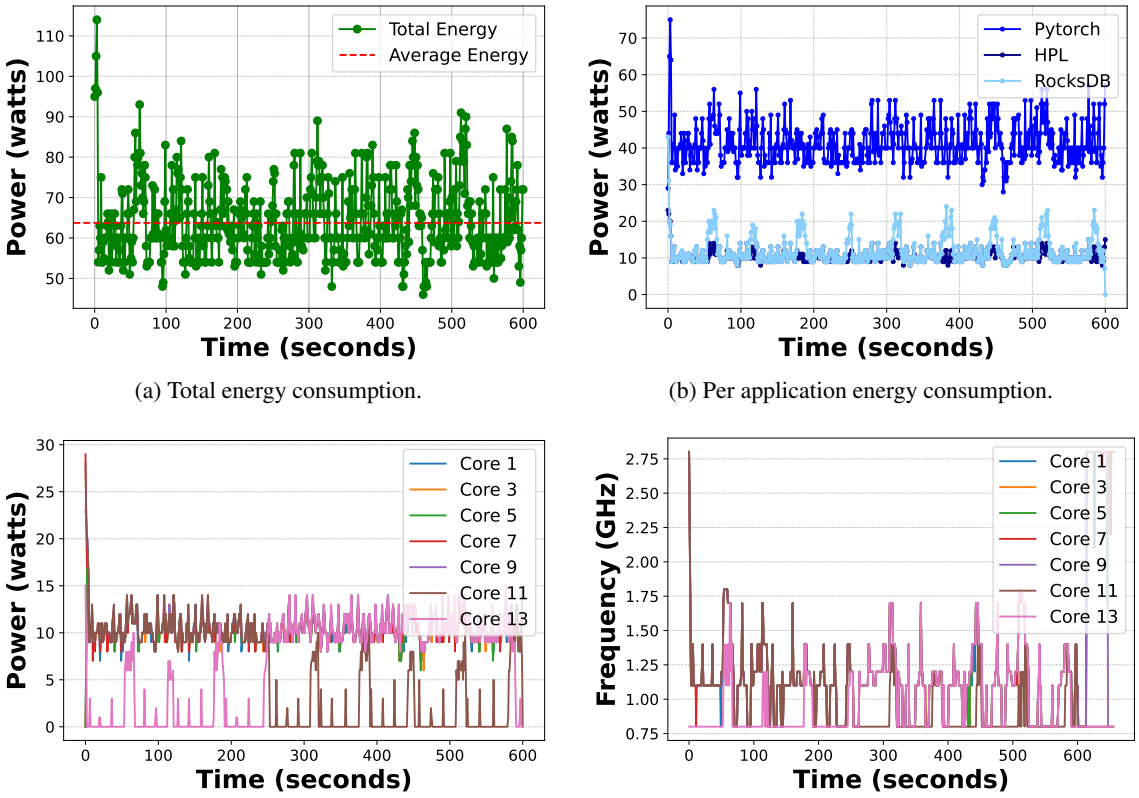
Figure B.4: Experiments for the fairness algorithm under the node-level energy control setting with a 100 W target.

Table B.4: Application efficiency with 100 W limit fairness algorithm.

Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	30953137 operations, 51051 ops/sec	213.872 W/MOps
HPL	Time to solve the equations (seconds): 0.19, 53.12, 57.56, 32.22, 0.05, 1.47, 11.33, 56.03.	580.75 W/Eq
PyTorch	Trained until training epoch 3, iteration 48128 of 50000.	298.316 W/KIter

was successfully limited in a way that reached an average of 64 W (Figure B.5a). Such a strict limit led to a decrease in the energy spent by *PyTorch* to around 40 W and by *HPL* (that did not even finish) to around 10 W, as shown in Figure B.5b. *RocksDB* spent more or less the same as in the last test, with occasional spikes when one of its cores left the *idle* state (Figure B.5c). The energy distribution throughout the CPU cores remained stable, with around 10 W for each core, except for one of *RocksDB*' cores, which had periods in the *idle* state.

Each application's performance and average energy efficiency are represented in Table B.5. *RocksDB*'s, *HPL*'s, and *PyTorch*'s average energy efficiency were of 209.023 W/MOps, 790.875 W/Eq, and 289.651 W/KIter, respectively. The *RocksDB*'s energy efficiency was very similar to the one obtained for the evaluation with the 100 W limit and better than the one obtained for the



(a) Total energy consumption.

(b) Per application energy consumption.

(c) Per CPU core energy consumption.

(d) Per CPU core applied frequency.

Figure B.5: Experiments for the fairness algorithm under the node-level energy control setting with a 63 W target.

evaluation with the 75 W limit. Conversely, the *HPL* average efficiency was the worst obtained for the fairness algorithm evaluation. *PyTorch*'s efficiency was worse than the one that resulted from the 75 W limit but better than the one obtained from the 100 W limit.

Table B.5: Application efficiency with 63 W limit fairness algorithm.

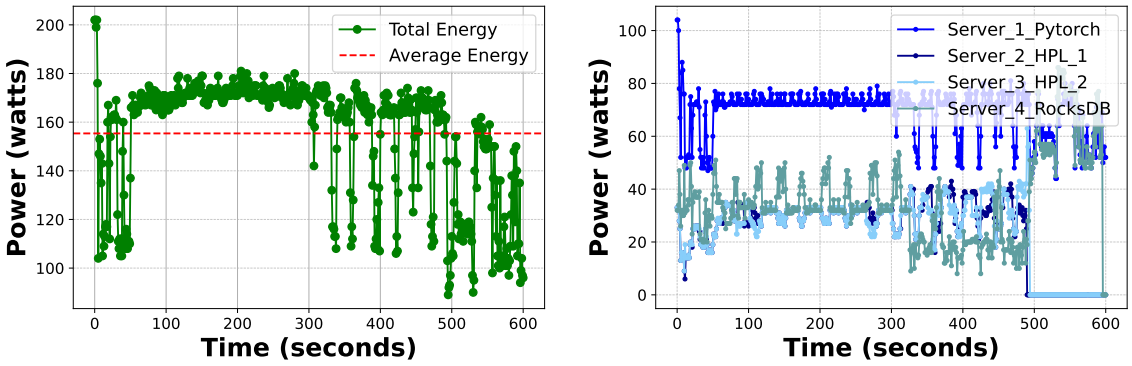
Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	34244991 operations, 56937 ops/sec	209.023 W/MOps
HPL	Time to solve the equations (seconds): 0.05, 74.23, 163.49, 87.78, 0.16, 4.44, 33.88, 126.31.	790.875 W/Eq
PyTorch	Trained until training epoch 2, iteration 34816 of 50000.	289.651 W/KIter

Appendix C

Rack controller evaluation

This section presents the additional tests conducted to evaluate the rack-level controller, following the methodology explained in Section 4.4.

C.1 Priority Control Algorithm



(a) Total energy consumption.

(b) Per server energy consumption.

Figure C.1: Experiments for the priority algorithm under the rack-level energy control setting with a 170 W target and high priority in *RocksDB*.

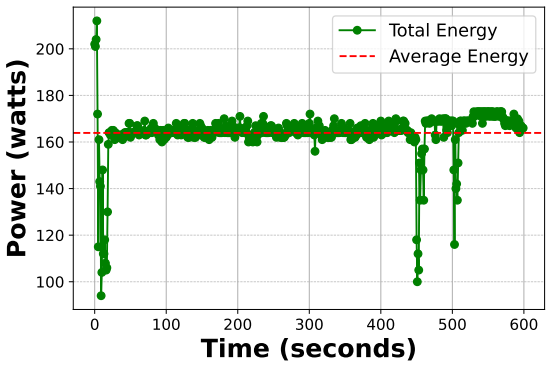
When setting the energy consumption limit to 170 W and applying the priority control algorithm with *RocksDB* having high piority (Server 4), HPL_1 and HPL_2 having medium priority (Server 2 and Server 3), and *PyTorch* having low priority (Server 1), one obtains the results presented in Figures C.1a and C.1b. The total energy consumption average was 158 W. Although this is a low value, around 10 W below the target, one can see in Figure C.1a that during a long period (between 50 and 305 s), the total energy was around 170 W. Besides, the average is highly influenced by the last seconds of the test (after the 500 s), when both *HPL* terminate, as depicted in Figure C.1b, and the total energy drops. The same picture shows that *RocksDB* spends more energy than *HPL*s and that *PyTorch*, even having low priority, spends more energy than the rest of the applications since it runs in more CPU cores, and is using alone a NUMA node. These facts

inhibit *PyTorch* from having the interference of other applications, as mentioned in Section 4.2.2, as it happens with *RocksDB* and the *HPL*s.

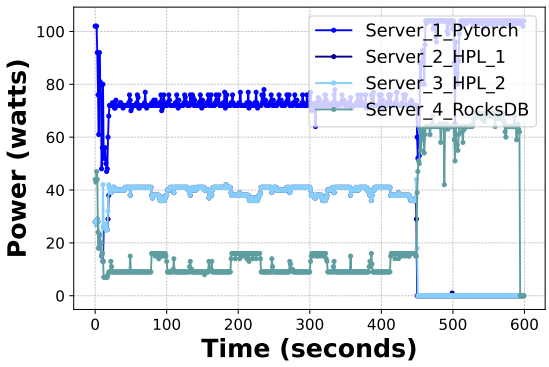
Table C.1 depicts each application's performance and average energy efficiency. *RocksDB* shows a efficiency with 442.554 W/MOps, indicating effective energy utilization given its high priority. The *HPL* applications demonstrate very similar equation completion times and energy efficiency. *PyTorch*, despite having low priority, consumed substantial energy due to its higher core utilization and isolated NUMA node, achieving 698.257 W/KIter.

Table C.1: Application efficiency with 170 W limit and high priority in *RocksDB*.

Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	47750157 operations, 79343 ops/sec	442.554 W/MOps
HPL_1	Time to solve the equations (seconds): 0.05, 39.07, 55.00, 25.72, 0.04, 1.41, 10.95, 49.97, 0.04, 25.71, 54.14, 28.33, 0.06, 1.82, 11.84, 59.03.	914.5 W/Eq
HPL_2	Time to solve the equations (seconds): 0.05, 38.69, 56.33, 25.72, 0.04, 1.41, 10.95, 50.02, 0.04, 28.13, 55.19, 27.76, 0.06, 1.77, 11.8, 55.17.	922.313 W/Eq
PyTorch	Trained until training epoch 2, iteration 9216 of 50000.	698.257 W/KIter



(a) Total energy consumption.



(b) Per server energy consumption.

Figure C.2: Experiments for the priority algorithm under the rack-level energy control setting with a 170 W target and high priority in *HPL*s.

Then, the priorities were defined as *RocksDB* having low piority (Server 4), HPL_1 and HPL_2 having high priority (Server 2 and 3), and *PyTorch* having medium priority (Server 1). Applying these priorities led to an average of 164 W (Figure C.2a), with two drops in the total energy consumption at 450 s (when both *HPL* terminated) and 502 s. This scenario also shows the priorities clearly (as seen in Figure C.2b) since both *HPL* spent much more energy than *RocksDB* (more than double). *PyTorch*, having more CPU cores, medium priority, and running in a separate NUMA node, still spent more energy than each *HPL*, which makes sense. However, if both *HPL* are taken as one single server, the sum of the energy spent by these applications reaches 80 W most of the

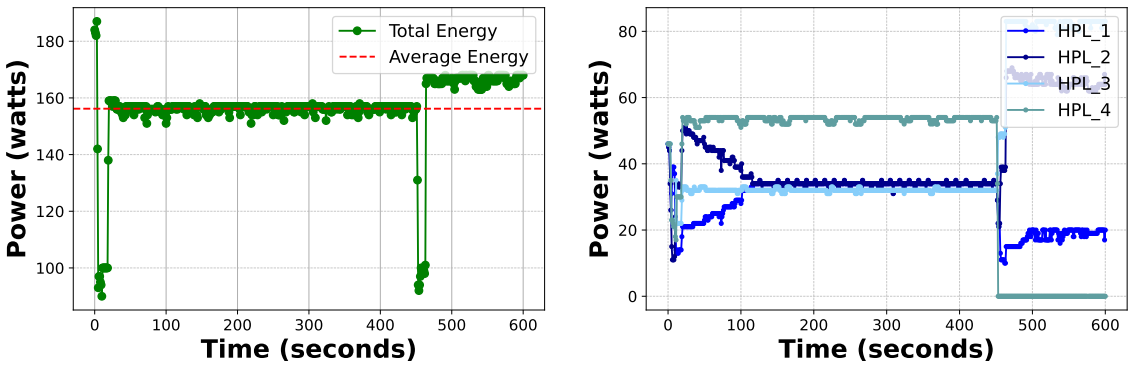
time, higher than *PyTorch*, even though *PyTorch* uses 4 CPU cores and both *HPL* uses a total of 2 cores.

It is also interesting to note how the algorithm adjusts when both *HPL* finish. After 450 s, both *PyTorch* and *RocskDB* start to spend more energy, trying to reach the target energy defined of 170 W. One can see in Figure C.2a that the algorithm was successful in doing so.

Each application's performance and average energy efficiency are shown in Table C.2. *HPL* applications, with high priority, show high energy efficiency values (1091.937 W/Eq and 1096.187 W/Eq), reflecting their intensive energy needs. *RocksDB* and *PyTorch*, with lower priorities, show worse efficiencies compared to the previous evaluation, with 696.112 W/MOps and 612.353 W/KIter, respectively.

Table C.2: Application efficiency with 170 W limit and high priority in *HPL*s.

Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	20523999 operations, 34102 ops/sec	696.112 W/MOps
HPL_1	Time to solve the equations (seconds): 0.05, 32.92, 49.87, 25.66, 0.04, 1.41, 10.93, 49.89, 0.04, 25.66, 49.92, 25.64, 0.04, 1.41, 10.93, 49.87.	1091.937 W/Eq
HPL_2	Time to solve the equations (seconds): 0.05, 32.25, 49.92, 25.68, 0.04, 1.41, 10.93, 49.92, 0.04, 25.68, 49.94, 25.66, 0.04, 1.41, 10.92, 49.89.	1096.187 W/Eq
PyTorch	Trained until training epoch 2, iteration 27648 of 50000.	612.353 W/KIter



(a) Total energy consumption.

(b) Per server energy consumption.

Figure C.3: Experiments for the priority algorithm under the rack-level energy control setting with a 170 W target and running 4 *HPL*s.

Lastly, as in the evaluation of the *node controller*, it was tested a scenario where each server would have the same number of used CPU cores. Therefore, the priority control algorithm was applied with an energy limit of 170 Watts and 4 *HPL* applications running, one with high priority, two with medium priority, and the last with low Priority. These applications were distributed through the NUMA nodes, each NUMA node having 2 *HPL*s, one of them having medium priority.

The total energy consumption was around 157 W most of the time, dropping below 100 W at 452 s when the high priority *HPL* terminated, followed by an augment to around 167 W as shown in Figure C.3a. Figure C.3b illustrates that HPL_4 spent more energy than the rest (around 54 W) and terminated at 452 s, as it was expected given the high priority. Until 452 s, HPL_2 and HPL_3, which had both medium priority, spent more or less the same energy (around 32 W), with HPL_2 spending a little more (around 34 W) since it is running in the same NUMA node as HPL_1 which has low priority and is spending less energy than HPL_4 which has high priority and is running in the same NUMA node as HPL_3 (caused by the interference of the active cores in the same NUMA node as explained in Section 4.2.2). Although HPL_1 has low priority and HPL_2 and HPL_3 have medium priority, given that the number of components having medium priority is larger than the number of components having high and low priority, the algorithm distributes a bigger portion of energy to the high priority component, and the other components get a similar portion. Having two medium priority *HPL*s causes the medium priority portion (despite being larger than the low priority portion) to be divided among more components, each component ending up with a portion similar to the low priority one.

After 452 s, when HPL_4 terminates, both *HPL*s with medium priority increase their energy consumption. HPL_3, which is now running alone in that NUMA node increases to above 80 W (the maximum for a single activated CPU core in a NUMA node, as seen in Section 4.2.2), and HPL_2, which is sharing the NUMA node with HPL_1, increases energy consumption to above 60 W. HPL_1, having low priority, drops its energy to around 19 W.

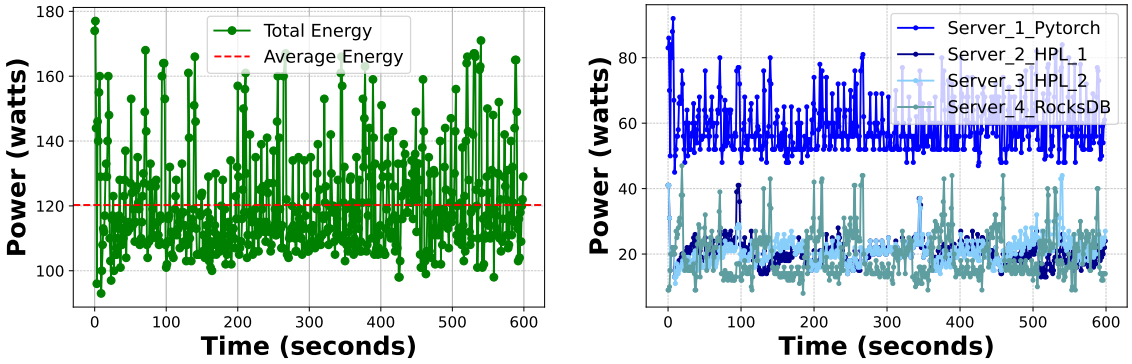
Table C.3: Application efficiency with 170 W limit and 4 *HPL*s.

Application	**Application Performance**	**Avg Energy Efficiency**
HPL_1	Time to solve the equations (seconds): 0.06, 65.77, 214.69, 106.91, 0.16, 4.56, 34.99, did not finish the rest.	unknown
HPL_2	Time to solve the equations (seconds): 0.05, 43.44, 196.52, 110.12, 0.05, 1.78, 10.89, 49.69, 0.04, 25.60, did not finish the rest.	unknown
HPL_3	Time to solve the equations (seconds): 0.05, 40.73, 81.62, 42.01, 0.07, 2.30, 17.75, 81.87, 0.07, 41.97, 61.37, 25.55, 0.04, 1.39, 10.88, 49.67.	1630.875 W/Eq
HPL_4	Time to solve the equations (seconds): 0.05, 32.86, 49.87, 25.66, 0.04, 1.40, 10.90, 49.82, 0.04, 25.63, 49.87, 25.63, 0.04, 1.41, 10.92, 49.82.	1476.063 W/Eq

This evaluation resulted in application performance and average energy efficiency as depicted in Table C.3. The efficiency of *HPL* applications varies, with high priority *HPL* achieving the best average energy efficiency (1476.063 W/Eq). The medium priority *HPL*, HPL_3 running alone in a NUMA node after the high priority *HPL* terminated (HPL_4), achieved the second best performance, with an average energy efficiency of 1630.874 W/Eq. The remaining *HPL*s did not even

finish the calculations, although it is important to note that HPL_1 had worse performance than HPL_2 due to their different priorities.

C.2 Fairness Control Algorithm



(a) Total energy consumption.

(b) Per server energy consumption.

Figure C.4: Experiments for the fairness algorithm under the rack-level energy control setting with a 130 W target.

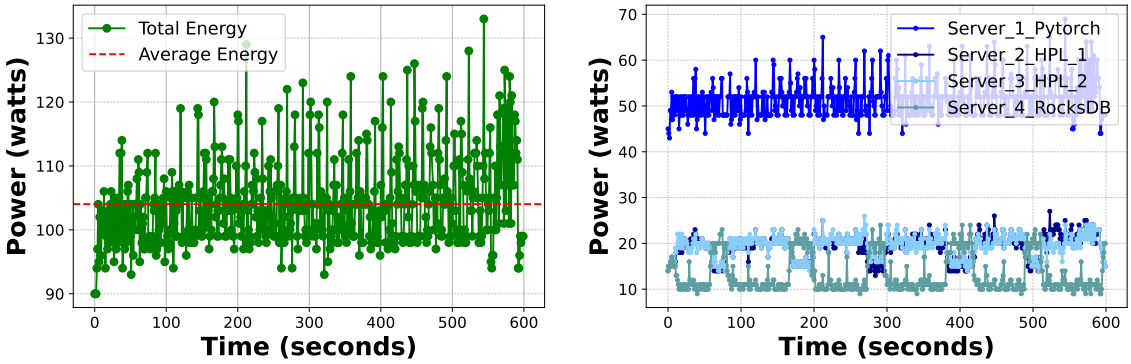
The fairness control algorithm test targeting a midpoint of the energy range involved setting the energy limit at 130 W. This experiment led to higher oscillations in the total energy consumption, with a resulting average of 120 W (Figure C.4a). Compared to the last experiment with a target of 180 W Figure C.4b depicts that an energy target of 130 W led to cuts mainly in the energy used by the *HPL*s and *PyTorch*, which dropped to around 20 and 60 W respectively. *RocksDB* behavior remained more or less the same, with slightly wider variations.

Table C.4: Application efficiency with 130 W limit and fairness algorithm.

Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	34628168 operations, 57108 ops/sec	338.135 W/MOps
HPL_1	Time to solve the equations (seconds): 0.04, 46.50, 73.07, 43.72, 0.07, 2.32, 16.45, 75.11, 0.06, 44.40, 86.10, 45.89, 0.07, 2.24, 15.40, did not finish the rest.	unknown
HPL_2	Time to solve the equations (seconds): 0.04, 51.91, 79.01, 42.53, 0.07, 2.33, 20.31, 71.99, 0.07, 46.33, 81.08, 37.02, 0.06, 2.04, 14.52, did not finish the rest.	unknown
PyTorch	Trained until training epoch 2, iteration 19456 of 50000.	510.697 W/KIter

The average energy efficiencies obtained for *RocksDB* and *PyTorch* applications were 338.135 W/MOps, and 510.697 W/KIter, respectively (as shown in Table C.4). The *HPL*s' time to solve equations increased, with performance issues leading to unknown average energy efficiencies.

RocksDB energy efficiency improved a lot (around 150 W), while *PyTorch*'s worsened a little (around 20 W).



(a) Total energy consumption.

(b) Per server energy consumption.

Figure C.5: Experiments for the fairness algorithm under the rack-level energy control setting with a 100 W target.

Finally, to test the algorithm's behavior with an energy target near the minimum value, a target of 100 W was chosen. The total energy consumption of this test shows narrower oscillations (Figure C.5a) up to a maximum of 133 W, with an average of 104 W, very close to the target. As seen in Figure C.5b, *RocksDB* and the *HPL*s remained restricted at around 20 W, while *PyTorch*'s energy dropped to around 50 W.

For this evaluation, *HPL* applications remained with unknown average energy efficiencies due to not terminating (Table C.5). *RocksDB* presented 382.984 W/MOps, an energy efficiency between the evaluations with the 180 and 130 W limits, and *PyTorch* resulted in 589.552 W/KIter, the worst energy efficiency for the fairness algorithm.

Table C.5: Application efficiency with 100 W limit and fairness algorithm.

Application	**Application Performance**	**Avg Energy Efficiency**
RocksDB	20455186 operations, 33734 ops/sec	382.984 W/MOps
HPL_1	Time to solve the equations (seconds): 0.19, 63.48, 22.16, 66.07, 0.10, 3.15, 24.43, 118.33, 0.10, did not finish the rest.	unknown
HPL_2	Time to solve the equations (seconds): 0.19, 63.28, 120.82, 59.91, 0.10, 3.16, 25.12, 122.64, 0.09, 62.72, did not finish the rest.	unknown
PyTorch	Trained until training epoch 2, iteration 2048 of 50000.	589.552 W/KIter