Performance- and Energy-Aware Architecture for Containerized Cloud–Edge Continuum Systems

by

Zouhir BELLAL

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, MAY 9, 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Pr. Nadjia Kara, thesis supervisor
Department of Software Engineering and Information Technology at École de technologie supérieure

Pr. Abdelouahed Gherbi, co-supervisor
Department of Software Engineering and Information Technology at École de technologie supérieure

Pr. Christine Tremblay, president of the board of examiners
Department of Electrical Engineering at École de technologie supérieure

Pr. Kaiwen Zhang, member of the jury
Department of Software Engineering and Information Technology at École de technologie supérieure

Pr. Marcos Dias de Assuncao, member of the jury
Department of Software Engineering and Information Technology at École de technologie supérieure

Pr. Abdelwahab Hamou-Elhadj, external independent examiner
Computer Software Engineering at Concordia University

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON 30, APRIL, 2026

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

Alhamdulillah, all praise and gratitude are due to Allah. This work would never have been achieved without His blessings, guidance, and grace. I would like to express my sincere gratitude to all those who directly or indirectly supported me and contributed to this work throughout the years of my Ph.D. studies.

First and foremost, I would like to thank my supervisor and thesis director, **Prof. Nadjia Kara**, and my co-supervisors, **Prof. Abdelouahed Gherbi** and **Dr. Laaziz Lahlou**, for their continuous guidance, encouragement, and invaluable support throughout this journey. Their mentorship, insight, and patience were fundamental to the completion of this research. I am deeply grateful for their trust and for the opportunities they provided during my doctoral studies.

My gratitude also goes to the members of the dissertation jury for accepting to evaluate this thesis. I am particularly honored by their presence and grateful for the time and effort they devoted to reviewing this work. I sincerely thank the thesis committee chair and the reviewers for their insightful comments and valuable feedback, which helped improve the quality of this dissertation.

I would also like to express my sincere thanks to all the members of **imaginLab** for their academic support, advice, and stimulating discussions throughout the past years.

This thesis was not only an academic endeavor but also a meaningful personal and professional journey that allowed me to meet remarkable people and work with inspiring teams. In this regard, I would like to express my sincere appreciation to my colleagues at **Ericsson**, particularly the **BCSS Innovation Team**, for their support, collaboration, and encouragement. I am especially grateful to **Tan**, **Tim**, and **Christian** for their exceptional mentorship and for the confidence they placed in me. Their guidance and openness created an environment that continuously motivated me to learn, contribute, and grow. I also thank the entire team for making me feel fully integrated and for involving me in stimulating discussions and innovative projects.

This journey also encouraged me to engage more actively with the broader open-source and sustainability communities. I would like to extend my sincere thanks to the **CNCF community** and the **Green Software Foundation**. In particular, I would like to thank **Ayrat Khayretdinov** for his trust and for giving me the opportunity to share my work with the **CNCF Montréal community**. I am also grateful to **Gosia Fricze** and all the members of the **Green Software Foundation Champions Program** for their support, encouragement, and inspiring discussions on sustainable computing.

Finally, and most importantly, I would like to thank my family, especially my father, **Ali**, and my mother, **Nadjia**. Their unconditional love, continuous encouragement, and constant prayers have been my greatest source of strength throughout this journey. This work would not have been possible without their support.

Architecture sensible à la performance et à l'efficacité énergétique pour les systèmes conteneurisés du continuum cloud-edge

Zouhir BELLAL

RÉSUMÉ

D'ici 2030, la demande en électricité du secteur des technologies de l'information (TI) devrait atteindre environ 3 200 TWh, principalement sous l'effet de la croissance rapide de l'intelligence artificielle (IA) et des services cloud-natifs. Cette évolution place l'efficacité énergétique au cœur des objectifs opérationnels pour les fournisseurs des infrastructures infonuagique. Néanmoins, atteindre une exploitation écoénergétique des environnements cloud-natifs demeure un défi majeur en raison de la mutualisation, de l'hétérogénéité des microservices et de la variabilité dynamique des charges de travail, où chaque service présente des caractéristiques de performance spécifiques ainsi que des contrats de niveau de service ((Service Level Agreement -SLA)) stricts. Afin d'éviter toute dégradation des performances, les fournisseurs d'infrastructures cloud exploitent fréquemment les ressources de calcul à leur niveau de performance maximal, ce qui entraîne d'importantes pertes d'efficacité énergétique.

Les processeurs modernes intègrent des mécanismes avancés de gestion de l'énergie, tels que la mise à l'échelle dynamique de la tension et de la fréquence (Dynamic Voltage and Frequency Scaling, DVFS), l'ajustement de la fréquence de l'uncore — qui regroupe les composants du processeur situés hors des cœurs de calcul, notamment certaines parties de la hiérarchie mémoire et de l'interconnexion — ainsi que le contrôle des états d'inactivité du processeur (C-states). Ces mécanismes permettent d'ajuster dynamiquement le compromis entre performance et consommation énergétique. Malgré leur potentiel, l'exploitation efficace de ces mécanismes dans les environnements cloud-native demeure difficile en raison de la complexité des interactions entre les paramètres matériels du comportement des charges de travail et des exigences de performance des applications hébergées. Les gouverneurs énergétiques existants, tels que intel_pstate et schedutil, s'appuient principalement sur des métriques d'utilisation matérielles et fonctionnent sans tenir compte des exigences de performance au niveau des microservices. Par conséquent, ils ne permettent pas de concilier efficacement efficacité énergétique et respect strict des exigences de performance.

Cette thèse vise à répondre à cette limitation en développant des approches de gestion énergétique sensibles et adaptées aux performances des architectures de microservices cloud-natifs. Les méthodes proposées coordonnent dynamiquement le contrôle de la fréquence des cœurs de l'unité central de traitement (Central processing Unit -CPU), de la fréquence uncore et des états d'inactivité, en fonction des caractéristiques des charges de travail et des contraintes de performance des microservices hébergés. Pour ce faire, cette thèse conçoit et étudie des stratégies de contrôle intelligentes, incluant des approches heuristiques ainsi que des contrôleurs basés sur l'apprentissage par renforcement, capables de s'adapter de manière dynamique à la variabilité des charges, aux interférences en environnement mutualisés et à l'évolution des exigences en matière de performances.

Les approches proposées sont implémentées et évaluées sur une plateforme expérimentale infonuagique native basée sur Kubernetes, en utilisant des charges de travail représentatives de microservices et une télémétrie énergétique très détaillée. Les résultats expérimentaux montrent que les solutions proposées réduisent significativement la consommation énergétique au niveau système, tout en garantissant le respect des exigences de performance strictes dans divers scénarios de déploiement et conditions de charge de travail. Ces résultats confirment l'efficacité d'une gestion de la puissance tenant compte de la charge de travail pour améliorer le rendement énergétique sans nuire aux performances des applications.

Cette thèse contribue ainsi au développement de stratégies novatrices de contrôle, d'implémentations de systèmes et d'analyses expérimentales pour le calcul cloud-native durable. Les solutions proposées offrent des mécanismes pratiques et déployables permettant d'améliorer la proportionnalité énergétique des infrastructures cloud, en participant à l'émergence de systèmes cloud durables, adaptatifs et sensibles aux performances.

Mots-clés: monitorage de la puissance, efficacité énergétique, microservices cloud-natifs, contrôle énergétique sensible à la performance, DVFS et ajustement de la fréquence uncore, apprentissage par renforcement.

Performance- and Energy-Aware Architecture for Containerized Cloud–Edge Continuum Systems

Zouhir BELLAL

ABSTRACT

By 2030, the electricity demand of the Information Technology (IT) sector is projected to reach approximately 3,200 TWh, primarily due to the rapid growth of Artificial Intelligence (AI) and cloud-native services. This rapid growth establishes energy efficiency as a fundamental operational objective for cloud providers. However, achieving energy-efficient operation in cloud-native environments remains challenging due to a multi-tenancy architectural model, heterogeneous microservices, and highly dynamic workloads, where each service exhibits distinct performance characteristics and strict Service Level Agreements (SLAs). To avoid violating performance requirements, cloud providers often operate compute nodes at maximum performance levels, leading to significant energy inefficiencies. Modern processors provide advanced power management mechanisms, including Dynamic Voltage and Frequency Scaling (DVFS), uncore frequency scaling, and CPU idle state (C-state) control, enabling runtime adjustment of the performance–power trade-off. Despite their potential, effectively exploiting these mechanisms in cloud-native environments remains difficult due to the complex relationship between hardware control parameters, workload behavior, and application-level performance. Existing power governors, such as intel_pstate and schedutil, rely primarily on hardware-level utilization metrics and operate without awareness of microservice-level performance requirements. Consequently, they are unable to ensure energy-efficient operation while maintaining strict performance compliance. This thesis addresses this limitation by developing performance-aware power management approaches tailored for cloud-native microservices. The proposed methods coordinate CPU core frequency, uncore frequency, and idle state control based on runtime workload characteristics and performance constraints. To achieve this, the thesis investigates and designs intelligent control strategies, including heuristic-based techniques and reinforcement learning–based controllers, capable of dynamically adapting to workload variability, multi-tenant interference, and evolving microservice performance requirements. The proposed approaches are implemented and evaluated on a real Kubernetes-based cloud-native testbed using representative microservice workloads and fine-grained power telemetry. Experimental results demonstrate that the proposed solutions significantly reduce system-level power consumption while maintaining strict performance requirement compliance across diverse deployment scenarios and workload conditions. These results validate the effectiveness of workload-aware power control in improving energy efficiency without compromising application performance. This thesis contributes novel control strategies, system implementations, and experimental insights for energy-efficient cloud-native computing. The proposed solutions provide practical and deployable mechanisms to improve the energy proportionality of modern cloud infrastructures, supporting the development of sustainable and performance-aware cloud systems.

Keywords: Power monitoring, Power efficiency, Cloud-native microservices, Performance-aware power control, DVFS and uncore frequency scaling, Reinforcement learning

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

ACPI	Advanced Configuration and Power Interface
AI	Artificial Intelligence
BE	Best-Effort
C0	Active C-state (fully powered CPU state)
C6	Deep idle C-state
CNCF	Cloud Native Computing Foundation
CRI	Container Runtime Interface
DRAM	Dynamic Random-Access Memory
DTCTP	Discrete Time/Cost Tradeoff Problem
DVFS	Dynamic Voltage and Frequency Scaling
ECM	Edge Computing-based Microservices
GHG	Greenhouse Gas
GOFS	Global Optimal Frequency Selection
HL-PPO	Hierarchical Proximal Policy Optimization
HRL	Hierarchical Reinforcement Learning
HT	Hyper-Threading
IO	Input/Output
IPC	Instructions Per Cycle
IPS	Instructions Per Second
IRQ	Interrupt Request
K8s	Kubernetes
KEDA	Kubernetes Event-Driven Autoscaler
KPI	Key Performance Indicator
LLC	Last-Level Cache

MDP	Markov Decision Process
P95	95th percentile
PKG	Package (entire CPU socket energy domain)
PP0	Power Plane 0 (core energy domain)
PP1	Power Plane 1 (integrated GPU energy domain)
PPO	Proximal Policy Optimization
RAPL	Running Average Power Limit
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RPS	Requests Per Second
SPEC	Standard Performance Evaluation Corporation
TDP	Thermal Design Power
UFS	Uncore Frequency Scaling
USI	Uncore Stressor Instances
VM	Virtual Machine

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

Hz	Hertz
GHz	Gigahertz
s	Second
V	Volt
W	Watt
J	Joule
σ	Standard deviation

INTRODUCTION

0.1 Motivation and Challenges

The accelerating digitalization of society, together with the large-scale deployment of artificial intelligence services, has made the energy consumption of computing infrastructures a major scientific, economic, and environmental concern. Recent projections indicate that global data-center electricity consumption is on track to more than double by 2030, reaching approximately 945 TWh, or just under 3% of total global electricity demand, with growth of roughly 15% per year from 2024 to 2030 (i scoop). This surge is largely driven by the rapid expansion of AI workloads, large language models (de Kruijff, 2025), real-time analytics (Saenko, a), and always-on digital services (Saenko, b). In this context, improving the energy efficiency of cloud platforms is no longer merely desirable; it has become a necessity, especially under emerging regulatory pressures (Ahuja, 2024), for the long-term sustainability of modern digital systems.

This challenge is particularly acute across the cloud–edge continuum. On the one hand, edge computing has emerged to support latency-sensitive and geographically distributed services by bringing computation closer to end users and data sources. On the other hand, centralized cloud platforms remain the backbone of large-scale service hosting. Across this continuum, applications are increasingly deployed using cloud-native paradigms, especially containerized microservices managed by orchestration platforms such as Kubernetes (Ermolenko, Kilicheva, Muthanna & Khakimov, 2021). These deployment models provide flexibility and scalability, but they also introduce new challenges for power management because applications execute under diverse workload patterns, strict performance requirements, and highly dynamic runtime conditions.

In such environments, multiple microservices often share the same physical compute node while exhibiting different workload characteristics. Some are primarily CPU-bound, others are memory-bound, and many display mixed and time-varying behavior. These services frequently operate under explicit performance requirements, such as latency and throughput constraints, that must be maintained despite dynamic load fluctuations and co-location effects. In multi-tenant execution settings, the interaction among co-hosted microservices through shared hardware resources can substantially affect both performance and energy consumption. Consequently, reducing power consumption without violating application-level performance requirements becomes a challenging systems problem that spans workload behavior, runtime orchestration, and processor architecture.

A wide range of approaches has been investigated to improve performance-aware energy efficiency in cloud platforms. These include workload consolidation (Mei, Wang, Chu, Liu, Leung & Li, 2021), resource allocation

and placement optimization (Zhang, Wen & Wu, 2013), as well as autoscaling (Ma, Huang, Zhou, Zhang & Chen, 2022) and load balancing in edge-cloud environments (Masip-Bruin, Marín-Tordera, Tashakor, Jukan & Ren, 2016). However, comparatively less attention has been devoted to exploiting the full capabilities of modern processor power-management features in a manner that is explicitly aware of application-level performance requirements. Yet these hardware capabilities provide direct control over the performance–power trade-off and therefore represent an important opportunity for more effective runtime optimization.

Modern processors expose several hardware mechanisms that can potentially improve energy efficiency. Dynamic Voltage and Frequency Scaling (DVFS) enables processors to adjust operating frequency and voltage to trade performance for power (Weiser, Welch, Demers & Shenker, 1996). More recent platforms also expose controls for the uncore subsystem, including shared components such as the last-level cache, memory controllers, and interconnects, thereby extending the performance–power trade-off beyond core computation alone (Gholkar, Mueller & Rountree, 2019a). In addition, processors support multiple idle power states that reduce energy consumption during inactive periods. Despite the availability of these mechanisms, current operating systems and vendor-provided governors remain predominantly utilization-driven. Policies such as `intel_pstate` and `Linux schedutil` generally rely on generic hardware-level indicators, most notably CPU utilization, to determine runtime frequency adjustments (ArchWiki contributors, 2026). While such heuristics are appropriate for generic throughput-oriented operation, they do not explicitly account for application-level performance objectives, such as service latency. This blind power management, regardless of application performance requirements, may result in performance violations, which is unacceptable, especially for latency-sensitive services. As a result, cloud providers are often pushed to adopt conservative configurations, such as running servers in high-performance mode with maximum core and uncore frequencies, to avoid performance degradation. The consequence is systematic power over-provisioning and poor energy proportionality.

However, achieving energy-efficient and performance-aware operation in such environments is not straightforward, because the problem spans multiple dimensions simultaneously. First, the underlying hardware platforms are heterogeneous. Edge nodes are often resource-constrained and may expose limited hardware control capabilities, whereas cloud servers provide richer processor control features but operate under more complex multi-tenant conditions. Second, applications themselves are heterogeneous: some workloads are CPU-bound, others are memory-bound, and many exhibit mixed behavior (Etinski, Corbalán, Labarta & Valero, 2012). Moreover, co-hosted microservices may have different performance requirements and unpredictable runtime load patterns. Even when workloads become idle from an application perspective, such as web services receiving no HTTP

requests, the underlying platform may still consume non-negligible power, which creates additional opportunities and challenges for idle-state power management. Third, the absence of a dedicated power-observability framework capable of enabling accurate power monitoring under realistic conditions makes it difficult to deeply analyze the power–performance trade-off, understand workload behavior, and support reliable decision-making strategies. Consequently, performance-aware power management across the cloud–edge continuum cannot be addressed through a single simplified model; rather, it must be studied progressively across deployment contexts, hardware constraints, workload characteristics, and execution states.

A first challenge arises from hardware constraints, especially at the edge. Many edge platforms, such as those from Odroid (ODR) and NVIDIA (NVI), expose simplified power-control interfaces. These often include chip-wide DVFS, where a single frequency setting applies to the entire processor package (Kim, Gupta, Wei & Brooks, 2008). Under such constraints, multiple co-hosted services with different workload characteristics and performance requirements must share the same hardware-level decision. This makes power management intrinsically difficult, since reducing frequency may benefit some services while degrading others. The challenge becomes even broader in edge environments when moving from a single node to edge clusters hosting distributed and serverless workloads, where resource allocation and power management must also account for function execution dynamics and node-level heterogeneity. As a result, even at the edge, performance-aware power control must reason jointly about application heterogeneity, hardware limitations, deployment conditions, and shared execution constraints.

A fundamental challenge concerns power observability. Any meaningful power-optimization strategy requires trustworthy information about the relation between workload behavior, hardware configuration, and power consumption. In cloud-native environments, however, obtaining such visibility is difficult. Container orchestration platforms introduce abstraction layers that obscure the mapping between software entities and physical hardware resources, making accurate power attribution to individual containers or microservices an open problem (Bellal, Lahlou, Kara, Murphy & Nguyen, 2025c). Although several container-level power-monitoring tools have been proposed, such as Scaphandre (Hubblo), SmartWatts (Fieni, Rouvoy & Seinturier, 2020), and Kepler (**?**), their practical accuracy under dynamic runtime conditions remains questionable. This limitation is critical because fine-grained observability is needed not only to study power–performance behavior, but also to support decision-making systems that aim to optimize power consumption in containerized cloud-native environments.

For this reason, accurate power observability constitutes a necessary foundation for performance-aware optimization in the cloud. Without reliable power measurements, it becomes difficult to characterize the true effect of hardware control knobs, construct trustworthy datasets, and derive meaningful models of workload behavior. Moreover, if

existing monitoring tools are not accurate enough to directly support runtime decision making, then alternative proxies for power consumption must be identified and exploited. This observation motivates an important part of this thesis: before designing advanced cloud-side control strategies, it is necessary to determine what can be measured reliably and what must instead be inferred from proxy signals.

Beyond observability, effective runtime control in cloud environments requires methods that can adapt to workload heterogeneity, shared-resource contention, and dynamic per-service load variation. In multi-tenant cloud servers, the performance of co-hosted services can be significantly affected by interference on shared resources (Ournani, Belgaid, Rouvoy, Rust, Penhoat & Seinturier, 2020), particularly uncore resources such as the last-level cache, memory bandwidth, and interconnects. This issue is especially important because performance-critical services may run on dedicated CPU cores, while uncore resources remain physically shared and cannot be exclusively reserved in the same way. Consequently, the performance sensitivity of a workload depends not only on its own computational nature and runtime load, but also on the behavior of neighboring workloads. These interactions make simple heuristic or purely utilization-driven power governors inadequate for fine-grained, performance-aware control.

Given the complexity and non-stationarity of cloud workloads, static or purely heuristic policies are often insufficient. This thesis therefore, investigates advanced control strategies, including reinforcement-learning-based approaches, that exploit fine-grained telemetry from hardware counters, container-level metrics, and application-performance signals. These controllers are designed to adapt runtime power settings so as to satisfy performance constraints while minimizing system-level power consumption under dynamic and interference-prone operating conditions. More importantly, they are designed for realistic cloud scenarios in which multiple microservices with different workload characteristics, distinct performance requirements, and dynamic load patterns coexist and interact through shared uncore resources.

This thesis addresses these challenges through the study of performance-aware power management for containerized microservices in cloud-native environments. Its central objective is to develop practical and deployable methods that improve energy efficiency while preserving application-level performance guarantees. To achieve this objective, the thesis combines system-level experimentation, power-observability methodology, and runtime control design. It addresses the problem progressively across the cloud–edge continuum, from resource-constrained edge platforms to cloud-scale multi-tenant servers. All proposed methods are implemented and evaluated on Kubernetes-based experimental testbeds designed to reproduce representative deployment conditions with realistic microservice workloads.

The results demonstrate that substantial improvements in energy proportionality can be achieved in containerized cloud environments when processor power-management capabilities are controlled in a manner explicitly informed by workload behavior and application-level performance objectives. This is evidenced, in cloud-native settings, by node-level power reductions of 23–30% while maintaining performance-requirement violations below 2–3%. The same principle extends to the edge, where workload-aware DVFS-based control reduces microservice energy consumption by 5–23% while preserving latency guarantees. More broadly, this thesis contributes practical methods, experimental frameworks, and system-design insights to support the development of energy-efficient cloud-native platforms amid rapidly growing digital energy demand.

0.2 Problem Statement

Despite the availability of modern hardware power-management mechanisms such as DVFS and uncore frequency scaling, cloud-native platforms still lack practical methods for exploiting these capabilities to reduce total power consumption in a manner that is explicitly aware of application-level performance requirements. This challenge is particularly difficult in large-scale cloud-native environments, where multiple abstraction layers, including virtualization, containerization, and orchestration, separate applications from the underlying hardware resources. In addition, hardware control knobs are often exposed at coarse granularity, such as per-core or per-processor scope, which further complicates fine-grained performance-aware control.

Moreover, existing runtime power governors provided by operating systems or hardware vendors remain largely utilization-driven and are therefore blind to application-level requirements, especially the strict latency constraints of performance-critical microservices. Similarly, many solutions proposed in the literature are evaluated under limited or simplified settings and do not adequately reflect the complexity of real cloud environments, where multiple factors can influence the power–performance trade-off. At the same time, the lack of accurate and trustworthy power observability in containerized environments limits the ability to characterize workload behavior, analyze the power–performance trade-off, and design reliable optimization strategies.

Accordingly, the central problem addressed in this thesis is the following: *How to design practical and deployable performance-aware power-management mechanisms for containerized applications across the cloud–edge continuum that reduce total system power consumption while preserving application-level performance requirements under realistic multi-tenant conditions, heterogeneous workloads, dynamic application runtime loads, and shared-resource interference.*

0.3 Objectives and Contributions

0.3.1 Objectives

The main objective of this thesis is to develop practical and deployable methods for performance-aware and energy-efficient power management in cloud-native environments across the cloud–edge continuum. More specifically, the goal is to minimize system power consumption while preserving, and when necessary guaranteeing, the performance requirements of deployed microservices under realistic execution conditions.

To achieve this main objective, this thesis pursues the following specific objectives:

- investigate how processor power-control mechanisms can be exploited effectively under hardware constraints, especially at the edge, where simplified interfaces such as chip-wide DVFS impose shared decisions across co-hosted services;
- establish a systematic methodology for evaluating power observability in containerized cloud-native environments, with the goal of determining whether existing monitoring tools can provide sufficiently accurate information for analysis and runtime decision making;
- design runtime control strategies capable of adapting to workload heterogeneity, dynamic load variation, and shared-resource interference in order to jointly improve energy efficiency and preserve application-level performance in cloud environments.

0.3.2 Contributions

This thesis makes the following main contributions:

- It investigates workload-aware power-control strategies for resource-constrained edge platforms, with particular emphasis on hardware limitations such as chip-wide DVFS and the difficulty of making shared frequency decisions for co-hosted services with different performance requirements (C, E).
- It extends this edge-side investigation to broader and more heterogeneous edge-cluster settings, including serverless computing environments, where power management and resource allocation must account for distributed execution conditions, node heterogeneity, and finer-grained hardware control capabilities such as per-core DVFS (E).

- It investigates the exploitation of processor idle-state hardware features as a complementary mechanism for dynamic resource management under idle or underutilized workloads, thereby expanding performance-aware power management beyond active execution phases (D).

- It establishes the first systematic methodology for studying power observability in cloud-native environments and analyzes the practical limitations of container-level power-monitoring tools under realistic runtime conditions. We show that accurate power observability is a necessary prerequisite for meaningful performance-aware optimization in the cloud, and this motivates the use of reliable proxy signals when direct container-level power measurements are not sufficiently accurate (B).

- It develops advanced runtime control strategies based on reinforcement learning that exploit fine-grained telemetry from hardware and application-level signals to adjust core and uncore frequency settings under dynamic and interference-prone real-world cloud scenarios (A).

- It implements and evaluates the proposed methods on Kubernetes-based experimental testbeds that reproduce representative cloud and edge deployment conditions with realistic microservice workloads (A, B, D, E).

- It demonstrates that substantial improvements in energy proportionality can be achieved when processor power-management capabilities are controlled in a manner explicitly informed by workload behavior and application-level performance objectives (A, B, C, D, E).

0.3.2.1 List Publications and Conference

A. **Zouhir Bellal**, Laaziz Lahlou, Nadjia Kara, Timothy Murphy, and Tan Phat Nguyen, "K8S Power Irrigation: Deep Reinforcement Learning for Performance-Aware Power Efficiency of Kubernetes Cloud-Native Microservices," submitted to *IEEE Transactions on Green Communications and Networking*, March 2026.

B. **Zouhir Bellal**, Laaziz Lahlou, Nadjia Kara, Timothy Murphy, Tan Phat Nguyen, and Arif Ahmed, "Investigating the Potential of Kepler Towards Power Observability for Sustainable Cloud Computing," accepted in *IEEE Transactions on Green Communications and Networking*, 2026.

C. **Zouhir Bellal**, Laaziz Lahlou, Nadjia Kara, and Ibtissam El Khayat, "GAS: DVFS-Driven Energy Efficiency Approach for Latency-Guaranteed Edge Computing Microservices," accepted in *IEEE Transactions on Green Communications and Networking*, 2024.

D. **Zouhir Bellal**, Laaziz Lahlou, Nadjia Kara, Timothy Murphy, and Tan Phat Nguyen, "EC6: Enhancing Energy Efficiency in Kubernetes Through Dynamic Extension of CPU Deep Idle States (C6)," in *2025 16th International Conference on Network of the Future (NoF)*, pp. 200–208, 2025.

E. **Zouhir Bellal**, Laaziz Lahlou, Nadjia Kara, Timothy Murphy, and Tan Phat Nguyen, "3GC: A Deadline-Aware and Energy-Efficient Resource Allocation Scheme for Serverless Edge Computing," in *2025 IEEE 22nd Consumer Communications & Networking Conference (CCNC)*, pp. 1–9, 2025.

0.3.2.2 Research Impact and Knowledge Transfer.

Beyond its academic contributions, this thesis has also attracted interest from the practitioner community. Parts of the results have been disseminated not only through academic venues, but also through industrial and professional communities, reflecting their practical relevance to real-world cloud-native power management. In particular, the outcomes of this work received strong interest and support from our industrial partner, *Ericsson*. In addition, selected results were presented to practitioner-oriented communities, including the *Cloud Native Computing Foundation (CNCF) Montreal* community and the *Green Software Foundation*, where they contributed to broader discussions on sustainable cloud computing and energy-aware software infrastructures.

0.4 Thesis Organization

This thesis is organized as follows. The literature review chapter presents the overall research methodology, including the literature review, gap identification, and problem formulation across edge power efficiency, container observability, and cloud power efficiency. Chapter 1 addresses performance-aware energy efficiency at the edge through DVFS-based control for latency-sensitive microservices. Chapter 2 investigates container-level power observability and introduces an accuracy-validation framework for power monitoring tools in cloud-native environments. Chapter 3 focuses on cloud-side power efficiency and presents a reinforcement learning–based approach for performance-aware power control under multi-tenant interference. Finally, the appendices provide additional material extending the thesis contributions to serverless edge clusters and idle-state power management in Kubernetes.

0.5 Research Methodology

Figure 0.1 summarizes the overall methodology adopted in this thesis. The methodology is organized as a progressive research process that begins with the overall research goal, proceeds through literature review

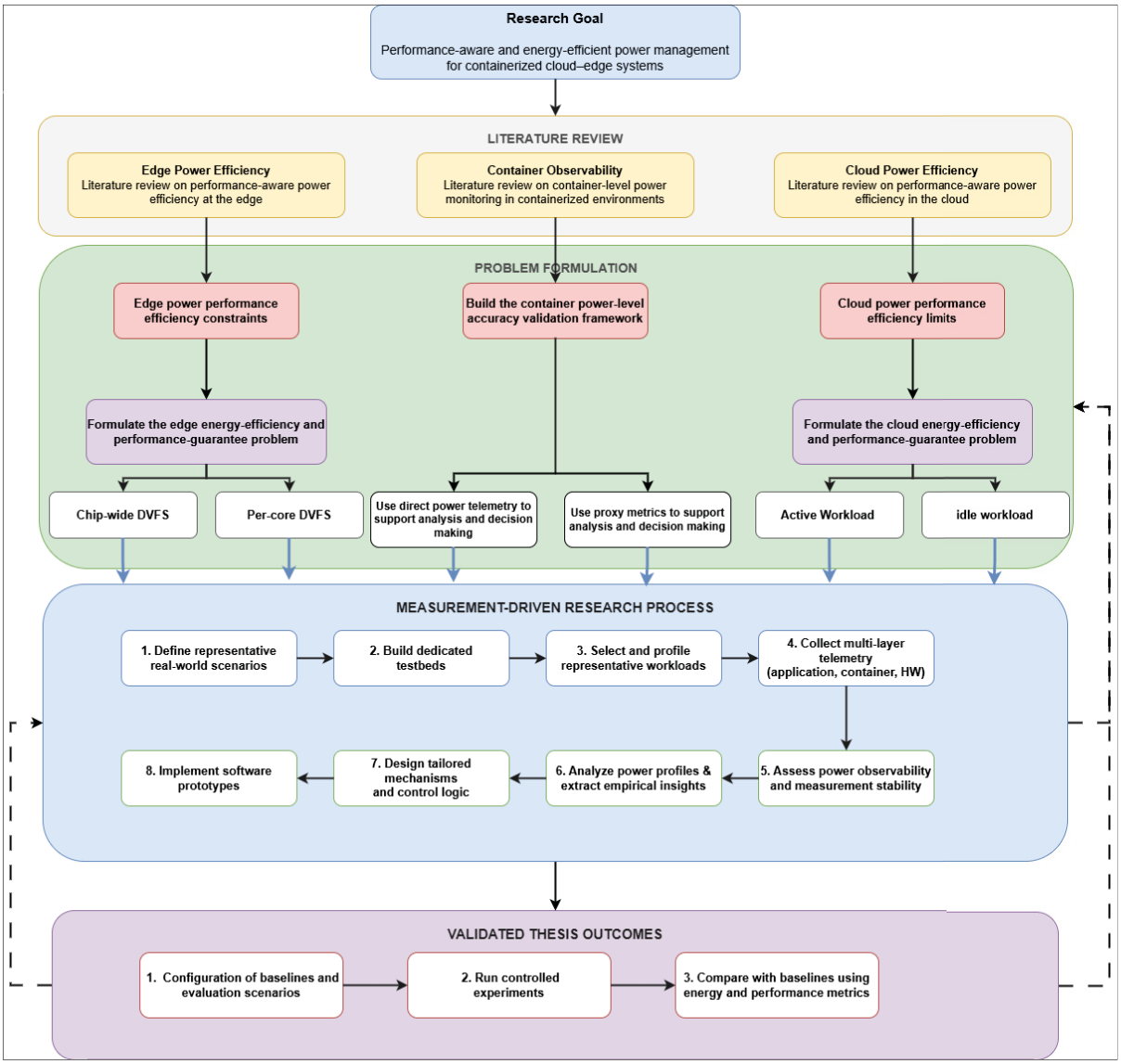


Figure 0.1 Overall methodology

and problem formulation, and then converges into a common measurement-driven process for solution design, implementation, and validation. The central objective is to develop mechanisms for performance-aware and energy-efficient power management in containerized cloud–edge systems.

This thesis is motivated by the need to reduce power and energy consumption in containerized environments without violating application-level performance requirements. The problem is studied across the cloud–edge continuum because the available control knobs, deployment constraints, workload characteristics, and sources of inefficiency differ between edge and cloud systems. The methodology therefore considers both environments within a unified research framework while accounting for their distinct operational conditions.

0.5.1 Research Motivation and Gap Identification

The methodology is grounded in a structured literature review that is systematically organized into three core domains, reflecting the primary dimensions of the research problem and enabling a targeted identification of existing limitations and open challenges.

0.5.1.1 Edge power efficiency

The first review domain focuses on performance-aware power efficiency at the edge. It examines prior work on hardware-based power-control mechanisms, particularly DVFS, latency-aware control, heterogeneous edge infrastructures, and serverless edge computing. The investigation covers both single-node and multi-node power-control strategies. The purpose of this review is to identify the limitations of existing methods when hardware control is constrained and workload requirements are heterogeneous (CPU-intensive, memory-intensive, mixed).

0.5.1.2 Container observability

The second review domain focuses on container-level power monitoring and observability in containerized environments. It examines the available monitoring tools, their modeling assumptions, their observability limits, and their suitability for supporting runtime analysis and control decisions. This review is essential because any later control strategy depends on the reliability of the underlying power information.

0.5.1.3 Cloud power efficiency

The third review domain focuses on performance-aware power efficiency in cloud systems. It covers cloud-native runtime control, multi-tenancy, resource interference, core and uncore power management, and Kubernetes-based execution environments. It also investigates how hardware power-efficiency features, such as DVFS, uncore frequency scaling, and idle-state control, can be exploited in cloud environments. The purpose of this review is to identify why existing cloud power-management methods remain insufficient for performance-sensitive containerized workloads.

0.5.2 Problem Formulation

After the literature review, the methodology moves to formal problem definition.

0.5.2.1 Edge-side problem formulation

The edge problem is formulated as an energy-efficiency and performance-guarantee problem under constrained hardware control. Two control granularities are considered:

- **chip-wide DVFS**, where co-hosted services share a common frequency decision;
- **per-core DVFS**, where CPU frequency can be controlled more selectively.

This formulation is considered under both single-node and multi-node heterogeneous edge settings. The objective is to determine how hardware power-control decisions can be adapted to workload behavior while preserving performance requirements.

0.5.2.2 Observability problem formulation

The observability problem is investigated through the following question: can container-level power-monitoring tools be used reliably to support analysis and runtime decision making? Addressing this question requires defining how such tools should be evaluated, under which runtime conditions they fail, and whether their outputs remain sufficiently accurate to support a runtime decision-making system.

To address this problem, the methodology establishes a rigorous validation framework to determine whether container-level power-monitoring tools provide sufficiently reliable data for analysis and control. This stage also supports a critical methodological decision:

- if a tool achieves acceptable accuracy, its direct telemetry can be used to support analysis and decision making;
- otherwise, proxy metrics for container power consumption must be identified to support the decision-making process.

0.5.2.3 Cloud-side problem formulation

The cloud problem is formulated as an energy-efficiency and performance-guarantee problem in realistic cloud-native environments. Two execution conditions are considered:

- **active workload conditions**, where the challenge is runtime control under heterogeneous microservices, load variation, and shared uncore interference;

- **Idle workload conditions**, in which microservices may continue to consume power even when they are idle from a business-logic perspective; for example, a web service receiving no HTTP requests may still retain and consume system resources.

The objective is to determine how power-control mechanisms can be applied in cloud environments while remaining consistent with application-level performance requirements and multi-tenant execution constraints.

0.5.3 Measurement-Driven Research Process

Once the problems are formulated, the methodology converges into a common measurement-driven research process. This constitutes the core of the thesis.

0.5.3.1 Define representative real-world scenarios

For each problem context, representative real-world scenarios are defined. This includes the deployment setting, workload type, performance target, and hardware-control scope. The purpose is to ensure that the research remains grounded in realistic operating conditions rather than abstract assumptions.

0.5.3.2 Build dedicated testbeds

Dedicated experimental infrastructures are then constructed for both edge and cloud studies. These testbeds are designed to provide control over hardware settings (e.g., computing servers), workload placement, and system noise, thereby enabling reproducible and interpretable measurements.

0.5.3.3 Select and profile representative workloads

Representative workloads are selected and profiled under the relevant hardware-control knobs. This includes realistic workloads and well-established benchmarks, such as `stress-ng` and memory-bandwidth benchmarks. This stage characterizes how different workloads and performance metrics respond to changes in frequency, control granularity, and runtime conditions. It is especially important because CPU-bound, memory-bound, and mixed services do not exhibit the same power–performance behavior.

0.5.3.4 Collect multi-layer telemetry

The methodology collects telemetry from multiple layers of the system, including:

- application-level metrics;

- container or runtime metrics;
- hardware-level counters and power-related signals.

This multi-layer view is required because no single metric is sufficient to explain the behavior of cloud-native workloads under realistic operating conditions.

0.5.3.5 Assess power observability and measurement stability

The collected telemetry is then assessed for reliability. This stage determines whether container-level power signals remain stable and trustworthy under realistic runtime factors such as frequency scaling, execution-state changes, and co-runner interference. Different hardware and software mechanisms are used to reduce noise and stabilize the measurements.

0.5.3.6 Analyze power profiles and extract empirical insights

The power and performance profiles are then analyzed to extract the key empirical insights. These insights may include workload sensitivity to control knobs, interference effects, and the reliability limits of certain telemetry signals. This stage transforms raw measurements into design knowledge.

0.5.3.7 Design tailored mechanisms and control logic

The extracted insights are then used to design problem-specific mechanisms. These mechanisms target constrained edge control, heterogeneous allocation, or cloud runtime power control. The key principle is that the design is derived from measurement and analysis rather than from generic heuristic assumptions.

0.5.3.8 Implement prototypes

The designed mechanisms are implemented as working prototypes in the experimental platforms. This step is necessary to verify practical feasibility, integration with orchestration environments, control responsiveness, and deployment constraints.

0.5.4 Evaluation and Validation

The final stage of the methodology is a common evaluation process used to validate the proposed mechanisms.

0.5.4.1 Configuration of baselines and evaluation scenarios

Relevant baselines are first selected, including default Linux governors, default Kubernetes behavior, or other baseline strategies representative of current practice. Evaluation scenarios are also defined to reflect realistic edge and cloud workload conditions.

0.5.4.2 Run controlled experiments

Controlled experiments are then executed on the dedicated testbeds and, where relevant, complemented by simulation or emulation. The purpose is to compare the proposed mechanisms with baseline strategies under consistent and reproducible conditions.

0.5.4.3 Compare using energy and performance metrics

The comparison is based on both energy-related and performance-related metrics. These include:

- power and energy consumption;
- latency or deadline satisfaction;
- performance violations.

When relevant, the evaluation may also consider additional indicators such as interference effects, robustness under dynamic load, and execution-state behavior.

0.5.5 Thesis Outcome

The outcome of this methodology is a set of validated practical methods for performance-aware and energy-efficient power management across the cloud–edge continuum. More importantly, the methodology establishes a complete research path from literature grounding and problem formulation to measurement-based design, implementation, and validation. In this sense, the contribution of the thesis is not limited to individual mechanisms; it also lies in the systematic way power efficiency, observability, and performance guarantees are studied together in containerized environments.

CHAPTER 1

GAS: DVFS-DRIVEN ENERGY EFFICIENCY APPROACH FOR LATENCY-GUARANTEED EDGE COMPUTING MICROSERVICES

Zouhir Bellal, Laaziz Lahlou, Nadjia Kara and Ibtissam El Khayat

Department of Software Engineering and Information Technology at École de technologie supérieure,

1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Article published in the journal

IEEE Transactions on Green Communications and Networking

on MARCH, 1, 2025

Abstract: Edge computing-based microservices (ECM) are pivotal infrastructure components for latency-critical applications such as Virtual Reality/Augmented Reality (VR/AR) and the Internet of Things (IoT). ECM involves strategically deploying microservices at the network's edge to fulfill the low latency needs of modern applications. However, achieving efficient resource and energy consumption while meeting the latency requirement in the ECM environment remains challenging. Dynamic Voltage and Frequency Scaling (DVFS) is a common technique to address this issue. It adjusts the CPU frequency and voltage to balance energy cost and performance. However, selecting the optimal CPU frequency depends on the nature of the microservice workload (*e.g.,* CPU-bound, memory-bound, or mixed). Moreover, various microservices with different latency requirement can be deployed on the same edge node. This makes the DVFS application extremely challenging, particularly for a chip-wide DVFS implementation for which CPU cores operate at the same frequency and voltage. To this end, we propose *GAS*, ener**G**y **A**ware micro**S**ervices edge computing framework, which enables CPU frequency scaling to meet diverse microservice latency requirement with the minimum energy cost. Our evaluation indicates that our CPU scaling policy decreases energy consumption by 5% to 23% compared to Linux governors while maintaining latency requirement and significantly contributing to sustainable edge computing.

Keywords: Edge Computing, Microservice, DVFS, Energy-efficient, Container Autoscaling

1.1 Introduction

In an era dominated by digitalization and the rise of technologies based on artificial intelligence, the demand for computing capacity has increased, making the IT infrastructure increasingly power-hungry. Globally, data centers account for approximately 3% of total electricity consumption and are expected to increase to 4%, roughly 3 trillion

kWh, in the next decade (Ene). This rapid escalation in energy demand stresses the need for more sustainable and energy-efficient solutions. Cloud data centers contribute significantly to global energy consumption, but they are not the only contributors. The need for low latency and high bandwidth in modern applications renders conventional clouds inadequate. Therefore, edge computing has emerged as a solution, positioning services closer to data sources to meet low-latency demands. In edge computing, applications are deployed as microservices on different edge nodes. Recent breakthroughs in container management and orchestration tools, such as K3s[1], MicroK8s [2], and KubeEdge [3], have been explicitly designed as alternatives to Kubernetes to operate in resource-constrained edge computing environments. This combination of edge computing as a deployment infrastructure and microservices as a design strategy results in a robust and flexible solution known as Edge Computing based Microservices (ECM) (Ermolenko *et al.*, 2021). This evolution emphasizes the role of edge computing in today's technological landscape, especially in its contribution to overall energy consumption. Consequently, the development of energy-efficient approaches explicitly tailored for edge computing has evolved from an option to a necessity.

In response to the critical issue of energy consumption, numerous strategies have been proposed (Ma *et al.*, 2022; Masip-Bruin *et al.*, 2016; Mei *et al.*, 2021; Zhang *et al.*, 2013). Among these well-established power-saving methods, Dynamic Voltage and Frequency Scaling (DVFS) stands out (Weiser *et al.*, 1996). DVFS is a power optimization strategy that allows one to adjust the frequency and voltage of the CPU. Each frequency correlates with a specific performance level and a unique energy footprint. The main objective of DVFS is to balance performance levels and energy efficiency without compromising the performance requirement. The Linux kernel uses DVFS through various power governors. These governors control the CPU frequency to save energy or maximize performance. For example, the *powersave* governor runs the CPU at a low frequency to conserve power, while the *performance* governor uses the highest frequency for optimal performance (CPU). Besides these static governors, the Linux kernel also includes dynamic ones that adjust the CPU frequency based on CPU usage. However, these built-in power governors use generic algorithms that often overlook the specific characteristics of running tasks.

On the one hand, the performance of CPU-bound microservices is notably sensitive to the CPU frequency, and it is more beneficial in terms of energy consumption and performance to run them at higher frequencies. Reducing the frequency in the case of CPU-bound microservices may reduce power use but extend execution time, resulting in an increase in total energy consumption. On the other hand, the performance of memory-bound microservices has less sensitivity to CPU frequency, allowing for potential energy savings from frequency reduction without impacting their performance (Etinski *et al.*, 2012). However, speculating about the energy and performance behavior relative to CPU frequency in real-world microservice applications presents significant challenges. This complexity arises from the intricate interplay between CPU, Input/Output (I/O), and memory resource usage.

[1] https://k3s.io

[2] https://microk8s.io

[3] https://kubeedge.io

Furthermore, the application of DVFS in ECM environments poses significant challenges. Edge computing nodes face constraints that go beyond computational limitations, including hardware-specific limitations partially related to DVFS controllers. Various platforms, from embedded systems akin to the ODROID XU-3 (ODR) and NVIDIA Jetson (NVI) to specialized server processors from Intel and AMD (Doweck, Kao, Lu, Mandelblat, Rahatekar, Rappoport, Rotem, Yasin & Yoaz, 2017; Singh, Rangarajan, John, Henrion, Southard, McIntyre, Novak, Kosonocky, Jotwani, Schaefer et al., 2017), adopt Chip-Wide DVFS. Within this architecture, all CPU cores are constrained to operate at the same CPU frequency and voltage. The adoption of Chip-Wide DVFS over its alternative, per-core DVFS, stems mainly from the complexities and costs associated with the latter's implementation. Specifically, in a per-core DVFS architecture, each core requires its own voltage regulator, amplifying the design complexity and the chip's physical dimensions. Moreover, per-core voltage regulators generally suffer from a lower voltage conversion efficiency, in contrast to the single global voltage regulator used in the Chip-Wide DVFS architecture (Kim *et al.*, 2008). Edge nodes that employ Chip-Wide DVFS controllers in the context of ECM create a complex operational landscape. In this environment, numerous microservices with varying workload characteristics (e.g., CPU-bound, memory-bound, or mixed workloads) and diverse latency requirements are forced to operate under the same CPU frequency. Identifying an optimal CPU frequency that simultaneously satisfies the different latency requirements of all microservices while minimizing energy consumption creates an intricate challenge. To address these challenges, we introduce *GAS*, ener**G**y **A**ware micro**S**ervices edge computing framework, which enables CPU frequency scaling to optimize energy efficiency and guaranteed performance demands. *GAS* is described in section IV. This paper details *GAS*'s frequency scaling policy, which is designed specifically to:

1. Identify the optimal CPU frequency that minimizes energy consumption while satisfying the latency requirements of diverse microservices with various workload characteristics, all hosted on the same edge node.

2. Enable dynamic adjustment of CPU frequency to match microservices load (*e.g.,* queued requests), thus consistently maintaining the desired latency and energy efficiency.

3. Support less sophisticated edge devices with a Chip-Wide DVFS controller, where all microservices must operate under the same CPU frequency.

We implemented and evaluated *GAS* using real-world data collected from our home-grown testbed. Experimental results show that our approach significantly outperforms the OS power governors and reduces energy consumption by 5% up to 23%. At the same time, *GAS* ensures that latency requirements are met with no requests being violated. The structure of this paper is as follows. Section 2 introduces the background and related works, laying the groundwork for our proposed solution. Section 3 formulates the energy efficiency problem in microservice edge computing, considering latency requirements and Chip-Wide DVFS constraints. Section 4 presents our solution in

detail. Section 5 outlines the evaluation methodology. Section 6 presents the results and discusses the key findings. Section 7 discusses limitations and future research directions. Finally, Section 8 concludes the article.

1.2 Background & Related works

Introduced in the 90s, DVFS emerged as a promising technique for reducing power consumption in computing systems by adjusting both the voltage and frequency of the system with respect to the workloads (Macken, Degrauwe, Van Paemel & Oguey, 1990). Operating systems (OSs) use DVFS through different power governors to balance performance and energy savings. For example, the Performance governor consistently selects the maximum CPU frequency to maximize performance, albeit at the expense of power consumption. In contrast, the Powersave governor aims to minimize power usage by using the minimum CPU frequency but sacrificing performance. Governors like Schedutil and Conservative dynamically adjust the CPU frequency according to real-time system load metrics (*e.g.,* CPU usage). However, these governors are agnostic to the nature of applications' workload and latency requirements. Meeting the latency requirements of various hosted microservices in ECM environments is crucial. However, using such generic governors can compromise the latency requirements of these microservices (CPU). To mitigate this issue, various strategies have been developed to adjust the CPU frequency according to performance needs while ensuring energy efficiency. For example, Alzahrani et al. (Alzahrani, Tari, Zeephongsekul, Lee, Alsadie & Zomaya, 2016) propose the Energy-Based Auto Scaling (EBAS) approach, dynamically adjusting resource allocation and core frequency to meet SLA requirements while minimizing energy. The strategy utilizes a predictive model to proactively allocate CPU cores and configure their frequency based on incoming requests. The work of (Chang & Liang, 2011; Lin, Chen, Liu & Wu, 2018; Pietri & Sakellariou, 2014; Zhang, Wang & Hu, 2015) also leverage DVFS, adjusting the CPU frequency to minimize total energy consumption during task execution while ensuring that task deadlines are met. Recent work in (Yang, Zhao, Li & Zhang, 2024) introduces E2DSched, a reinforcement learning approach for dependent application (*i.e.,* Directed Acyclic Graph (DAG) representing the job workflow) scheduling based on DVFS. The objective is to determine an optimal CPU frequency for edge nodes to meet the request deadlines of the whole application while minimizing energy consumption. The selection of CPU frequency for each task is strategically determined by the task's importance within the DAG. For tasks where execution time minimally impacts the overall QoS, a lower CPU frequency is preferred to reduce energy consumption. Conversely, if a task is identified as critical within the DAG structure, a higher frequency is prioritized to minimize execution time.

However, the effects of DVFS can differ significantly depending on the nature of the application workload (*e.g.,* CPU-intensive, memory-intensive, or I/O-intensive), and it is not always advantageous. Therefore, it is imperative to consider workload characteristics when applying DVFS. To address this challenge, Patrou et al. (Patrou, Kent, Siu & Dawson, 2022) present the Energy-Aware Policy. Unlike other approaches, this policy includes an initial profiling step to evaluate the workload of requests and adjust CPU frequencies accordingly. Thus, requests

are categorized into two groups: CPU-driven, which benefits from high CPU frequencies, and non/less CPU-driven, which is better served at lower CPU frequencies to save energy. A request is considered non/less CPU-driven if it consumes less energy on the CPU cores than on the Dynamic Random Access Memory (DRAM) and uncore. Otherwise, it is classified as CPU-driven. This policy enforces the maximum CPU frequency for CPU-driven requests, while for non-/less CPU-driven requests, it selects a CPU frequency from a predetermined list (*e.g.,* [1.30 GHz - 1.70 GHz]). These frequency values were identified through empirical analysis (Patrou, Kent, Siu & Dawson, 2021). Using a predefined frequency list for microservices may not be optimal as it relies on limited benchmarks that may not account for diverse workloads. Moreover, this approach is closely tied to the Node.js runtime environment, which may limit its applicability to diverse use case. Furthermore, when handling multiple concurrent workloads, it selects the highest CPU frequency, which does not necessarily result in optimal energy efficiency.

In (Tang, Ke, Fu, Jiang, Wu, Peng & Gao, 2022), the authors propose Demeter, a QoS-aware power management controller for heterogeneous workloads in public clouds. Demeter's profiler initially marks workloads as latency-critical (LC), but if they exhibit high CPU usage and minimal network traffic, they are reclassified as best-effort (BE). The main difference between the two workloads is that LC workloads enjoy strict low latency (*e.g.,* social media services and search engines). In contrast, BE workloads have no strict latency requirements (*e.g.,* offline analytics). For frequency scaling, the Demeter runs LC workloads at high frequency while selecting the optimal frequency for BE workloads based on CPU usage. Demeter partitions server CPU cores into three areas: hot, warm, and cold. The hot area hosts LC workloads, the warm area serves BE workloads, and the cold area contains idle cores. The number of cores in each area is dynamically adjusted based on the CPU resource demands from running workloads. However, the frequency settings determined by a DVFS approach may be overridden or even not applied during the power-throttling process. In fact, modern CPUs come with a fixed Thermal Design Power (TDP), indicating the maximum power dissipation. When a CPU's power usage reaches the TDP, the power-throttling process is triggered, reducing the frequency of all cores to mitigate thermal risks. Although this mechanism effectively controls thermal output, it can degrade performance. Therefore, it is crucial for DVFS-based solutions to consider the power throttling process to maintain the integrity of Quality of Service (QoS) commitments (Kumbhare, Azimi, Manousakis, Bonde, Frujeri, Mahalingam, Misra, Javadi, Schroeder, Fontoura et al., 2021). Demeter temporarily stops frequency scaling in the warm area when TDP is reached, setting all warm area core frequencies to 3.0 GHz until power stabilizes and falls below TDP. However, running LC workloads at high frequencies due to low latency needs, regardless of workload types, can result in inefficient energy use. Even non-/less CPU-intensive workloads can be latency-critical, as latency requirements often derive from the service owner and not from the workload nature. Moreover, Demeter requires specific hardware prerequisites. It needs a multi-core processor to enable core separation into different areas. For the independent control of the CPU frequency in each area, the support of per-core DVFS is mandatory. However, some edge computing devices may lack these features, creating a potential barrier to Demeter's successful setup.

In (Jia & Zhao, 2021), the authors introduced RAEF, a runtime system that optimizes energy efficiency without compromising the latency of serverless functions. RAEF employs a binary search approach to find the optimal balance between latency and energy consumption by exploring two-dimensional resources, namely frequency and CPU core count. Similarly, (Rastegar, Shafiei & Khonsari, 2023; Tzenetopoulos, Masouros, Soudris & Xydis, 2023) optimize serverless workflows. It adjusts the core frequency through a parameterized linear function based on frequency and latency considerations for optimal performance.

We have conducted a comparative analysis with related work in the field, focusing on the following key features:

- **Multi-microservice DVFS Configuration:** Considers multiple hosted microservices and meets diverse latency requirements of different hosted microservices.
- **Multi-workload Nature:** Accounts for the varied nature of workloads (CPU-bound, memory-bound, and mixed) associated with different hosted microservices.
- **Chip-Wide DVFS Support:** Operates on edge nodes that support chip-wide DVFS.
- **Online DVFS Configuration:** Dynamically adjusts CPU frequencies based on real-time microservice load (*e.g.,* number of requests).
- **Power-Throttling Awareness:** Mitigates performance degradation during the power-throttling process to maintain consistent service quality.
- **Specific Requirement:** Requires specific infrastructure or custom runtime environments to operate.

Table I-1 summarizes the comparative analysis, highlighting how our approach aligns with and improves upon existing works.

Table 1.1 Summary of existing works

References	Multi-microservice	Multi-workload Nature	Chip-Wide DVFS Support	Online DVFS Configuration	Power-Throttling Awareness	Specific Requirement
(Alzahrani *et al.*, 2016)	✓	x	x	x	✓	No
(Pietri & Sakellariou, 2014)	✓	x	x	x	x	No
(Zhang *et al.*, 2015)	✓	x	x	x	x	No
(Lin *et al.*, 2018)	✓	x	x	x	x	No
(Chang & Liang, 2011)	x	✓	x	x	✓	No
(Patrou *et al.*, 2022)	✓	✓	x	✓	x	Node js runtime
(Tang *et al.*, 2022)	✓	✓	x	✓	✓	No
(Yang *et al.*, 2024)	✓	✓	x	✓	x	No
(Jia & Zhao, 2021)	x	✓	x	✓	✓	No
(Tzenetopoulos *et al.*, 2023)	✓	x	✓	x	✓	No
(Rastegar *et al.*, 2023)	✓	✓	x	x	x	No
GAS	✓	✓	✓	✓	✓	No

In contrast to existing approaches, our proposed solution, *GAS* is 1) exclusively designed for ECM environments. It is particularly beneficial for edge devices that are constrained to operate all cores with one frequency, as in Chip-Wide DVFS and single-processor architectures. *GAS* can 2) dynamically adjust the CPU frequency to guarantee stringent latency requirements for all hosted microservices at minimal energy costs. Furthermore, the proposed solution 3) prevents performance degradation during the power throttling process by blocking any additional incoming requests to the edge.

1.3 Motivation

In the context of a smart city, three diversified IoT applications, each presenting a unique workload nature and latency requirements. These IoT applications are deployed as microservices on the same edge node, utilizing containerization technologies like Docker for simplified management and deployment:

- *Traffic Management System (CPU-Intensive)*: Processes real-time video using machine learning for vehicle and pedestrian detection at major intersections, requiring high CPU resources and low-latency responses for effective traffic flow management.
- *Environmental Monitoring System (Mixed Workload):* Utilizes both memory and CPU resources to analyze data from sensors monitoring air quality and weather conditions, crucial for timely public health and safety alerts.
- *Local Video Caching Service (Memory-Intensive):* Supports large events by storing and delivering popular video content on-demand, demanding extensive memory for high-quality, rapid media streaming.

The challenge lies in maintaining the distinct latency and workload requirements of these applications while minimizing energy consumption, particularly under the constraints of a Chip-Wide DVFS controller. This controller mandates all CPUs to operate at the same frequency. The built-in power governor often selects CPU frequency that does not fit all host applications, leading to potential mismatches with different application needs and resulting in either poor performance or excessive energy use.

Our proposed solution, GAS, considers the distinct latency demands and workload specifics of each microservice to dynamically select the optimal CPU frequency. This approach not only ensures that each application meets its latency requirements but also optimizes overall energy consumption.

1.4 Assumptions & System model

In this section, we list our assumptions and introduce the system model of the proposed solution. Fig. I-1 illustrates a reference model for edge computing clusters consisting of multiple worker nodes that host various microservice instances. At the edge level, *GAS* leverages DVFS to provide an energy-efficient CPU frequency scaling policy. This

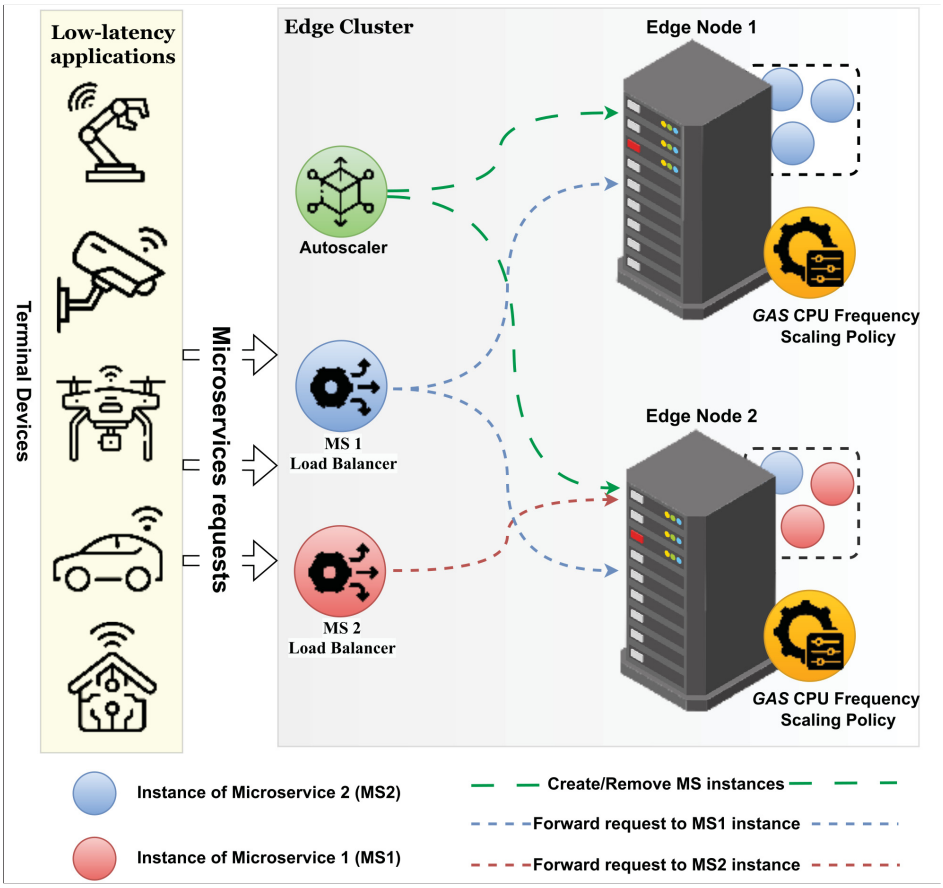


Figure 1.1 Reference model of *GAS*

policy allows edge nodes to operate at the optimal CPU frequency, ensuring microservices' response times meet latency requirements at the lowest possible energy cost. At the cluster level, *GAS* can improve the decision-making in autoscaling policy by triggering the creation of a new microservice instance in response to high load, thereby maintaining the latency requirements.

1.4.1 Assumptions

We formulate our problem considering the following assumptions:

1. ***Homogeneous Edge nodes:*** We assume that all edge nodes in our cluster are homogeneous. They share identical software stacks, including the operating system and container engine, and have consistent hardware attributes, such as the CPU type and supported frequencies.

2. ***Chip-Wide-DVFS controller support:*** This implies that all CPU cores operate at the same frequency at any given time.

3. *Steady frequency over a period of time:* We assume that the voltage/frequency setting remains the same during a given timeslot τ.

4. ***Microservice's characteristics*** This study addresses a microservice category distinguished by two characteristics:

   a. *Sequential Request Handling:* Microservices process requests sequentially due to business logic or runtime constraints.

   b. ***Stable Workload:*** Microservices ensure consistent computational workload due to input standardization, as illustrated by the following examples.

      - *QR Code Generator Microservice:* This microservice accepts a fixed-length string (e.g., URL) and outputs a QR code image. QR code generation largely depends on the length of the string, not its content. Thus, with varied inputs, the workload remains consistent.
      - *Video Generator Microservice:* This microservice takes a script to produce a video where an AI agent acts and speaks the script lines. This relies on Natural Language Processing to understand the script, Text-to-Speech for the agent's dialogue, and an AI model for animating the AI agent. With a consistent script length, the video generation workload should remain stable.

1.4.2 System Model

In this section, we present our system model and problem formulation.

Edge Cluster Model: We consider a system where a master node manages a cluster C, constituting a set of edge nodes represented as $C = \{e_1, \ldots, e_n\}$. As a typical strategy to handle online job scheduling. We define $\mathcal{T}$ as a set of fixed time instances $\mathcal{T} = \{t_1, ..., t_s\}$ and $\forall t_i \in \mathcal{T}$, $t_{i+1} - t_i = \tau$, where τ is a fixed time slot. Each time instance represents a discrete decision point for the system. We use $t_{now} \in \mathcal{T}$ to denote the current system time instance. In the rest of this model, the notations are presented for a time instance t_{now} and are summarized in Table I-2. As edge nodes share the same hardware characteristics, they support an identical set of valid CPU frequencies denoted by $F = \{f_1, \ldots, f_y\}$. The minimum and maximum CPU frequencies allowed are determined by $f_{minF} = \min\{F\}$ and $f_{maxF} = \max\{F\}$, respectively. In addition, we use $\varsigma_k \in F$ to indicate the current CPU frequency of the edge node e_k. Moreover, we use TDP to denote the TDP of the edges' processors, which refers to the average power consumption that a cooling system must be able to dissipate (specified by the manufacturer in watts). Finally, $P(e_k)$ refers to the current power consumption of the edge e_k.

Micorservices:

In edge computing environments, microservices can be deployed across various edge nodes using multiple instances. We define the set of all microservices as $\mathcal{M} = \{M_1, \ldots, M_s\}$, and instances of microservice M_j as

Table 1.2 Summary of the most commonly used notations

Model Parameters	
$t_{now} \in \mathcal{T}$	The current system time instance
$e_k \in C$	The k-th edge node within the cluster C
F	The set of valid CPU frequencies for edge nodes
$f_{minF} \in F$	The minimum supported CPU frequency for all edge node
$f_{maxF} \in F$	The maximum supported CPU frequency for all edge node
$\varsigma_k \in F$	The current CPU frequency of the edge node e_k.
TDP	Processor's Thermal Design Power for all edges
$P(e_k)$	The current power consumption of the edge e_k
$\mathcal{M}$	The set of microservices deployed on the cluster cluster C
$M_j \in \mathcal{M}$	The j-th microservice deployed at the cluster C
$\mathcal{I}_j$	The set of instances of microservice M_j
RPM_j	The number of requests in the global queue for the microservice M_j
S_j	Maximum tolerated latency for microservice M_j requests in seconds.
L_j	The maximum number of waiting requests in the global queue to trigger the initiation of a new instance for M_j.
χ_m^j	The time to process a request for M_j at frequency f_m
λ_m^j	The energy to process a request for M_j at frequency f_m
$I_q^j \in \mathcal{I}_j$	The q-th instance of the microservice M_j
$\xi_q^j \in C$	The host of the instance I_q^j
Φ_k	The set of hosted instances at edge node e_k.
RPI_q^j	The number of waiting requests in the local queue for the instance I_q^j
Q_q^j	The queue of instance I_q^j
$r_p \in Q_q^j$	The p-th waiting request for execution by I_q^j
a_p	The arrival time of request r_p
b_p	The time when the request r_p starts to execute.
V_q^j	a parameter indicating if the queue of the instance I_q^j is non-empty i.e. $V_q^j = 1$ and 0 otherwise.
γ_m^k	a decision variable indicating if $f_m \in F$ is selected as CPU frequency at edge node e_k (i.e. $\gamma_m^k = 1$) or not (i.e. $\gamma_m^k = 0$)

$\mathcal{I}_j = \{I_1^j, I_2^j, \ldots, I_q^j\}$. The deployment of each instance, requiring resources such as CPUs, memory, and storage, is predefined in the microservice's deployment file (e.g., the YAML deployment file as in Kubernetes).

We define the latency for each microservice M_j as S_j, representing the time required to process a request. A common approach for managing incoming requests involves a load balancer that places these requests in a global queue before forwarding them to microservice instances, ensuring efficient distribution across available instances. The number of requests currently queued for processing by microservice M_j is denoted by RPM_j.

As the number of queued requests increases and instances begin to experience overload, it is essential to scale up by creating new instances to maintain acceptable latency levels. To manage this effectively without frequent scaling, a threshold L_j is established, defining the maximum number of requests that can be queued and tolerated to be deleted beyond the specified latency S_j before creating a new instance.

As the number of queued requests increases and instances begin to experience overload, it becomes necessary to scale up by creating new instances to maintain acceptable latency levels. To avoid excessive instance creation, a threshold L_j is defined for each microservice M_j. This threshold represents the maximum number of requests that can wait in the global queue before triggering the creation of a new instance. It is configured according to the latency requirement S_j, the profiling information, and the expected time required to make a new instance available.

Both S_j and L_j are parameters defined by the microservice owner, allowing for customized control over service responsiveness. Further, we denote χ_m^j and λ_m^j as the time and energy required to process a request for microservice M_j at CPU frequency $f_m \in F$.

Micorservice instance:

Edge node can host multiple microservice instances; we designate $\xi_q^j \in C$ as the edge that hosts the q-th instance of the microservice M_j. We use $\Phi_k = \{I | \forall M_j \in \mathcal{M}, I_q^j \in \mathcal{I}_j, \xi_q^j = k\}$ to represent the set of hosted instances at the edge node e_k. Each instance, I_q^j, relies on a local queue to store the incoming requests. We denote the queue of instance I_q^j as $Q_q^j = \{r_1, r_2, \ldots, r_z\}$, where $r_p \in Q_q^j$ is a waiting request, and $RPI_q^j = |Q_q^j|$ as the number of queued requests for the instance I_q^j. Instances execute requests sequentially. We also use b_p and a_p to denote respectively, the time when the instance begins executing and the arrival time of the request r_p.

1.4.3 Problem Formulation

The problem of optimizing energy while guaranteeing latency in edge computing emerges when an edge node is required to meet the latency demands of various microservice instances, each characterized by distinct workload natures. This must be achieved under the constraint that all CPU cores of the edge node operate at the same CPU frequency at any given time, which is imposed by the chip-wide DVFS controller. In each time instance ($t_i \in \mathcal{T}$), we can formally define the problem as follows:

$$\min \sum_{I_q^j \in \Phi_k} \sum_{m=1}^{y} \gamma_m^k \times V_q^j \times \lambda_m^j$$

$$\text{s.t.} \quad \sum_{m=1}^{y} \gamma_m^k \times V_q^j \times (\chi_m^j + (b_p - a_p)) \leq S_j, \quad \forall e_k \in C, \forall I_q^j \in \Phi_k, \forall r_p \in Q_q^j \tag{1.1}$$

$$\sum_{m=1}^{y} \gamma_m^k = 1, \ \forall e_k \in C \tag{1.2}$$

$$\gamma_m^k \in \{0, 1\}, \ \forall e_k \in C, \ f_m \in F \tag{1.3}$$

Here, the decision variable γ_m^k indicates if $f_m \in F$ is selected as CPU frequency at edge node e_k (i.e. $\gamma_m^k = 1$) or not (i.e. $\gamma_m^k = 0$). V_q^j is a boolean parameter that indicates if the queue of the instance I_q^j is non-empty ($V_q^j = 1$) or not ($V_q^j = 0$), ensuring the model only considers instances with pending requests. V_q^j can be given as

$$V_q^j = \begin{cases} 1, & \text{if } RPI_q^j > 0, \\ 0, & \text{if } RPI_q^j = 0. \end{cases} \tag{1.4}$$

The problem is subject to a list of constraints. Constraint (1.1) ensures that the total time (the execution and waiting times) for any request does not exceed the specified latency defined in its microservice. Here, the term $b_p - a_p$ denotes the waiting time of the request r_p in the local queue Q_q^j. Constraint (1.2) ensures that all CPU cores operate at the same frequency, which is a constraint imposed by the chip-wide DVFS controller. Finally, constraint (1.3) ensures that the decision variable is binary, which is necessary for selecting a specific CPU frequency.

Proposition: The problem is NP-hard. The goal of our problem is to select a CPU frequency that can satisfy the request so that the total energy consumption is minimized and the execution time does not exceed the latency requirement. This problem falls into the well-known Discrete Time/Cost Tradeoff Problem (DTCTP), a recognized NP-hard problem (De, Dunne, Ghosh & Wells, 1997; Skutella, 1998). Therefore, the presented problem is NP-hard.

1.5 Overview of *GAS*

This section lists the essential components of *GAS* and how *GAS* interacts with the orchestration tool components. Fig. 1.2 presents the *GAS* architecture, which consists of three main components.

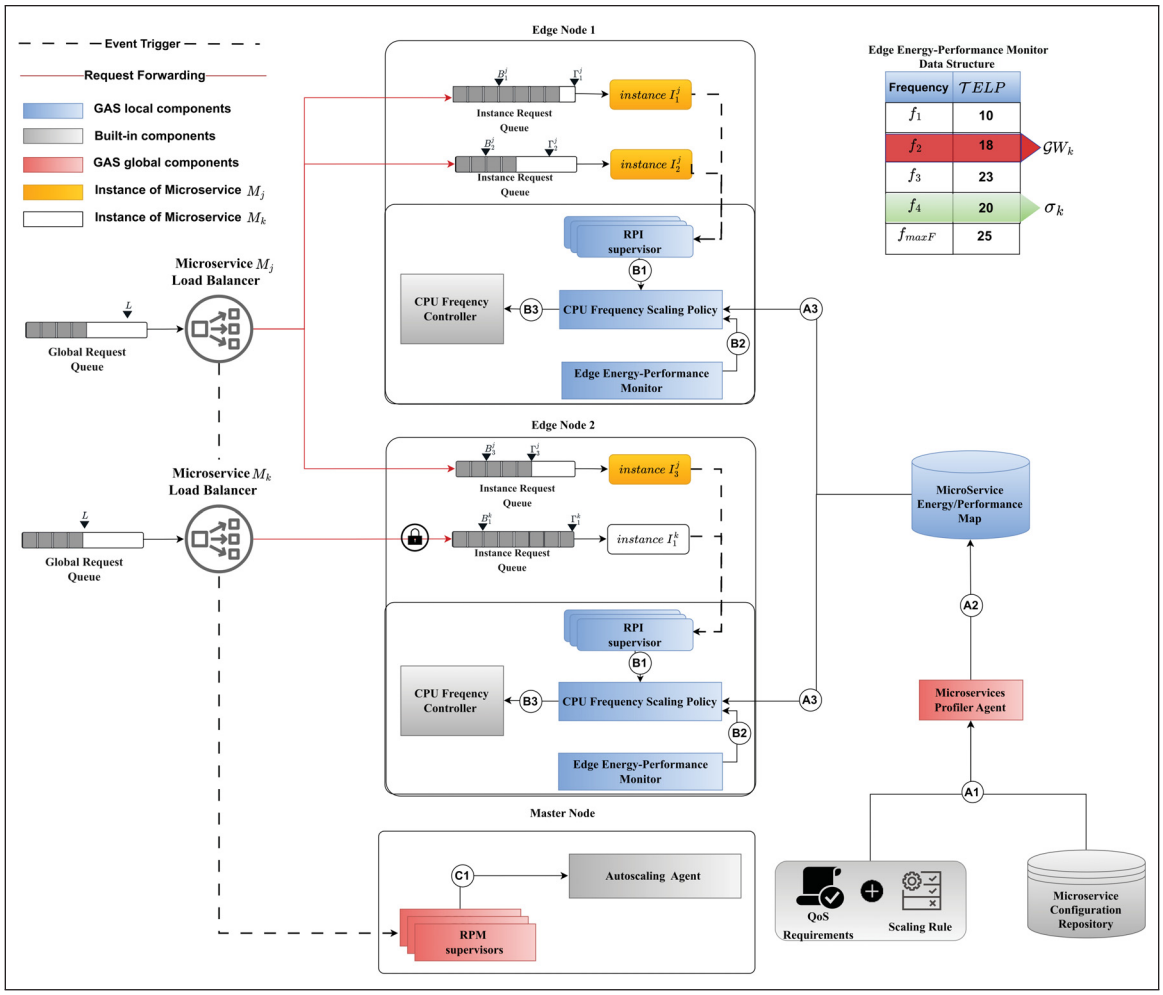


Figure 1.2 Overall architecture of *GAS*

1.5.1 Microservice Profiler Agent

The Microservice Profiler Agent collects data on the time and energy needed to complete a request at each CPU frequency to build a comprehensive understanding of the behavior of each microservice vis-a-vis CPU frequency. *GAS* maintains a repository of microservice deployment files within the cluster (see Fig. 1.2). When deploying a new microservice, the Profiler Agent initially utilizes these files to instantiate an instance of that microservice (See step A1 Fig. 1.2). A generic request is then used to invoke the microservice and stimulate a response.

The agent then measures the energy consumption and the time it takes to complete the execution of the request and repeats that for each supported CPU frequency. The collected data are used to build a microservice energy/performance map (See step A2 in Fig. 1.2). This map helps to understand the impact of CPU frequency on each microservice.

The profiling cost is linear in the number of supported CPU frequencies and the number of repetitions used to obtain stable measurements. If R profiling requests are executed at each CPU frequency, the number of profiling executions required for microservice M_j is:

$$N^j_{\text{prof}} = R \times |F|. \tag{1.5}$$

This profiling step is performed offline or at deployment time and is not part of the runtime critical path. Once the energy/performance map is built, *GAS* only reads the stored values of χ^j_m, λ^j_m, Υ^j_m, and ELP^j_m during runtime. Under the homogeneous-edge-node assumption, the same profile can be reused across edge nodes with the same hardware and software configuration. Reprofiling is required only when the microservice implementation, input characteristics, or deployment platform changes significantly.

The Agent integrates the microservice latency requirements into the map, such as microservice latency and the scaling rule that indicates when a new instance should be created (See step A1 Fig. 1.2).

1.5.2 CPU Frequency Scaling Policy

The CPU Frequency Scaling Policy is the heart of *GAS*. Its goal is to dynamically adjust CPU frequency, ensuring that all hosted microservices on the edge node meet latency requirements while minimizing energy consumption.

In this work, *GAS* controls the CPU frequency through the operating-system DVFS interface. Voltage is not directly controlled by *GAS*; rather, it is implicitly managed by the hardware and operating system as part of the processor's supported voltage/frequency operating points. Therefore, when *GAS* selects a CPU frequency, the platform applies the corresponding voltage setting according to its DVFS configuration. The proposed policy operates only within the official CPU frequency range exposed by the system and does not modify voltage tables. The problem formulated in Section 1.4.3 involves dynamically adjusting CPU frequencies in real-time based on the fluctuating demand of various microservices. To address this problem, we introduce GAS, an event-driven rule-based approach which continuously monitors the number of requests for each microservice instance and makes immediate adjustments to CPU frequencies based on a rule system that can be efficiently triggered by events (e.g., the number of requests for an instance crossing predefined thresholds).

This event-driven nature aligns well with the operational requirements of edge computing environments, where rapid adjustments are necessary to handle bursts in demand and prevent latency violations.

GAS splits the CPU frequency adjustment into two main processes:

1. **Global Optimal Frequency Selection (*GOFS*)**:

This process is invoked when an instance needs to adjust the CPU frequency of the edge node, considering the latency requirements of all hosted instances to determine the optimal CPU frequency that minimizes energy consumption.

2. **Acceptable Frequency Adjustment *(AFA)***:

   For each instance, GAS independently determines its minimum acceptable CPU frequency to maintain latency guarantees and invokes the GOFS process when necessary to adjust the CPU frequency of the edge node.

By combining these two processes, *AFA* limits the CPU frequencies to those that ensure latency requirements (feasible frequencies) and thus narrows the solution space to feasible frequencies, which significantly reduces the complexity of the optimization problem. Meanwhile, *GOFS* evaluates the set of feasible frequencies (those higher than the minimum acceptable frequency) and selects the most energy-efficient option. This integration ensures both immediate responsiveness to changing workloads and overall system efficiency.

1.5.2.1 Global Optimal Frequency Selection (GOFS)

This process involves selecting the optimal CPU frequency that meets the latency requirements of different microservice instances. Algorithm 1 illustrates this selection process. To achieve this, GAS needs first to enumerate the appropriate CPU frequencies for each instance on the edge node to guarantee the required latency. This is achieved using a threshold-based policy.

We use Υ_m^j to represent the maximum throughput of microservice M_j at CPU frequency f_m. Throughput refers to the number of requests that can be processed sequentially without compromising the latency requirements of the microservice and is given by:

$$\Upsilon_m^j = \frac{S_j}{\chi_m^j} \tag{1.6}$$

Thus, to determine the feasible CPU frequencies that guarantee latency requirements for instance I_q^j, we use the following expression:

$$\widetilde{F}_q^j = \{f_m \mid \forall f_m \in F, \Upsilon_m^j - \frac{t_{now} - \min\{a_p \mid r_p \in Q_q^j\}}{\chi_m^j} \geq RPI_q^j\} \tag{1.7}$$

The term $t_{now} - \min\{a_p | r_p \in Q_q^j\}$ refers to the longest waiting time of a request for the instance I_q^j, in which the term $\min\{a_p | r_p \in Q_q^j\}$ indicates the earliest request arrival time in the queue. The optimal frequency in GAS is the one with the minimum energy loss per second while ensuring the required latency. We denote ELP_m^j as the energy

loss per second when processing a request from microservice M_j using CPU frequency f_m, which is calculated as:

$$ELP_m^j = \frac{\lambda_m^j - \min\{\lambda_{m'}^j | \forall f_{m'} \in F\}}{\chi_m^j} \tag{1.8}$$

In essence, the energy loss of a microservice at a given frequency is gauged against the minimal energy loss per second that can be obtained. It highlights the variance between the energy used at a particular frequency and the least possible energy usage. For example, let us consider that a request from microservice M_1 is executed at CPU frequencies f_1 = 2.2GHz, f_2 = 2.6GHz, and f_3 = 3.0GHz. The energy-time pairs for processing a request at f_1, f_2, and f_3 are respectively (10, 5), (12, 4), and (8, 2). Thus, the energy loss per second for the frequencies f_1, f_2, and f_3 is 0.4, 1, and 0 respectively, indicating that f_3 results in minimal energy loss and thus it could be the best frequency to run microservice M_1 if it maintains the latency requirement. Note that ELP_m^j and Υ_m^j are calculated once during the profiling step (see step A2 in Fig. 1.2) and provided by the MicroService Energy/Performance Map (see step A3 in Fig. 1.2).

To determine the optimal frequency, our policy begins by identifying the minimum CPU frequency that can guarantee the latency requirements of all hosted instances on a given edge node (Alg. 1 line 3). We denote W_q^j as the lowest acceptable frequency for instance I_q^j that meets the latency requirement of microservice M_j:

$$W_q^j = \min\{\widetilde{F}_q^j\} \tag{1.9}$$

Thus, the global lowest acceptable frequency that ensures the latency requirements of all instances on edge node e_k is given by:

$$\mathcal{G}W_k = \max\{W_q^j \mid I_q^j \in \Phi_k\} \tag{1.10}$$

Our policy uses the global lowest acceptable frequency $\mathcal{G}W_k$ as the bottom threshold for the optimal CPU frequency (Line 2 Alg 1). With this safeguard, *GAS* can avoid the selection of any CPU frequency that could lead to latency requirement violations for any hosted microservice at the edge node e_k. It is crucial to note that all CPU frequencies at or above this global lowest frequency are considered valid (*i.e.,* supported by the edge node) and are candidates for selection by the GAS, as all edge nodes are considered homogeneous, thereby supporting the same set of CPU frequencies.

We define $\mathcal{T}ELP_m^k$ as the aggregate energy loss per second of all instances hosted on edge e_k for a particular CPU frequency f_m:

$$\mathcal{T}ELP_m^k = \sum_{I_q^j \in \Phi_k} (ELP_m^j \times V_q^j) \tag{1.11}$$

Where $ELP_m^j \times V_q^j$ represents the energy loss per second for an active instance (with waiting requests), while idle instances (with no requests) do not contribute to $\mathcal{T}ELP$. We derive the set of potential optimal frequencies for edge e_k (Line 3 Alg 1) Alg 1) as the feasible CPU frequency that results in the lowest total energy loss:

$$\widehat{F_k} = \{f_m \mid \mathcal{T}ELP_m^k = \min\{\mathcal{T}ELP_{m'}^k \mid \forall f_{m'} \in F, f_{m'} \geq \mathcal{G}W_k\}\} \tag{1.12}$$

Our policy favors the lowest frequency in scenarios where various CPU frequencies produce the lowest energy consumption (Line 4 Alg 1). This preference minimizes power consumption and reduces thermal output, contributing to further energy savings needed for cooling and extending the hardware lifespan. Thus, the optimal CPU frequency is given by:

$$\sigma_k = \min\{\widehat{F_k}\} \tag{1.13}$$

Once the optimal CPU frequency is identified, the *CPU frequency scaling policy* directs the *CPU Frequency Controller*—the operating system's interface for CPU frequency configuration—to adjust the frequency to the identified optimal level (Line 5 Alg 1), as depicted in step B3 of Fig. 1.2.

Algorithm 1: Global Optimal Frequency Selection (*GOFS*).

Input: e_k: the k-th edge node
Global Var: F: set of valid CPU frequencies, σ_k: the optimal frequency of the edge e_k

1 **Procedure** *GOFS*(e_k)*:*
2 $\quad \mathcal{G}W_k = \max\{W_q^p \mid I_q^p \in \Phi_k\}$;
3 $\quad \widehat{F_k} = \left\{f_m \mid \mathcal{T}ELP_m^k = \min\left\{\mathcal{T}ELP_{m'}^{k'} \mid \forall f_{m'} \in F, f_{m'} \geq \mathcal{G}W_k\right\}\right\}$;
4 $\quad \sigma_k = \min\{\widehat{F_k}\}$;
5 $\quad Set\_CPU\_Freq(\sigma_k)$;
6 **end**

1.5.2.2 Acceptable Frequency Adjustment (AFA)

AFA is a process that enables GAS to continuously update the minimum CPU frequency required to maintain latency requirements for each deployed instance independently. For each instance deployed at the edge node e_k, GAS CPU frequency scaling policy uses an independent *AFA* run on the background to monitor the instance's load (i.e., the number of queued requests) and take the corresponding actions. This *AFA* process is responsible for managing the instance throughout its lifecycle.

The key idea behind adjusting the CPU frequency within GAS is to use two thresholds for each instance. When the request count for an instance exceeds the top threshold, it triggers an increase in the CPU frequency. Conversely, when the request count falls below the bottom threshold, it requests a decrease in the CPU frequency. This mechanism optimizes *GAS* overhead, allowing the system to react only when necessary rather than at every request arrival or completion.

- **Top Threshold**: The top threshold indicates the throughput associated with the current CPU frequency f_κ for a specified edge e_i and microservice instance I_q^j. Represented as Γ_q^j, it is defined as:

$$\Gamma_q^j = \{\Upsilon_\kappa^j \mid \gamma_\kappa^i = 1\} \tag{1.14}$$

Algorithm 2: Acceptable Frequency Adjustment *(AFA)*

Input: I_q^j: instance to supervise, e_k: the edge node hosting the instance I_q^j
Global Var: ς_k: The current CPU frequency of the edge node e_k, $\mathcal{GW}_k$: the global low acceptable frequency of the edge e_k

Procedure *AFA*(e_k, I_q^j):
  $B_q^j \leftarrow 0$; $\Gamma_q^j \leftarrow 0$;
  $cf \leftarrow \varsigma_k$; $Lock \leftarrow$ False;
  $Idle \leftarrow$ True;
  while *(Instance_Is_Alive(I_q^j))* **do**
    $RPI_q^j \leftarrow RPI\_supervisor(Q_q^j)$;
    if *($RPI_q^j = \Upsilon_{maxF}^j$ or $P(\xi_q^j) \geq TDP$)* **then**
      $Lock \leftarrow$ True;
      Continue;
    end
    if *($RPI_q^j < \Upsilon_{maxF}^j$ and $P(\xi_q^j) < TDP$ and $Lock$)* **then**
      $Lock \leftarrow$ False;
    end
    if *($RPI_q^j = 0$)* **then**
      $B_q^j \leftarrow 0$; $\Gamma_q^j \leftarrow 0$;
      $W_q^j = f_{minF}$;
      $Idle \leftarrow$ True;
      $GOFS(e_k)$;
    end
    if *($RPI_q^j > \Gamma_q^j$)* **then**
      if *($Idle$)* **then**
        $Idle \leftarrow$ False;
        $Update(B_q^j); Update(\Gamma_q^j)$;
      end
      `// the term` $(t_{now} - \min\{a_p | r_p \in Q_q^j\})$ `refer to the longest waiting time of a request for the instance` I_q^j
      $\widetilde{F}_q^j = \{f_m \mid \forall f_m \in F, \Upsilon_m^j - \frac{t_{now} - \min\{a_p | r_p \in Q_q^j\}}{\chi_m^j} \geq RPI_q^j\}$;
      $W_q^j = \min\{\widetilde{F}_q^j\}$;
      $GOFS(e_k)$;
      Continue;
    end
    if *($RPI_q^j \leq B_q^j$)* **then**
      $\widetilde{F}_q^j = \{f_m \mid \forall f_m \in F, \Upsilon_m^j - \frac{t_{now} - \min\{a_p | r_p \in Q_q^j\}}{\chi_m^j} \geq RPI_q^j\}$;
      $W_q^j = \min\{\widetilde{F}_q^j\}$;
      if *($W_q^j = \mathcal{GW}_k$)* **then**
        $GOFS(e_k)$;
      end
      Continue;
    end
    if *($\varsigma_k \neq cf$ and not($Idle$))* **then**
      $Update(B_q^j)$;
      $Update(\Gamma_q^j)$;
      $cf \leftarrow \varsigma_k$;
    end
  end
end

- **Bottom Threshold**: The bottom threshold specifies the number of requests that can be processed with the preceding CPU frequency $f_{\kappa-1}$ without compromising the latency requirements. It is expressed as:

$$\mathcal{B}_q^j = \{\Upsilon_{\kappa-1}^j - \frac{t_{now} - \min\{a_p \mid r_p \in Q_q^j\}}{\chi_{(\kappa-1)}^j} \mid \gamma_\kappa^i = 1\} \tag{1.15}$$

where $\Upsilon_{\kappa-1}^j$ is the throughput associated with the preceding CPU frequency $f_{\kappa-1}$. Note that the term $t_{now} - \min\{a_p \mid r_p \in Q_q^j\}$ refers to the longest waiting time of a request for the instance I_q^j.

As detailed in Algorithm 2, when a new instance is created on the cluster, an *AFA* process is created and assigned to that instance to supervise and manage the minimum required CPU frequency. Initially, the *AFA* sets both thresholds to zero, considering the instance idle.

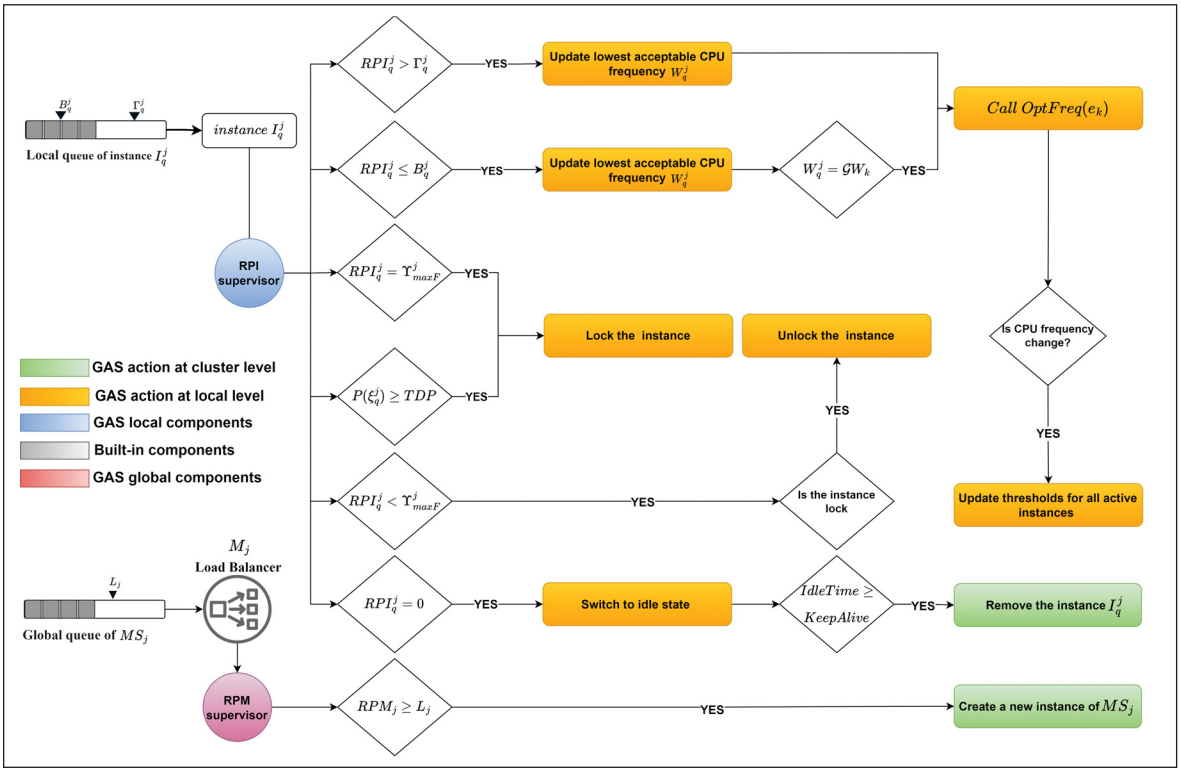


Figure 1.3 *GAS* action workflow at local and cluster levels.

At the initialization, *AFA* retains the current CPU frequency of the hosting edge node (see lines 1-4). While the instance is alive, AFA monitors the instance queue via *RPI supervisor* (see Fig. 1.2) to detect any changes in the number of requests in the queue RPI_q^j (line 6 corresponding to step B1 Fig. 1.2). Based on the observed fluctuations in RPI_q^j, *GAS* executes the corresponding actions. Fig. II-2 depicts a workflow summarizing GAS actions at the local and cluster levels. To ensure that instances' pending requests do not exceed the capacity of an

edge node (i.e. maximum requests without compromising latency, defined as peak throughput at the highest CPU frequency and expressed as Υ^{j}_{maxF}), *AFA* can lock the instance queue, temporarily rejecting incoming requests until the queue is unlocked (lines 7-13).

GAS considers two key scenarios to lock the instance queue:

1. When power consumption of the edge node hosting the instance hits or exceeds its TDP, enabling *GAS* to mitigate the risk of performance degradation due to CPU frequency reduction resulting from the power-throttling process.

2. When RPI^{j}_{q} has reached the edge node's maximum processing capacity Υ^{j}_{maxF} (*e.g.,* see instance I^{k}_{1} in Fig. 1.2), thus preventing potential performance violations.

The load balancer refrains from forwarding requests to a locked instance. When no instance is available for incoming requests, as with microservice M_k in Fig. 1.2, requests are queued in the global queue until an instance becomes available, *GAS*, via the *RPM supervisor*, monitors the global queue and triggers the Autoscaler to create a new instance when microservice M_j reaches the threshold L_j (See step C1 in Fig. 1.2). The interaction between queue locking and autoscaling is therefore governed by the profiling information collected for each microservice. During profiling, *GAS* determines Υ^{j}_{m}, which represents the maximum number of requests that microservice M_j can process sequentially at frequency f_m without violating its latency requirement. In particular, Υ^{j}_{maxF} represents the maximum safe processing capacity of an instance when the edge node operates at the highest supported CPU frequency. Once RPI^{j}_{q} reaches Υ^{j}_{maxF}, the capacity of the instance cannot be further increased through frequency scaling. Therefore, assigning additional requests to this instance would risk latency violations, and *GAS* locks its local queue.

When an instance is locked, the load balancer redirects incoming requests to other unlocked instances of the same microservice. If all instances of M_j are locked, this indicates that frequency-based adaptation is no longer sufficient to absorb the incoming load. However, *GAS* does not immediately create a new instance for each blocked request. Instead, blocked requests are temporarily placed in the global queue, allowing existing instances to become available again and avoiding unnecessary instance creation. The autoscaler is triggered only when the global queue reaches the threshold L_j, indicating that the accumulated waiting requests can no longer be safely delayed without risking latency violations. At this point, a new instance is created and the queued requests are forwarded to it.

1.5.3 Edge Energy-Performance Monitor

It is essential to monitor the dynamic parameters of the system, ensuring that they are immediately accessible by the CPU frequency scaling policy (See step B2 Fig. 1.2). It employs a map that links each frequency with

its corresponding current energy loss. Furthermore, the monitor consistently tracks the low acceptable global frequency and the optimal frequency (Fig. 1.2 includes an illustration of the map's structure).

1.6 Evaluation Methodology

1.6.1 Experimental Setup

Experiments were performed on an Intel i7-8700 processor (see Table II-2 for testbed details). To ensure the integrity of the measurements, the memory frequency was set at 2.4 GHz, and the Turbo Boost feature was disabled to maintain a stable CPU frequency.

Disabling Turbo Boost also ensures that the evaluation does not rely on overclocked or opportunistic turbo frequencies. Turbo frequencies are hardware-managed and depend on thermal headroom, power limits, and the number of active CPU cores. Including such states would make the available frequency range workload- and temperature-dependent, which could reduce measurement reproducibility and complicate the interpretation of *GAS*'s frequency-selection decisions.

The swap space was also disabled, and the memory was limited to 1 GB. Microservices were executed on an isolated core using the *isolcpus* boot option to isolate the environment from external interference. For better isolation, we rely on the BenchExec framework (Beyer, Löwe & Wendler, 2019). BenchExec is an open-source benchmarking framework designed to ensure reliable benchmarking. It offers various isolation mechanisms at multiple levels, including the file system, memory, and last-level cache.

Table 1.3 Experimental testbed configuration.

Component	**Specification**	**Component**	**Specification**
CPU model	Intel i7-7700	L1 Cache	256KB
Sockets	1	L2 Cache	1024KB
CPU Frequency	800MHz - 3.6GHz	L3 Cache	8MB
Memory Frequency	2.4GHz	OS	Ubuntu 22.04
Logical Cores	8	Memory	16GB

1.6.2 Micorservice workload

To evaluate the performance of our solution, we use a diverse set of workloads sourced from the TACLe Benchmarks (Falk, Altmeyer, Hellinckx, Lisper, Puffitsch, Rochange, Schoeberl, Sørensen, Wägemann & Wegener, 2016), a well-established benchmark in real-time embedded systems. A summary of the selected workloads is detailed in Table 1.4. Workloads were categorized according to (El Khazen, Amor, Kougblenou, Gogonel & Cucu-Grosjean,

2023). This study examined the sensitivity of TACLe workloads to variations in memory frequency. Workloads were classified according to their sensitivity to memory frequency: high sensitivity as memory-bound, medium sensitivity as mixed, and low or no sensitivity as CPU-bound. This study treats the execution of each selected workload, repeated 300 times, as analogous to a microservice processing a single request. Subsequently, we execute 50 requests for each microservice at each valid CPU frequency. At each frequency, we record the execution time and the total energy consumption (including DRAM and package). Energy measurement is collected using the CPU Energy Meter (Beyer & Wendler), an Intel Running Average Power Limit (RAPL) based tool integrated with the BenchExec framework. After execution, 19.44% of the data outliers were removed. The collected data show reliable measurements proof for execution time (relative standard deviation: 0.02% to 0.05%) and energy consumption (relative standard deviation: 4.44% to 12.59%). The higher variability in energy consumption is attributed to factors such as thermal conditions and the challenge of completely isolating the environment. Consequently, performance-energy maps for each microservice were generated by averaging energy consumption and execution time across CPU frequencies. Fig. 1.4 presents the impact of CPU frequency on energy consumption for different microservices. To further approximate real-world conditions, we assign specific latency requirements to each microservice, indicating the time frame within which each request must be processed.

Table 1.4 Microservices description

Microservice	**Description**	**Category**
quicksort	Quick sort of vectors	CPU-bounded
matrix	Matrix multiplication	CPU-bounded
petrinet	Petri net simulation	Memory-bounded
ndes	Complex embedded code	Memory-bounded
epic	Pyramid image coder	Mixed workload

Furthermore, we used a variant invocation rate for request arrivals, categorized as high, low, and random, to evaluate *GAS*'s performance in different scenarios. We detail them as the following:

1. **High Rate**: The number of requests falls within the range of 70% to 100% of the microservice's peak throughput at the highest CPU frequency.
2. **Low Rate**: The number of requests is between 1% and 30% of the maximum throughput.
3. **Random Rate**: The number of requests can vary from the minimum to the maximum throughput.

To thoroughly assess the energy efficiency and performance of *GAS* under various load conditions, the evaluation proceeds through a series of rounds, increasing from 1 to 100. In this setup, each round must be fully completed,

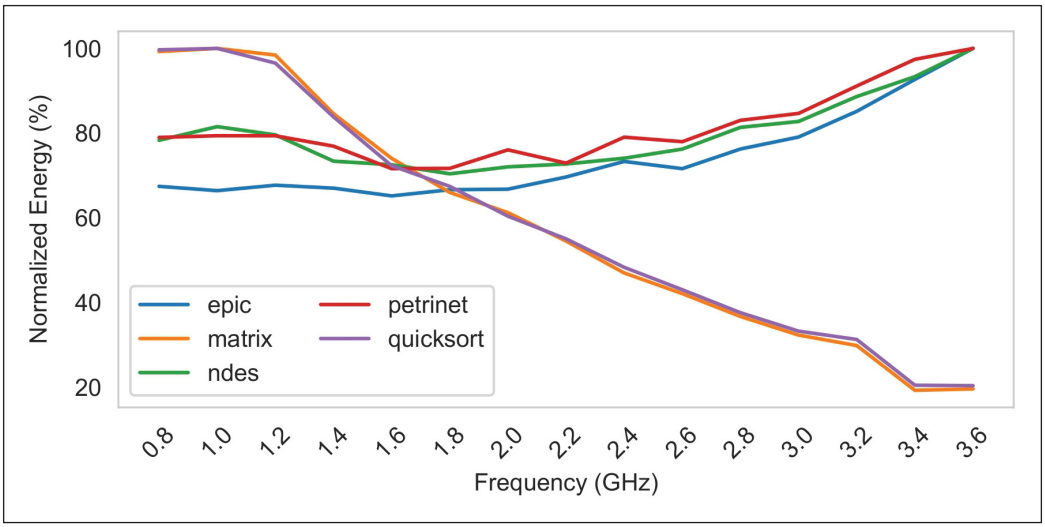


Figure 1.4 The impact of CPU frequency on the energy consumption of each microservice.

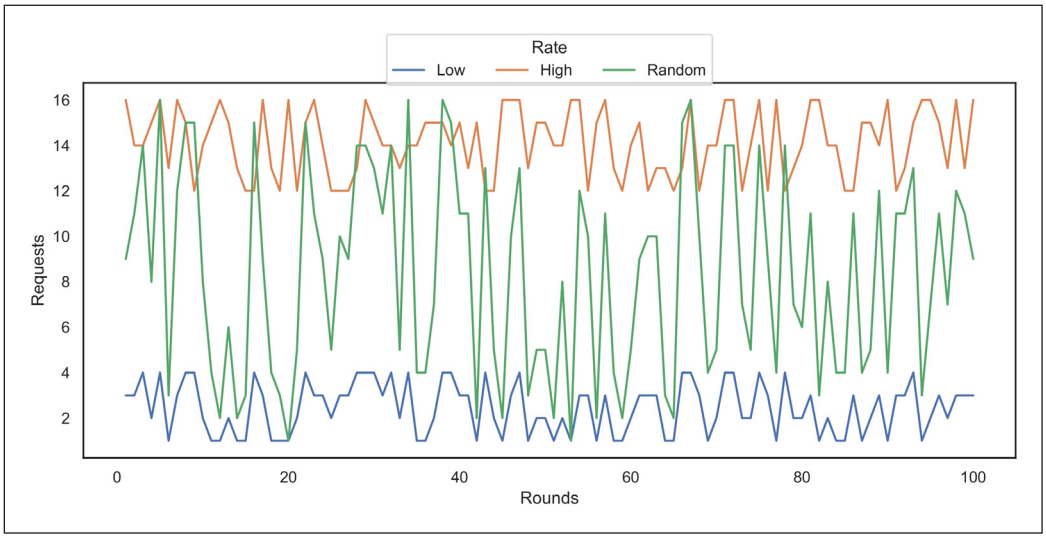


Figure 1.5 The number of requests per round for the 'petrinet' microservice under different rates.

with all requests processed, before advancing to the next. The size of this batch falls within the lower and upper ranges of the selected invocation rate. We utilize a uniform random generator to pick a number within this range. Fig. 1.5 displays the varying number of requests per round for the 'petrinet' microservice under different rates. Notably, the peak throughput of the 'petrinet' microservice at the highest CPU frequency is 16 requests. The high-rate setting is used to emulate bounded bursty arrivals. The upper bound of this burst is not selected arbitrarily; it is derived from the profiled peak throughput of each microservice at the highest supported CPU frequency. In other words, the 100% load point corresponds to the maximum number of requests that the microservice can process at f_{maxF} while satisfying its latency requirement. Therefore, the evaluation assumes that, at the peak

profiled burst, the available local processing capacity is sufficient when the edge node operates at the maximum CPU frequency. Bursts exceeding this profiled maximum capacity are treated as overload conditions that require the global queue and autoscaling mechanism rather than frequency scaling alone.

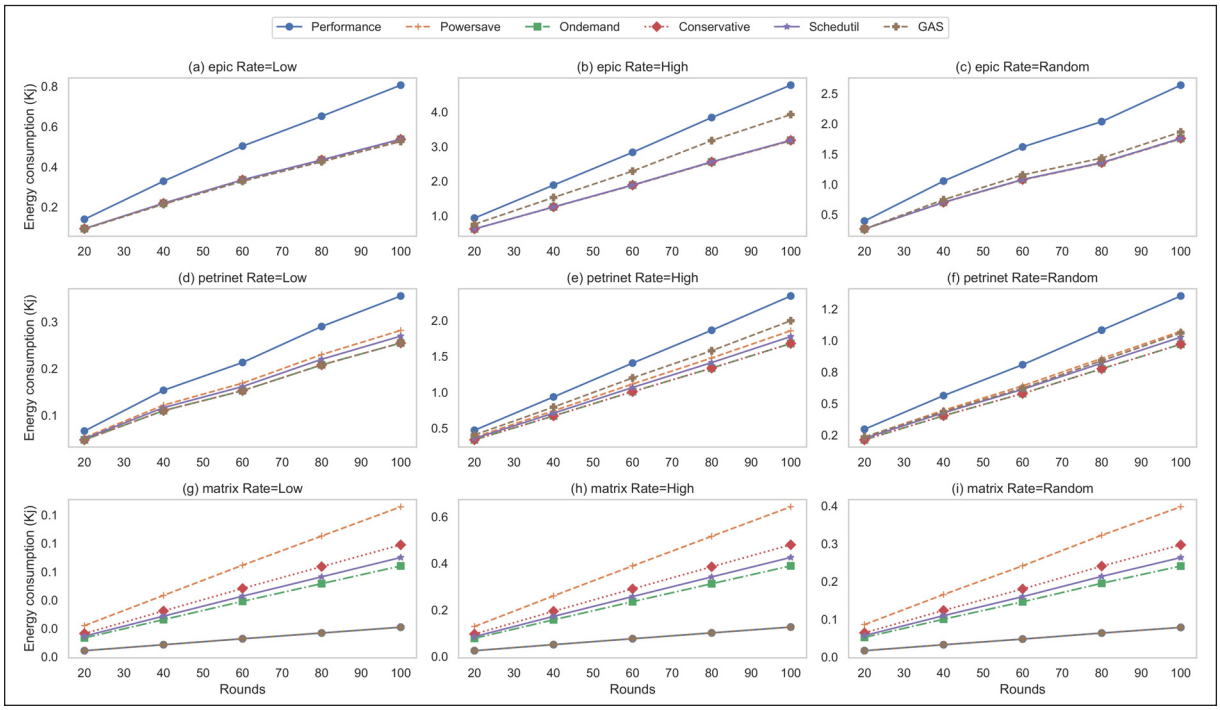


Figure 1.6 Energy consumption of different microservices under different power governors and different invocation rates

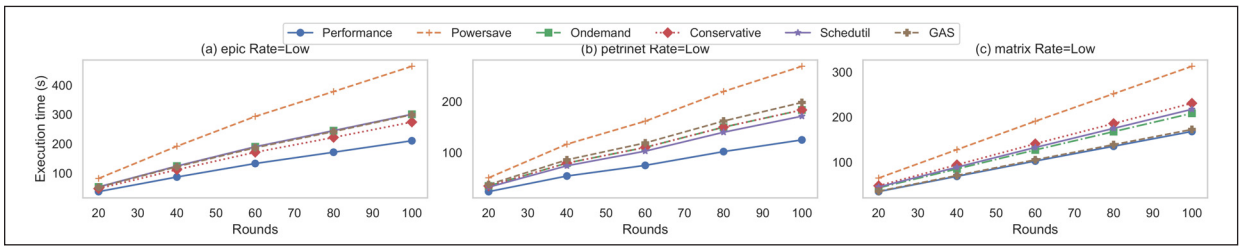
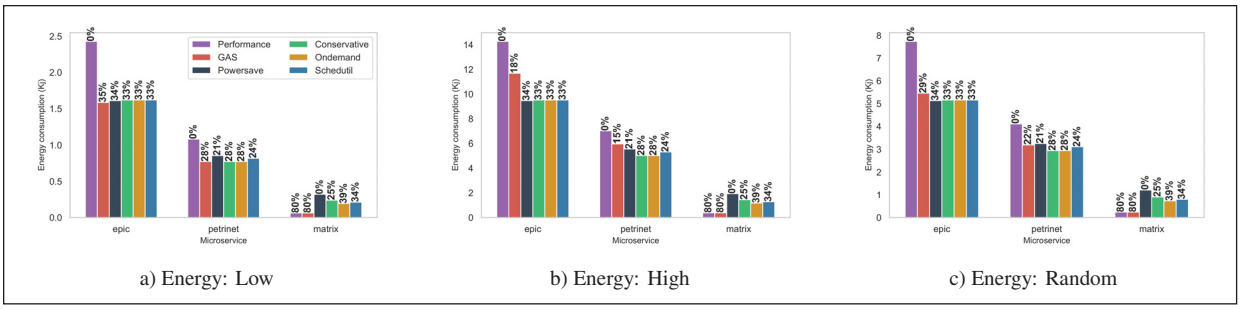


Figure 1.7 Execution time of different microservices under different power governors and different invocation rates

1.7 Performance Evaluation

The implementation of *GAS* frequency scaling policy can be achieved by setting the userspace governor with the Advanced Configuration and Power Interface (ACPI) CPUfreq scaling driver. To provide a comprehensive



a) Energy: Low b) Energy: High c) Energy: Random

Figure 1.8 Overall energy consumption of different power governors at various invocation rates.

performance evaluation, we evaluated our policy against different built-in power governors from the same ACPI scaling driver, such as (1) performance, (2) powersave, (3) conservative, (4) schedutil, and (5) ondemand.

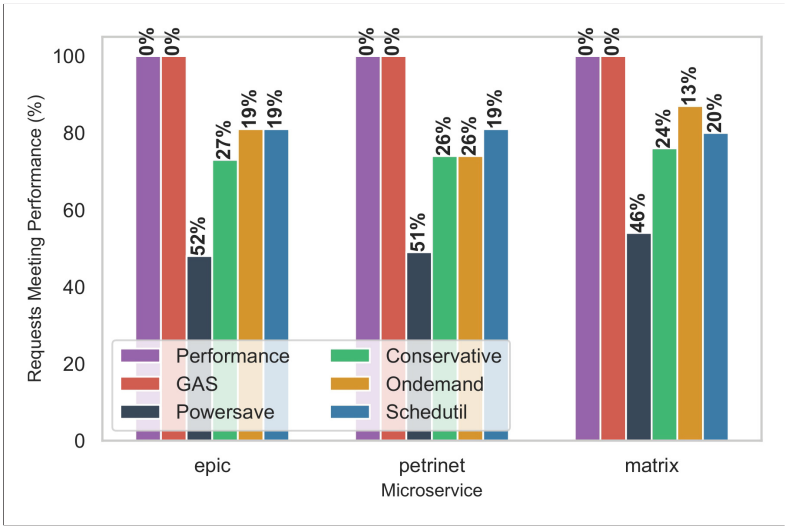
Throughout our evaluation, we use several performance indicators:

- **Energy Consumption:** The total energy consumption, measured in Kilojoules (Kj).
- **Execution Time:** The total execution time, measured in seconds.
- **Performance:** The percentage of requests processed within the required latency, indicating adherence to latency requirements. In this evaluation, latency is treated as a hard per-request constraint rather than a percentile-based metric such as P95. Therefore, each request is counted as successful only if its individual response time does not exceed the latency requirement S_j.
- **Energy gain:** The reduction in energy consumption relative to the 'performance' governor, expressed as a percentage.
- **Slowdown:** The increase in total execution time relative to the 'performance' governor.

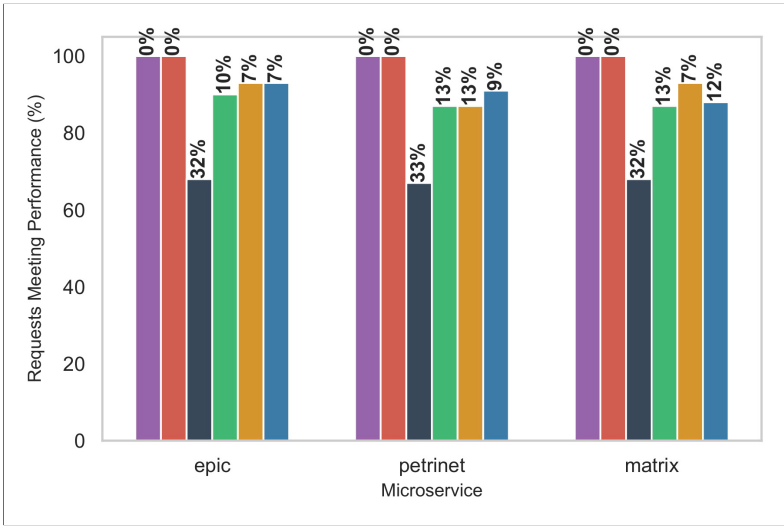
It is important to note that the evaluation does not investigate the effects of auto-scaling on maintaining the performance. The evaluation is performed in two scenarios. The first is under a single workload type, while the second is under a combination of workload types.

1.7.1 Scenario 1 (Single-workload evaluation)

In this scenario, we aim to investigate how our solution performs with each type of workload. To do so, we evaluated our solution with 'epic', 'matrix', and 'petrinet' microservices, each representing a distinct workload type. We calculate the energy consumption and execution time under various power governors at different invocation rates, as shown in Figs. 1.6 and 1.7, respectively. Our solution shows high adaptability with the different types of workloads, especially at low invocation rates where CPU frequency choice is influenced solely by workload nature since there



a) Performance: High



b) Performance: Random

Figure 1.9 Overall performance of different power governors at various invocation rates.

is less pressure to meet latency requirements. For 'epic' in Fig. 1.6(a), energy use is similar to other governors except for 'performance', which uses more energy at maximum CPU frequency. Compared to the performance governor, our solution achieves an overall energy reduction of 35%, slightly outperforming the 'powersave' at 34% and other governors by 33%.

Fig. 1.8a shows the overall energy consumption for each governor at low rates for different microservices. Although similar in energy use, significant differences in execution time are observable in Fig. 1.7(a), especially with the 'powersave' governor, which has resulted in longer execution times due to the usage of lower frequency, increasing total energy. The 'performance' governor achieves always the minimal execution time by leveraging the maximum

CPU frequency. Our solution slows down the overall execution time by 41% compared to the 'performance' governor, similar to 'ondemand' and 'schedutil' governors at 42%, while the 'powersave' governor extends it by 120%. A noteworthy observation is the behavior of the 'conservative' governor, which slowed down the overall execution by 29%. This is due to its decision to operate at 1.8 GHz compared to the 1.6 GHz frequency of our solution. This minor frequency variation endows our solution with an additional 2% energy savings advantage. In memory-intensive workloads like the 'petrinet' microservice, *GAS* and the 'conservative' governor significantly reduce energy consumption by 28% compared to the 'performance' governor, ondemand, and schedutil, which achieve 21% and 24%, respectively, as shown in Fig. 1.8a. The 'powersave' mode, despite its low-power intent, results in the longest execution times, extending the execution time by 116% while offering minimal energy benefits. This highlights the counterproductiveness of this governor in terms of energy.

Our solution extends the execution time by 59%, while ondemand and conservative extends it by 47%, and 37% by the 'schedutil' governor. However, 'schedutil' comes at the cost of slightly higher energy consumption. This is attributed to the different CPU frequencies chosen i.e., 2 GHz for 'schedutil', 1.8 GHz for both 'ondemand' and 'conservative', and 1.6 GHz for *GAS*. Our policy selects the lowest possible frequency when multiple CPU frequencies result in equivalent minimal energy consumption. This decision indirectly contributes to improved energy efficiency. Lower CPU frequencies lead to reduced power usage, resulting in lower thermal output and, thus, reduced cooling costs. In CPU-bound microservice, our solution matches the 'performance' governor in energy efficiency, significantly outperforming other governors in both energy efficiency and execution time, as detailed in Fig. 1.6c and Fig. 1.7c. It reduces energy consumption by 80% and execution time by 55% compared to 'powersave' while 'ondemand', 'schedutil', and 'conservative' reduce energy consumption by 39%, 34%, and 23% and execution time by 33%, 30%, and 26%, respectively. It is important to note that at low invocation rates, all power governors meet the latency requirements for various microservices. To further investigate the effectiveness of *GAS*, we replicated the initial experiment under a random rate for typical real-world scenarios and a high rate for extreme conditions.

As depicted in Fig. 1.6d through Fig. 1.6g, built-in governors show a consistent energy consumption profile regardless of invocation rate variations. This stems from being agnostic to both workload characteristics and latency requirements. On the contrary, our solution shows high adaptability to varying invocation rates across diverse workloads. Fig. 1.8 and Fig. 1.9 present, respectively, the overall energy consumption and performance across 100 rounds for different power governors, highlighting their effectiveness at various rates. Under a random invocation rate with the 'epic' microservice, our solution successfully guarantees the latency requirements of all requests (see Fig. 1.9b) while achieving a 29% reduction in energy consumption compared to the 'performance' as indicated in Fig. 1.8b. Even at high invocation rates, our solution maintains latency requirements (see Fig. 1.9a) while achieving a reduction of 18% in energy consumption, as shown in Fig. 1.8b. While Other governors reduce energy consumption by up to 34% they yield significant performance violations. For example, under 'powersave',

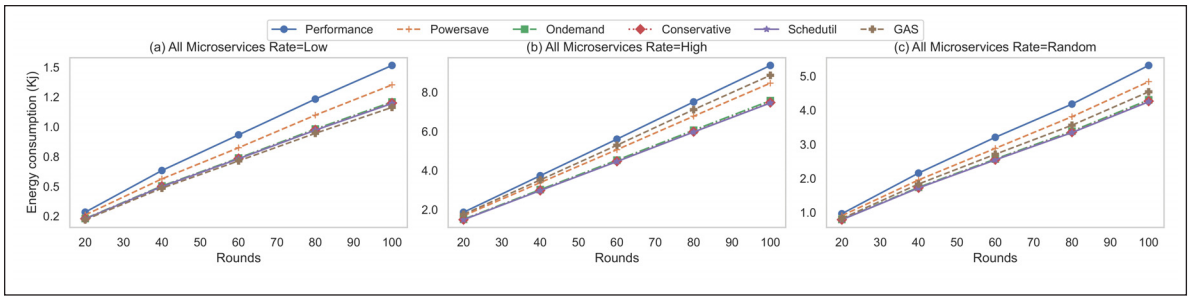


Figure 1.10 Energy consumption under different power governors with various invocation rates for multi microservices in concurrency.

32% of requests fail to meet latency requirements, increasing to 52% at high rates. Similar degradation is observed with 'conservative' (violations ranging from 10% to 27%), 'ondemand' (7% to 19%), and 'schedutil' (7% to 19%). In the case of the 'petrinet' microservice, our solution consistently adheres to all latency requirements at both random and high rates (see Fig. 1.9b for the random rate and Fig. 1.9a for the high rate), while consuming 22% less energy under random rates and 15% under the high rate, as shown in Fig. 1.8c and Fig. 1.8b respectively. Notably, the rate of latency requirements violations is higher in 'petrinet' compared to the 'epic' microservice in both rates. For instance, the "Proportional Rise"—defined as Energy Reduction divided by QoS Violation—is consistently higher for 'petrinet' than for 'epic' across different governors. This emphasizes the critical role of understanding workload characteristics in making informed CPU frequency decisions. Concerning the 'matrix' microservice, *GAS* matches the 'performance' governor, reducing energy use by 80% compared to the 'powersave.' Other governors, 'conservative', 'ondemand', and'schedutil', achieve energy reductions of 25%, 39%, and 34%, respectively. Notably, *GAS* and the 'performance' consistently fulfill all performance benchmarks across varying rates, a feat not reached by the other governors. Under 'powersave', 32% to 46% of requests fail to meet latency requirements at high rates. Similar issues are observed with 'conservative' (13% to 24% violations), 'ondemand' (7% to 13% violations), and 'schedutil' (12% to 20% violations). These results underscore that *GAS* is adaptable to various types of workloads and consistently maintains energy efficiency and latency requirements across different invocation rate scenarios. Although other governors may match *GAS*'s performance in isolated cases, they lack consistent adaptability, affirming our solution's versatility.

1.7.2 Scenario 2 (Multi-Workload Evaluation)

The second phase of the evaluation extends the scope to include real-world scenarios where multiple microservices with different workload characteristics are running concurrently. Fig. 1.10 and Fig. 1.11 illustrate the energy consumption and execution time under various power governors at different invocation rates, respectively. At a low invocation rate, all governors meet latency requirements. However, our solution outperforms other governors in energy efficiency and execution time. Specifically, it reduces energy consumption by 23%, compared to

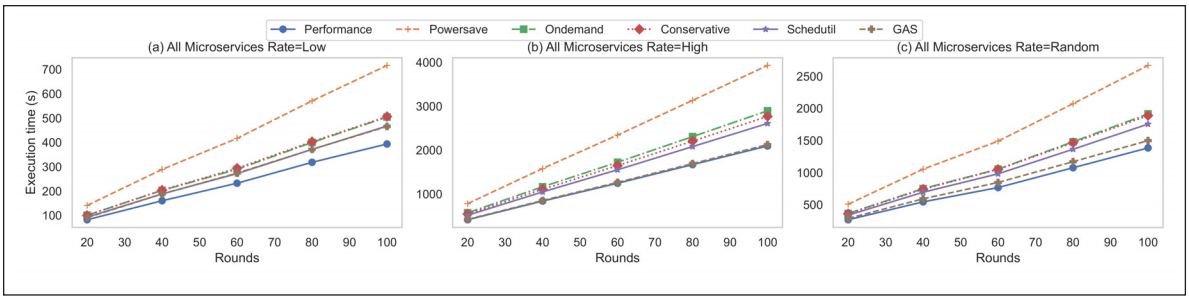
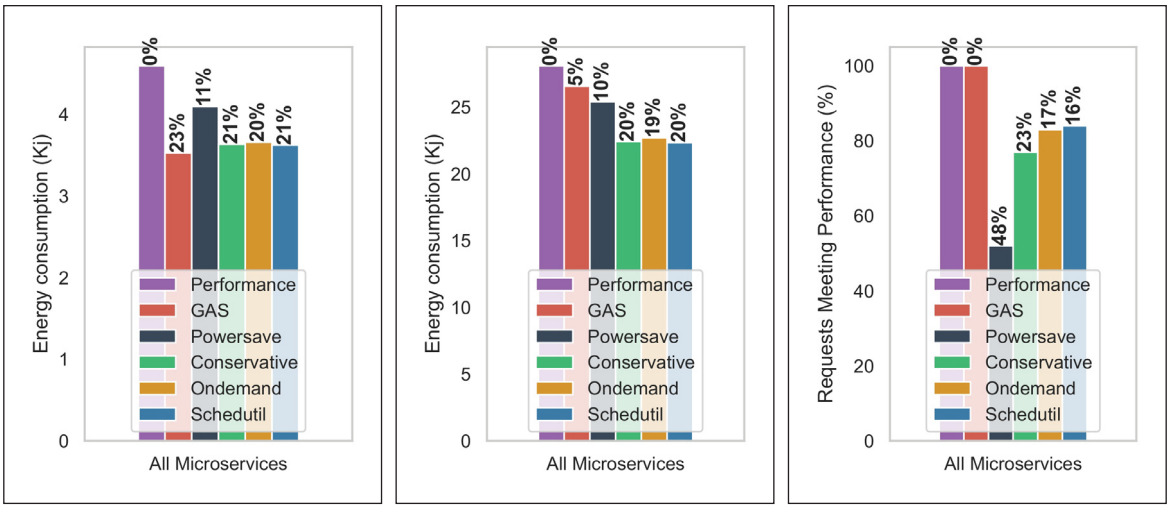


Figure 1.11 Execution time under different power governors with various invocation rates for multi microservices in concurrency.

11% by 'ondemand' and 20-21% by other governors, as illustrated in Fig. 1.12a. The plot in Fig. 1.12 presents the overall energy consumption and relative performance of different power governors under varying rates for multi-microservices running in concurrency. In terms of execution time, our solution achieves a 35% reduction relative to 'powersave', while 'conservative', 'ondemand', and 'schedutil' achieve reductions of 25%, 30%, and 35%, respectively, as shown in Fig. 1.11(a). These findings indicate that high energy consumption does not automatically translate to improved execution time, potentially leading to latency requirements violations even at higher energy costs. With random invocation rates, our solution consistently meets latency requirements while achieving a 15% reduction in energy consumption compared to the 'performance' governor (see Fig. 1.12d). Although 'conservative,' 'ondemand,' and 'schedutil' governors exhibit marginal gains in energy efficiency, surpassing *GAS* by up to 5% —they compromise on latency requirements. Specifically, these governors result in 11%, 8%, and 7% of request violations, respectively, as depicted in Fig. 1.12e. Such shortcomings are particularly intolerated in latency-sensitive applications, where adherence to latency constraints is non-negotiable. In contrast, the 'powersave' governor presents a less favorable trade-off between energy efficiency and performance reliability, incurring the highest rate of latency requirements violations at 29%, while only marginally reducing energy consumption by 9%.

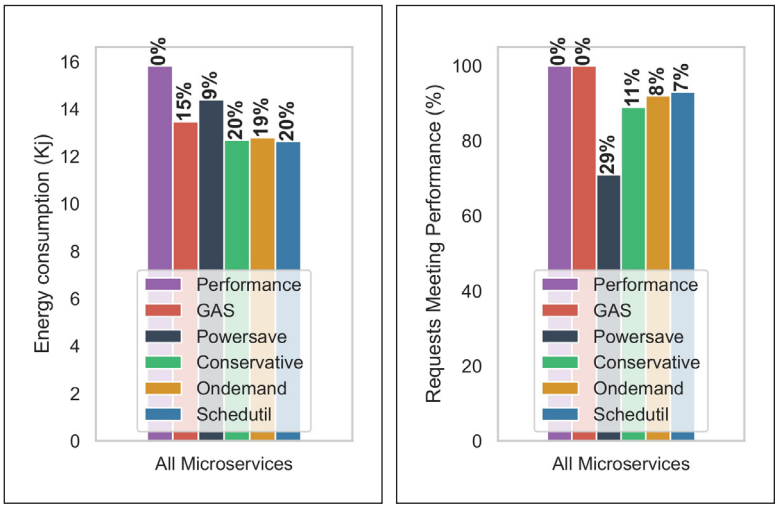
Under high invocation rates, our solution maintains its performance reliability, achieving a 5% energy reduction relative to the 'performance' governor. Although other governors exhibit the same energy efficiency patterns as in other rates (see Fig. 1.12b), they do so at the cost of high latency violations. Specifically, the 'powersave' governor results in 48% of requests failing to meet their latency requirements, followed by 23%, 17%, and 16% for 'conservative,' 'ondemand,' and 'schedutil,' respectively, as depicted in Fig. 1.12c. For deep investigation, we track the CPU frequency transactions of each one as depicted in Fig. 1.13 for random invocation rates (CPU frequency transactions trace for low and high rates available respectively via (Fre, a) and (Fre, b)). Our solution is more dynamic, with a frequency selection range of 1.6 to 3.6 GHz. In contrast, the 'conservative', 'ondemand,' and 'schedutil' governors operate within a narrower range of [1.4 GHz - 2.0 GHz], [1.6 GHz - 1.8 GHz], and [1.8 GHz - 2.0 GHz], respectively. These governors, which adjust frequency based solely on CPU usage, struggle when latency-sensitive microservices process requests sequentially due to their lack of awareness of queued requests and



a) Energy consumption at low invocation rates

b) Energy consumption at high invocation rates

c) Performance at high invocation rates



d) Energy consumption at random invocation rates

e) Performance at random invocation rates

Figure 1.12 Overall energy consumption and performance of different power governors at various invocation rates for concurrent multi-microservice workloads (percentages atop each bar indicate the reduction relative to the maximum).

the latency requirements of each microservice. Table 1.14 provides a consolidated summary of the key findings of Scenario 2. In contrast, our policy considers the nature of workloads, performance targets, and current microservice loads (*i.e.,* queued requests), resulting in a 15% reduction in energy costs with a 9% increase in execution time relative to the performance mode, showcasing its ability to effectively balance energy efficiency with performance without compromising latency requirements. Even under high invocation rates, it maintains a 5% energy reduction with only a 2% increase in execution time. Even in scenarios where latency is easy to achieve, as in the low rate, these generic dynamic governors do not effectively optimize execution times in proportion to energy savings. Our findings show that while 'conservative' and 'ondemand' governors slowdowns the overall execution time by

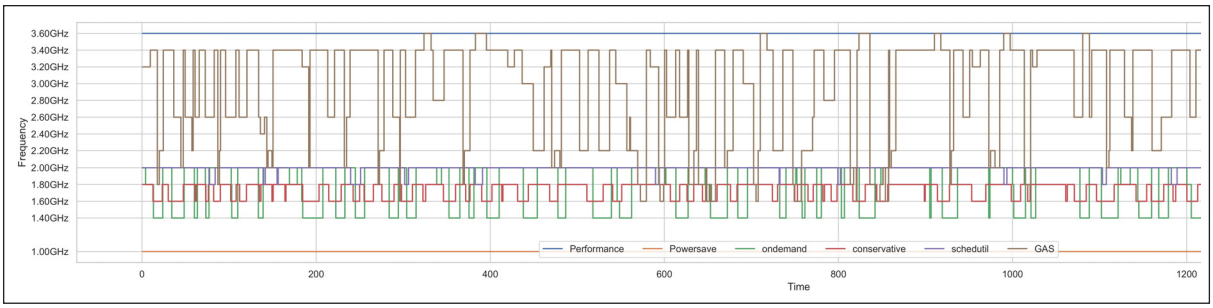


Figure 1.13 CPU frequency transactions under different governors for random invocation rate

Rate	Power governor	Execution time (s)	Performace (%)	Energy (Kj)	Energy Gain (%)	Slowdown (%)
HIGH	Performance	6257.04	100	28.09	0	0
HIGH	Powersave	11759.32	52	25.38	10	88
HIGH	Ondemand	8657.6	83	22.70	19	38
HIGH	Conservative	8267.48	77	22.41	20	32
HIGH	Schedutil	7796.9	84	22.34	20	25
HIGH	GAS	6353.11	100	26.56	5	2
RANDOM	Performance	4044.16	100	15.82	0	0
RANDOM	Powersave	7797.75	71	14.39	9	93
RANDOM	Ondemand	5568.46	92	12.79	19	38
RANDOM	Conservative	5509.9	89	12.69	20	36
RANDOM	Schedutil	5137.76	93	12.64	20	27
RANDOM	GAS	4400.16	100	13.46	15	9
LOW	Performance	1186.56	100	4.60	0	0
LOW	Powersave	2134.51	100	4.10	11	80
LOW	Ondemand	1497.7	100	3.66	20	26
LOW	Conservative	1507.07	100	3.63	21	27
LOW	Schedutil	1391.76	100	3.62	21	17
LOW	GAS	1389.89	100	3.53	23	17

Figure 1.14 A summary of key findings from Scenario 2

26-27%, and 'schedutil' by 17 %, they achieve a 20-21% reduction in energy usage, resulting in a low energy-time efficiency ratio (*i.e.,* energy savings over time extension) compared to our solution. Compared to other governors, *GAS* maintains performance reliability with low energy cost, even under a high invocation rate.

1.8 Discussion & Future work

Our finding as depicted in Table 1.14 shows that our solution reduces energy consumption at the cost of slightly increasing overall completion time compared to the performance governor, which operates at maximum CPU frequency. Specifically, the completion time increases by 2% at high invocation rates, 9% at random invocation rates, and 17% at low invocation rates. Despite these increases, all requests were processed within the required latency constraints. Our primary objective is to ensure no request misses its deadline while minimizing energy consumption.

Our solution achieves this by adjusting CPU frequency to maintain latency requirements and optimize energy usage, reducing energy consumption by 5%, 15%, and 23% at high, random, and low invocation rates, respectively. These results demonstrate that our approach effectively balances energy savings with performance reliability, which is crucial for latency-sensitive applications. By prioritizing latency requirements, we achieve energy efficiency without compromising deadline adherence. This confirms the effectiveness of our approach in edge computing, especially for latency-sensitive microservices. The results also highlight the limitations of conventional power governors, which fail to strike an optimal balance between performance and energy efficiency, especially in scenarios that require consistent performance reliability. Nevertheless, several limitations need to be addressed as follows:

1.8.1 Validity Concerns

Energy consumption is measured using the Intel RAPL interface, which ties the reliability of *GAS* to the reliability of this interface. Moreover, performance isolation in environments with concurrent microservices is challenging due to resource competition, resulting in unpredictable performance and energy outcomes (Sampaio & Barbosa, 2016). We use BenchExec for isolation during profiling, but complete isolation remains a challenge. In our future work, we plan to integrate advanced cache isolation strategies such as the one presented in (Novaković, Vasić, Novaković, Kostić & Bianchini, 2013) into our profiling mechanism.

1.8.2 Heterogeneous edge nodes

The heterogeneity of edge computing nodes complicates performance and energy profiling due to diverse hardware and software characteristics. One solution is to use scalable machine learning models, as mentioned in (Fieni, Rouvoy & Seiturier, 2021), which could estimate the energy performance of microservices based on the hardware characteristics and workload, reducing operational overhead and speeding up deployment.

1.8.2.1 Frequency transition latency

Frequency transition latency (*i.e.,* the duration required to change CPU frequency) is crucial in meeting the latency requirements of microservices. This latency depends on the hardware characteristics; it can range from tens to hundreds of microseconds (Mazouz, Laurent, Pradelle & Jalby, 2014), affecting performance and energy consumption (Gouicem, Carver, Lozi, Sopena, Lepers, Zwaenepoel, Palix, Lawall & Muller, 2020). To mitigate this, we plan to integrate request forecasting techniques such as (Wang, Zhu, Li, Jiang, Ramakrishnan, Zheng, Yan, Zhang & Liu, 2022b) into *GAS*. This will enable *GAS* to proactively adjust CPU frequencies, initiating frequency transitions only when potential energy gains are significant.

1.8.2.2 Limitations of current evaluation

In this work, we focus on evaluating our CPU frequency scaling policy. In future work, we plan to conduct a comprehensive evaluation of *GAS*, where we will extend the evaluation to investigate the impacts exerted by the instance queue lock mechanism and the auto-scaling policy on both performance and energy efficiency. Also, a comparative assessment against Kubernetes' default strategy will offer a more comprehensive understanding of the *GAS* framework's effectiveness.

In addition, the current evaluation focuses on bounded burst scenarios, where the burst size does not exceed the profiled peak throughput at the highest CPU frequency. Future evaluations will consider stronger overload bursts that exceed this local capacity and will quantify how the global queue, queue-locking mechanism, and autoscaler jointly affect latency preservation under such conditions.

1.8.2.3 Microservices with Variable Workloads

Our proposed approach is designed to effectively manage microservices with stable workloads, where the workload to process each request is nearly uniform. This stability allows for predictable performance and efficient resource allocation. However, in real-world scenarios, many microservices exhibit variable workloads, where the processing demands can fluctuate significantly across different requests. To extend our solution's applicability to microservices with variable workloads, we plan to enhance our approach by integrating a machine learning-based model. This model will dynamically predict processing times and energy requirements based on the characteristics of the input data.

1.8.2.4 progressive queue-admission score

Another possible extension is to replace the current binary queue-locking decision with a progressive admission-control mechanism. Instead of marking an instance only as locked or unlocked, *GAS* could assign each instance a queue-admission score that reflects its remaining safe processing capacity. For example, this score could be defined according to the distance between the current local queue size RPI_q^j and the maximum safe capacity Υ_{maxF}^j. Instances with larger remaining capacity would be assigned higher routing priority, while instances approaching Υ_{maxF}^j would progressively receive lower priority. The load balancer could then favor instances with higher scores and apply hard queue locking only as a final safeguard when an instance reaches its maximum safe capacity. This would make request routing smoother and reduce the potential disruptiveness of binary queue locking.

1.8.2.5 Turbo Frequencies and Undervolting

The current implementation of *GAS* does not consider overclocking, turbo frequencies, or undervolting. *GAS* operates only within the processor's officially supported non-turbo DVFS range, while voltage selection is implicitly handled by the platform. This design choice improves stability and reproducibility and avoids the risk of sudden power or thermal throttling under aggressive operating points. Nevertheless, turbo frequencies may offer opportunities for CPU-bound workloads under a race-to-idle strategy, where requests are executed faster and the processor can return to a lower-power state sooner. Exploiting this behavior would require a hardware-aware extension of *GAS* that accounts for active-core count, available thermal headroom, TDP limits, and the maximum turbo frequency permitted by the processor under each runtime condition. Similarly, undervolting could reduce power at a given frequency, but it requires platform-specific voltage-margin analysis and stability validation. We therefore leave the safe integration of turbo-aware and undervolting-aware control policies as future work.

1.9 Conclusion

In this paper, we develop *GAS* a CPU frequency scaling policy tailored for microservice edge computing environments. Our approach is particularly beneficial for latency-sensitive applications deployed on less sophisticated edge devices, especially when these devices opt for ship-wide DVFS controllers that require all CPU cores to operate at the same frequency. Our policy dynamically adjusts the CPU frequency based on the queued requests and the latency requirements of each hosted microservice. An initial profiling phase is essential, during which each microservice is executed at various frequencies to collect performance-energy metrics. The experiments were carried out using BenchExec, and the findings demonstrated that our approach successfully maintains the desired latency across diverse scenarios, reducing energy consumption by 5% to 23% when multiple microservices are running concurrently and by up to 80% when a single microservice is deployed. Additionally, to bolster performance reliability, our solution is designed to seamlessly integrate with built-in orchestration tools. In the future, we aim to extend our policy to support heterogeneous edge nodes with more focus on Per-core DVFS. Finally, we believe that our approach is a pivotal step toward achieving sustainability in edge computing.

CHAPTER 2

INVESTIGATING THE POTENTIAL OF KEPLER TOWARDS POWER OBSERVABILITY FOR SUSTAINABLE CLOUD COMPUTING

Zouhir Bellal, Laaziz Lahlou, Nadjia Kara, Timothy Murphy, Tan Phat Nguyen, Arif Ahmed and Mario Perez-Jimenez

Department of Software Engineering and Information Technology at École de technologie supérieure,

1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Article published in the journal

IEEE Transactions on Green Communications and Networking

on FEBRUARY 3, 2026

Abstract: Power monitoring is a cornerstone of sustainability efforts, especially as the energy demands of Artificial Intelligence (AI) workloads continue to rise. However, accurately measuring power consumption in cloud-native environments remains challenging due to technical constraints such as fine-grained monitoring requirements and the high abstraction introduced by virtualization and containerization. Kepler (Kubernetes-based Efficient Power Level Exporter), an open-source tool for container-level power monitoring, has emerged as a promising solution for cloud power observability. This paper outlines the key requirements for accurate power tracking in containerized environments and assesses Kepler's alignment with these criteria. However, its precision remains unvalidated, mainly due to the lack of a systematic evaluation methodology. To fill this gap, we introduce a novel accuracy validation framework tailored to assess container-level power monitoring tools under dynamic controlled multi-tenancy environments, including CPU frequency scaling, C-state transitions, and varying co-runner workloads (*i.e.,* co-hosted containers executing concurrently on other cores of the same processor socket). Using this framework, we perform the first in-depth evaluation of Kepler's accuracy in real-world cloud scenarios (e.g., dynamic power configuration settings, dynamic workloads). Our results show that Kepler's container-level power estimation exhibits a root mean squared error (RMSE) of 11.9 Watts against the RAPL ground truth, corresponding to an overestimation of approximately 15x. Its accuracy is highly sensitive to runtime factors such as CPU configuration and C-state transitions, which reveals critical limitations in Kepler's current power model and highlights the need for refinement. This work establishes the foundation for more precise and effective power observability in cloud computing and paves the way for sustainable cloud computing.

Keywords: Power monitoring, Cloud power observability, Container-level power monitoring, Power accuracy validation framework, Kepler accuracy validation

2.1 Introduction

By 2030, projections estimate that the IT sector's electricity demand could rise to 3,200 TWh (i scoop). However, these estimates may be conservative due to the rapid expansion of energy-intensive Artificial Intelligence (AI) workloads, particularly generative AI. For example, a single query to ChatGPT consumes 25 times more energy than a Google search(Saenko, a, b). A recent study predicts that the global energy capacity required to sustain AI growth will reach 68 GW by 2027 and 327 GW by 2030 (Pilz, Mahmood & Heim, 2025). Consequently, energy efficiency has emerged as a critical Key Performance Indicator (KPI) for cloud providers, which form the backbone of both cloud and AI services. However, achieving energy efficiency in modern cloud-native environments presents unique technical challenges.

Containerization and orchestration technologies such as Kubernetes (K8s) introduce hardware abstraction layers that obscure fine-grained resource usage, making container-level power monitoring difficult (Bellal *et al.*, 2025c). This lack of visibility complicates the development of container-level power monitoring tools, which are essential for supporting sustainable and energy-aware cloud computing (Fieni *et al.*, 2020).

Several tools have been proposed to address this gap, including Scaphandre (Hubblo), SmartWatts (Fieni *et al.*, 2020), and Kepler (kepler CNCF). Among them, Kepler has emerged as a leading solution for container-level power monitoring. Powered by leading organizations such as IBM and Red Hat, and backed by Cloud Native Computing Foundation (CNCF), Kepler becomes a cornerstone of sustainability initiatives, such as the Power Efficiency Aware Kubernetes Scheduler (PEAKS)(computing, b) and the Container-Level Energy-Efficient Vertical Pod Autoscaler (CLEVER) (computing, a). Despite its potential, Kepler's accuracy remains insufficiently investigated, particularly under dynamic and multi-tenant execution environments. Accurate power attribution in cloud-native platforms is influenced by several dynamic factors, including CPU frequency scaling, C-state transitions, and resource contention among co-running containers/workloads that introduce resource contention among co-running containers/workloads(Ournani *et al.*, 2020).

Existing validation practices often validated power models only at a coarse level, for example, by comparing the sum of all estimated container powers to the total node power (as read from hardware). This approach makes it challenging to ascertain the accuracy of estimating the power consumption of each container. Moreover, they disregard the impact of several dynamic factors on the accuracy. Yet, there is currently no comprehensive framework that enables systematic evaluation of container-level power monitoring accuracy under such real-world conditions. Without such a dedicated accuracy-validation framework, container-level power measurements remain inherently unreliable, and this uncertainty propagates to any energy-optimization technique or evaluation built on top of these tools. As a result, both scientific conclusions and practical decisions become unreliable.

This paper aims to address this gap by introducing a novel framework designed explicitly to evaluate the accuracy of container-level power monitoring tools. The proposed framework enables a comprehensive assessment of container power monitoring tools' accuracy under dynamic environments that reflect real-world complexity, including dynamic frequency scaling, dynamic C-state configurations, and dynamic co-runner workloads (i.e., co-hosted containers executing concurrently on other cores of the same processor socket within a multi-tenant platform).. Using the proposed framework, we provide an in-depth evaluation of Kepler's accuracy under different dynamic power adjustments (i.e., dynamic CPU frequency, dynamic C-state transition). Experimental results reveal that the dynamic environment settings and the co-tenancy workloads significantly influence Kepler's accuracy. In some scenarios, Kepler's container-level power estimation showed an RMSE of up to 15.3 Watts against the ground truth RAPL, with peak overestimations reaching about 15x under low-frequency conditions. These findings highlight the need for further refinement to improve Kepler's accuracy and reliability. However, accuracy alone is not enough to fully leverage power observability in cloud environments. To this end, this work goes beyond accuracy validation and defines the key requirements that container-level power monitoring tools must satisfy to enable full power observability. The main contributions of the paper are threefold: (i) Accuracy-validation framework: A dedicated framework for evaluating container-level power monitoring accuracy under dynamic and multi-tenant conditions. (ii) Kepler accuracy evaluation: An experimental study of Kepler's accuracy across dynamic power settings and multi-tenancy scenarios. (iii) Requirement-driven analysis: Defining the key requirements to enable full power observability in cloud environments and analyzing how current tools meet these requirements.

The rest of this paper is organized as follows. Section 2.2 outlines the background, defines key requirements for power monitoring tools, and reviews existing solutions with a focus on Kepler. Section 2.3 introduces the proposed Power Accuracy Validation Framework. Section 2.4 details the experimental results and discusses the framework limitations and open research challenges. Finally, Section 7 concludes with key insights and future directions.

2.2 Power Monitoring in Cloud Environments

2.2.1 Overview of Power Consumption in Cloud Data Centers

The total data-center server power consumption can be decomposed as:

$$P_{\text{server}} = P_{\text{CPU}} + P_{\text{Memory}} + P_{\text{Disk}} + P_{\text{NIC}} + P_{\text{Platform}}. \tag{2.1}$$

Here P_{CPU} denotes processor power, P_{Memory} memory-subsystem power, P_{Disk} disk-storage power, P_{NIC} network-interface-card power, and P_{Platform} residual on-chip/platform-components power (Ismail & Materwala, 2020). Indeed, within data centers, other infrastructure elements—such as racks, switches, and cooling systems—also

significantly contribute to total data center energy consumption. However, this paper focuses explicitly on computational nodes (servers). The server power consumption is typically divided into two main categories:

1. Static power consumption: The static power represents the power a system consumes when idle, primarily due to leakage currents in transistors and other semiconductor components. The hardware's physical properties influence this type of power and remain relatively constant, irrespective of the workload.

2. Dynamic power consumption: The dynamic power is associated with the active operation of the CPU. It is determined by the switching activity of transistors, which involves charging and discharging capacitors during computations. Dynamic power is modeled as follows:

$$P_{\text{dynamic}} \propto C \cdot V^2 \cdot f \tag{2.2}$$

Where C refers to switched load capacitance, V refers to the supply voltage, and f indicates the operating frequency. While here we focus on the CPU's static and dynamic power, the same principles apply to other components such as memory, network, and disk storage.

2.2.2 Power-Saving Mechanisms in Modern CPUs

Modern processors incorporate several hardware mechanisms to enhance power efficiency, which include C-states and DVFS (Dynamic Voltage and Frequency Scaling).

2.2.2.1 C-States (Idle Power Management)

C-States are idle power states designed to reduce power consumption when CPU cores are not actively processing tasks. When a CPU core becomes idle, it transitions into one of several predefined C-states (Intel Corporation, 2020a). Each successive C state represents a deeper level of sleep, offering higher power savings. The CPU enters a specific C-state only if the CPU idle duration exceeds a predefined period called the target C-state residency. The deeper the C-state (e.g., C6 compared to C1), the longer the idle time required, but the greater the power savings. During deep C-states, parts of the processor, such as the clock or power to specific components, may be turned off to minimize power consumption. However, careful consideration must be given to the energy-performance trade-off, as deeper C-states introduce higher wake-up latencies, potentially impacting performance if the core is reactivated too frequently (Kanev, Hazelwood, Wei & Brooks, 2014).

2.2.2.2 Dynamic Voltage and Frequency Scaling

DVFS is a technique that explores the trade-off between power consumption and performance in modern multi-core computing platforms. It allows for dynamically adjusting CPU voltage and frequency at runtime, balancing power usage and performance based on workload demands. DVFS selects a frequency-voltage pair, known as a P-state, where higher P-states correspond to lower frequencies and voltages, thereby reducing power consumption at the cost of performance. Modern servers support per-core frequency scaling, allowing fine-grained power management and performance control.

C-states and DVFS complement each other, forming the core of modern CPU power-saving strategies. C-states reduce static power by transitioning idle cores into low-power states. DVFS addresses dynamic power consumption by adjusting voltage and frequency during active operation. CPU power management integrates DVFS and C-states via dedicated subsystems. In Intel CPUs, the *CPU Idle Driver* manages C-state transitions, while the *P-State Driver* controls voltage and frequency scaling (P-states).

Power consumption in cloud servers is typically measured using different tools at two levels: the server level and the container level.

2.2.3 Server-level Monitoring Tools

Server-level power monitoring can be performed using various methods, including physical power meters (Bedard, Lim, Fowler & Porterfield, 2010), dedicated acquisition systems (Kavanagh, Armstrong & Djemame, 2016), and software-based power meters. Intel's RAPL (Running Average Power Limit) (RAP) is one of the most widely adopted energy profiling tools for Intel devices. This hardware feature facilitates real-time energy consumption monitoring across multiple CPU domains, including DRAM (Dynamic Random-Access Memory) and integrated GPUs. Originally introduced with Intel's Sandy Bridge architecture, RAPL has since evolved to support later AMD architectures. RAPL categorizes power usage into different power domains (Khan, Hirki, Niemi, Nurminen & Ou, 2018):

- **Package (PKG)**: Measures the energy consumption of the entire CPU socket, encompassing cores, integrated graphics, and uncore components (e.g., last-level caches and memory controllers). The total package power can be expressed as:

$$P_{\text{PKG}} = P_{\text{PP0}} + P_{\text{PP1}} \tag{2.3}$$

- **Power Plane 0 (PP0)**: Tracks the energy consumption of all processor cores in the socket.

- **Power Plane 1 (PP1)**: Monitors the energy consumption of the integrated GPU in the socket.
- **DRAM**: Reports the energy consumption of the main memory.

Although RAPL provides a reliable ground truth for power measurement at the CPU level, its granularity is limited to the CPU socket; RAPL cannot directly attribute energy consumption to individual software processes or virtualized entities such as containers or virtual machines (VMs). These limitations make RAPL fall short in scenarios that require fine-grained power monitoring, such as containerized environments.

2.2.4 Challenges of Container-Level Power Monitoring

The cloud computing environment is characterized by significant diversity and heterogeneity in its hardware infrastructure and the workloads it executes. This includes various servers with different CPU vendors, versions, GPUs, and supported technologies. In addition, multiple levels of virtualization (VMs, containerization) enable flexible resource management. Orchestration tools such as K8s or serverless computing models further enhance this flexibility by providing end-users with transparent and abstracted resource management. In this landscape, cloud-native applications are becoming increasingly agnostic to the underlying hardware, operating systems, and virtualization layers. Fig. 2.1 illustrates the abstraction layers separating applications from the hardware and the levels of heterogeneity at each layer. Given the complexity of cloud environments and the coarse granularity of built-in power meters (e.g., Intel RAPL), developing effective power monitoring tools for cloud-native applications is critical. These tools must be capable of accurately measuring and monitoring the power consumption of a cloud-native application across diverse configurations, irrespective of: *(i)* how the application is deployed; *(ii)* its dependencies; *(iii)* its integration or interaction with other infrastructure components (i.e., storage services, database services, security modules); the hardware on which it runs, and *(iv)* the type of resources it utilizes (i.e., disk storage, network, CPU, memory).

2.2.5 Key Requirements for Power Monitoring in Cloud Environments

The following outlines the key requirements we consider essential for container-level power monitoring. These requirements are equally applicable to virtual machine (VM) power monitoring.

2.2.5.1 Container-Level Granularity

In cloud environments, containers are the fundamental execution units for deploying applications. Consequently, monitoring the power consumption of an application necessitates an accurate estimation of its containers' power usage, making container-level power granularity a critical requirement for cloud power monitoring tools. Typically, this could be achieved by distributing the total server power across containers proportionally, based on resource

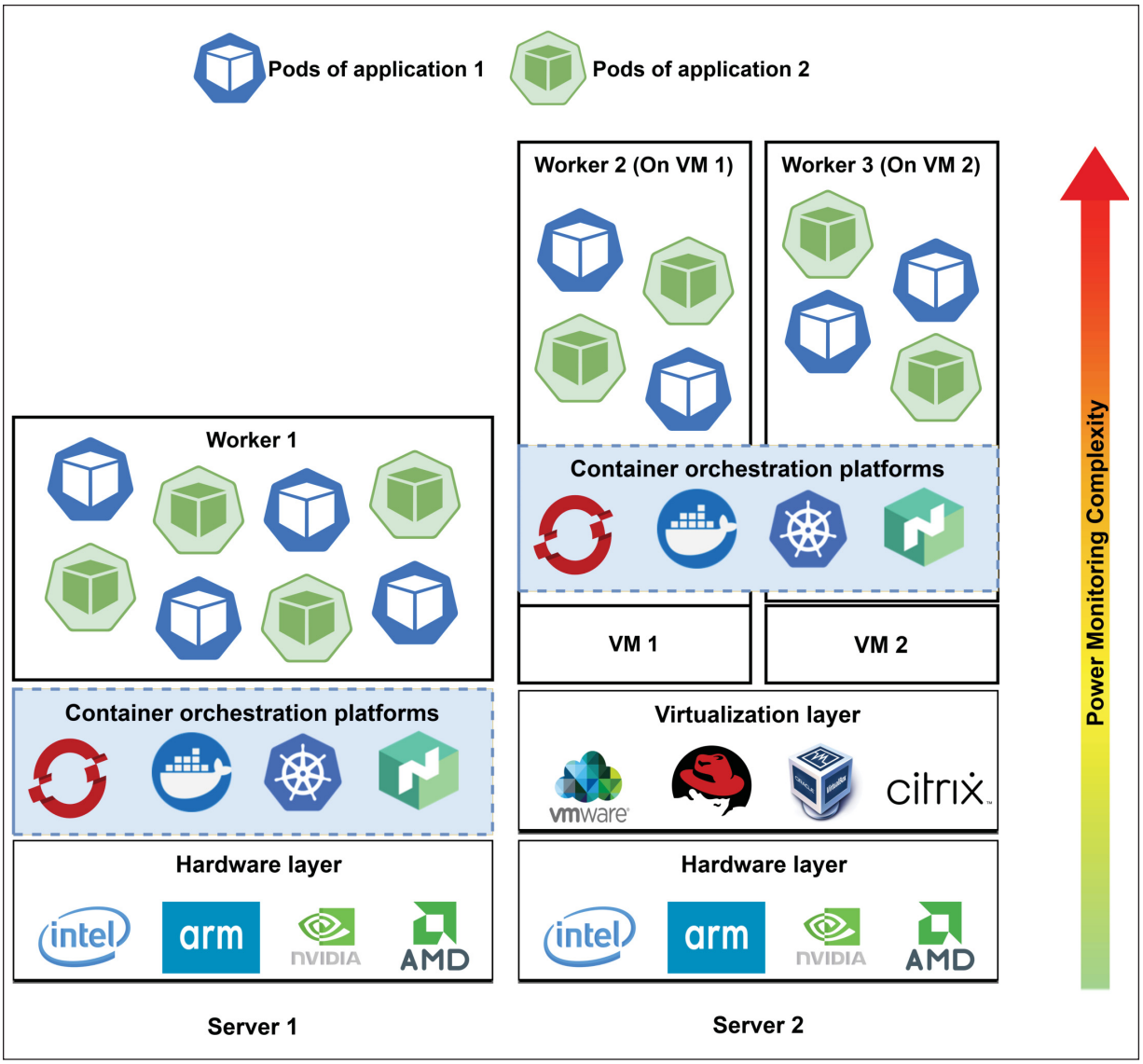


Figure 2.1 Complexity of container power monitoring

usage metrics of the applications over the server, such as CPU and memory utilization. However, this estimation becomes significantly more challenging in environments where direct power measurements are unavailable, such as on virtual machines or bare-metal servers lacking built-in power meters. In such cases, monitoring tools must first estimate the total system or node power before attributing consumption to individual containers.

2.2.5.2 Transparent Power Aggregation

Cloud-native applications typically consist of multiple containers distributed across cloud nodes. The total power consumption for a specific application or component (e.g., database, firewall, gateway) represents the aggregated

consumption of all related containers or pods, regardless of their physical placement across nodes. Achieving transparent and accurate power aggregation poses challenges, such as real-time synchronization and efficient metric collection, commonly addressed using standard monitoring tools like Prometheus.

2.2.5.3 Consideration of Idle Power Consumption

Applications consume power even when idle, such as event-driven microservices waiting for events. This occurs because resources like CPU and memory remain allocated, preventing the release or shutdown of the underlying resource. As a result, idle software components contribute significantly to overall power usage, yet this factor is often overlooked in traditional power monitoring approaches. According to Standard Performance Evaluation Corporation (SPEC) SPECpower benchmark (Standard Performance Evaluation Corporation (SPEC), 2008) , active idle power consumption can vary widely, ranging from 20% to 60% of the power consumed at maximum utilization. With rising regulatory pressure, accurate power consumption reporting is increasingly critical. Thus, precise measurement of idle power is essential, especially in cloud environments where clients might be charged proportionally to their total energy usage.

2.2.5.4 Support for Multi-Power Domains

Running containers utilize multiple resources, including CPU, GPU, memory, disk storage, and network, as outlined in (eq. 2.1). Together, these resources contribute to the overall power consumption of a container. Certain components, such as the CPU, GPU, and memory, benefit from well-established embedded power meters, such as Intel RAPL for CPU/memory and NVIDIA NVML for GPU. However, power consumption in other domains, such as network and storage, remains largely overlooked. This gap in observability poses significant challenges, particularly for Input/Output (IO)-intensive applications, where network and storage are the main power contributors. Therefore, power monitoring tools must support multiple power domains to achieve comprehensive container power monitoring.

2.2.5.5 Low Overhead

Container power estimation often depends on ML models using hardware performance counters as input.However, relying on custom performance counters like FLOPS (Floating Point Operations Per Second) or MIPS (Million Instructions Per Second)—which hardware does not directly provide—necessitates additional processing, introducing additional overhead. This issue is intensified by the high sampling frequency (milliseconds to seconds) needed for precise power measurements. Consequently, minimizing overhead is critical to ensuring efficient and accurate power monitoring without adversely affecting system performance.

2.2.5.6 Continuous Integration /Continuous Delivery (CI/CD) Integration

Power monitoring tools for cloud-native environments must be designed to focus on ease of deployment, seamless integration, and lifecycle management, enabling them to be integrated naturally into modern CI/CD pipelines. Standardized CI/CD mechanisms such as Helm charts, Customize, and K8s Operators should be leveraged to simplify installation, configuration, and upgrades of power monitoring artifacts.

2.2.5.7 Support for Orchestration Frameworks

Managing applications in the cloud often relies on container orchestration frameworks such as K8s and OpenShift. Supporting power monitoring across diverse orchestration frameworks is essential. This ensures accurate and consistent power consumption tracking, regardless of the underlying orchestration frameworks.

2.2.5.8 Integration with Telemetry Stacks

Seamless integration with widely adopted telemetry stacks—such as Prometheus, Grafana, and OpenTelemetry—is essential for enabling efficient monitoring workflows in cloud-native environments. Power monitoring tools must support these frameworks to ensure compatibility with existing observability pipelines and to simplify deployment for administrators and developers. Such integration eliminates the need for custom instrumentation, reduces configuration overhead, and ensures that power metrics are seamlessly incorporated into standard monitoring dashboards and alerting systems.

2.2.5.9 Hardware Architecture Independence

The inherent hardware heterogeneity in cloud computing—including variations in CPU architectures, processor generations, disk storage, and GPU models—requires power monitoring tools to support diverse configurations. To provide accurate and scalable observability, these tools must abstract underlying hardware differences, offering a consistent view of power consumption across platforms. Such architecture independence enables seamless deployment in distributed environments, eliminating hardware-specific adjustments and ensuring scalable, vendor-neutral power monitoring. This generalization is crucial for reliable energy observability in modern cloud environments.

2.2.6 Survey of Container-Level Power Monitoring Tools

The rapid adoption of containerization as the dominant technology for deploying and managing cloud-native applications has made container-level power monitoring necessary.

To this end, several container-level power monitoring tools have been developed. cWatts++ (Phung, Lee & Zomaya, 2019), a closed-source tool, estimates power consumption using RAPL, hardware performance counters, and CPU temperature sensors. It offers application-agnostic monitoring by leveraging a third-party performance counter collector, namely PAPI (Mucci, Browne, Deane & Ho, 2015). However, using PAPI introduces significant overhead. For example, a single PAPI read can take up to 2000 CPU cycles (Community, 2024b), whereas a raw hardware counter read would only take tens of cycles (Community, 2024a).

SmartWatts (Fieni *et al.*, 2020) is an open-source power monitoring tool for containers. Unlike existing solutions that rely on static power models requiring offline training or hardware-specific configurations, SmartWatts utilizes online calibration and leverages readily available hardware performance counters to dynamically adjust its power models for CPU and DRAM at runtime. This self-calibrating approach enables accurate power estimations without prior training or specialized equipment, making it highly adaptable to diverse hardware platforms and workload conditions. However, SmartWatts depends on a fixed set of performance counters, which may not be available on all hardware architectures.

SelfWatts (Fieni *et al.*, 2021) addresses this limitation by automatically selecting relevant performance counters based on the underlying architecture, thereby optimizing power models for heterogeneous platforms and improving accuracy under dynamic workloads. Nevertheless, both SmartWatts and SelfWatts rely on *cgroup v1*, which introduces non-negligible overhead when mapping power consumption from processes to containers. WattsApp (Mehta, Harvey, Rana, Buyya & Varghese, 2020) leverages neural networks for power-aware scheduling by estimating container-level power consumption and mitigating power cap violations through resource reallocation or migration. However, unlike SmartWatts and SelfWatts, its offline-trained model limits adaptability to diverse architectures or varying workloads.

Recent tools such as Scaphandre (Hubblo) and Energat (Hè, Friedman & Rekatsinas, 2024) aim to provide thread- or process-level power monitoring but require custom queries to approximate container-level granularity. Moreover, Energat introduces up to 10% overhead during energy measurement. Process-level power accuracy validation for PowerAPI (the underlying framework used by SmartWatts and SelfWatts) and Scaphandre, as presented in (Cadorel & Saingre, 2024), reveals that even under controlled conditions, both tools exhibit notable deviations from hardware-based ground truth—up to 19.1% for PowerAPI and 17.4% for Scaphandre.

The work in (Pijnacker, Setz & Andrikopoulos, 2025a) introduces KubeWatt, which estimates container-level power consumption in K8s environments. However, it relies solely on CPU utilization, disregarding other contributors such as RAM power. As a result, it may under- or overestimate both static and dynamic power components, leading to inaccurate measurements. While these tools provide valuable insights into container-level power consumption, they often lack the granularity, accuracy, and efficiency required for modern cloud infrastructure.

Recently, Kepler has emerged as one of the most promising tools for container-level monitoring.

Kepler: Kubernetes-based Efficient Power Level Exporter

Kepler is an open-source project that facilitates container-level power monitoring across heterogeneous computing platforms, including Intel, AMD, ARM, and NVIDIA GPUs. It supports multiple power data interfaces such as Intel RAPL, NVIDIA NVML, ACPI, and Redfish to obtain accurate power readings from diverse hardware sources. Although Kepler provides support for AMD and ARM architectures (Jiang, 2023), this functionality remains under active development and currently lacks the same level of hardware integration on Intel platforms (Gomez-Selles, 2024).

To collect system metrics with minimal overhead, Kepler utilizes an in-kernel eBPF-based library to gather performance counters, including CPU usage, clock cycles, and cache misses. In its latest release, Kepler migrated from *cgroups v1* to *cgroups v2* for container resource aggregation, reducing the overhead by up to 42%. Kepler is tailored for K8s-native environments, providing simple deployment via Helm charts or manifests. It integrates with Prometheus and Grafana to enable real-time visualization of energy metrics without additional setup. In Kepler, the power consumption is estimated per process and aggregated across containers, pods, and namespaces in distributed environments. Kepler divides power consumption into:

- **Idle Power:** Allocated based on the Greenhouse Gas (GHG) Protocol using resource proportions (e.g., CPU cores). For instance, a container using 8 out of 64 cores in a 160W idle system would receive 20W. However, it is important to note that this idle-power attribution feature is not yet implemented in Kepler's latest release (version 7.12).
- **Dynamic Power:** Kepler default power attribution model is the Ratio Power Model, which distributes the dynamic power proportionally based on resource utilization. For instance, a container consuming 75% of CPU resources is assigned 75% of the total dynamic CPU power.

While Kepler depends on total node-level power to perform container-level attribution, Kepler uses pre-trained power models in systems without direct power measurement interfaces (e.g., virtual machines or bare-metal hosts without built-in power meters such as Intel RAPL). These models are automatically generated by the Kepler Model Server and stored in a public repository, KeplerDB (Sustainable Computing IO, 2025), enabling deployment-wide access and consistent estimation. The node or VM-level estimated power derived from these models is then used to attribute container-level power consumption.

Table 2.1 compares representative tools against the key requirements for cloud power monitoring. In addition, the comparison highlights that Kepler imposes lower monitoring overhead than competing solutions, primarily because

Table 2.1 Feature comparison of container-level power-monitoring tools

Tool	Granularity			Idle Power	Open Source	Supported Power Domains	Telemetry Stacks	CI/CD Integration	Supported Infrastructure	
	C	P	N						With PM	Without PM
Cwtt++	Yes	No	No	No	No	CPU, DRAM	N/A	N/A	Intel	N/A
SmartWatts	Yes	No	No	No	Yes	CPU, DRAM	N/A	N/A	Intel	N/A
SelfWatts	Yes	No	No	No	Yes	CPU, DRAM	N/A	N/A	Intel	N/A
Scaphander	Yes	Yes	No	No	Yes	CPU, DRAM	Prometheus, Grafana	Helm	Intel	VM
EnergAt	Yes	No	No	Yes	Yes	CPU, DRAM	N/A	N/A	Intel	N/A
WattsApp	Yes	No	No	No	No	CPU, DRAM Net, Storage,	N/A	N/A	Intel ARM	N/A
Kepler	Yes	Yes	Yes	Yes	Yes	CPU, DRAM	Prometheus, Grafana	Helm ,Operator, Manifests, RPM	Intel, ~AMD, ~ARM, ~Nvidia GPU	VM, Bare Metal

it relies on eBPF-based kernel-level observability rather than intrusive polling or user-space aggregation. The table also shows that Kepler provides the broadest orchestration support, with native integration for widely adopted cloud-native platforms such as Kubernetes and OpenShift, whereas tools such as Scaphandre and SmartWatts offer only limited or Kubernetes-specific support, and others do not provide explicit orchestration integration. Kepler surpasses other tools, emerging as a strong candidate to enable power observability in cloud environments. However, its accuracy, particularly at the container level, has not been extensively investigated (Andringa, 2024; Pijnacker, 2024). Thus, we restrict our accuracy investigation to Kepler, as tools such as SmartWatts, SelfWatts, and Scaphandre have already been evaluated in (Cadorel & Saingre, 2024; Jay, Ostapenco, Lefèvre, Trystram, Orgerie & Fichel, 2023), showing limitations in accuracy. In contrast, tools like WhatsApp and CWatts++ are not open-source, which makes it challenging to conduct accuracy evaluations.

2.2.7 Challenges in Container-Level Power Accuracy Validation

A major challenge in validating container-level power attribution accuracy is the lack of a reliable ground-truth reference at the container granularity. To address this, several recent validation protocols have been proposed to assess the accuracy of container- and process-level power monitoring tools.

He et al. propose EnergAt (Hè *et al.*, 2024), a thread-level and NUMA-aware energy-attribution model validated using a summation-based protocol. The procedure includes: (i) estimating total host energy via Intel RAPL, and (ii) executing diverse stress-ng workloads (CPU-intensive, memory-intensive, and mixed) while computing their energy using EnergAt's analytic model. Attribution is deemed accurate when the sum of attributed energy—including EnergAt's overhead—matches the RAPL-measured host energy. The three workloads show an average deviation of 4.52%. The protocol also evaluates a noisy-neighbor scenario where a mixed workload (target) runs concurrently

with a mixed co-runner. The attributed energy of the target remains unchanged regardless of the co-runner, which is unexpected because mixed co-runners introduce uncore contention (LLC pressure, memory-bandwidth interference) that typically increases package-level energy. Yet the protocol still considers accuracy acceptable despite the absence of this variation. Pijnacker et al. assess Kepler's accuracy (Pijnacker *et al.*, 2025a). Their protocol focuses on attribution accuracy in the presence of idle pods: several idle pods (zero CPU usage) are created, followed by an N-CPU stressor pod; once the stressor is active, all idle pods are simultaneously removed. The results show attribution inconsistencies: Kepler assigns idle power to idle pods and redirects dynamic power to the *system_processes* namespace (background OS activity). This dynamic power increases when idle pods are added and decreases when they are removed. Although the protocol reduces background-process interference, it operates under default DVFS and C-state settings, limiting its ability to isolate attribution errors from power-state variability.

Cadorel and Saingre analyze the accuracy of process-level attribution tools (PowerAPI, Scaphandre) using a dedicated validation protocol (Cadorel & Saingre, 2024). Their evaluation studies the attribution under Hyper-Threading (HT), TurboBoost, and a co-runner workloads. Similar to EnergAt, the use of a single co-runner configuration is insufficient, as co-runner impact depends on socket load and is often non-linear. The protocol also ignores different co-runner types (memory-intensive vs. CPU-intensive). The protocol also evaluates attribution accuracy only against total processor power, preventing domain-level analysis (e.g., PKG vs. DRAM) that is essential for understanding attribution errors in multi-tenant settings.

Furthermore, these validation protocols share common limitations. First, they lack strong isolation from non-experimental overhead: CPU sockets are not fully isolated, allowing OS background processes to run concurrently with the target workloads. Second, they operate under default, dynamic power-management settings, with features like DVFS and C-states left enabled. Consequently, the measured power reflects not only the experimental workload but also autonomous hardware power-state transitions and background OS activity, introducing confounding variability that complicates the interpretation of attribution accuracy.

Prior studies evaluate power attribution accuracy only under narrowly defined or largely static conditions—such as the presence or absence of idle pods or the use of a single fixed co-runner workload under hyperthreading/Turbo (on/off) configurations. These scenarios are insufficient to capture the complexity of real deployments. In multi-tenant environments, heterogeneous containers (e.g., CPU-intensive, memory-intensive) share CPU sockets and uncore resources such as the last-level cache and memory controller, creating resource contention that directly affects power attribution accuracy. This variability is further amplified by server-side power-management mechanisms, including Intel P-states and C-states, which dynamically adjust core frequency and manage idle states. These mechanisms introduce runtime variability in power behavior, making it challenging to ensure consistent and reliable validation results. To address these challenges, we propose a novel accuracy validation framework specifically designed to assess the accuracy of container-level power monitoring tools under dynamic conditions. The framework enforces

high levels of isolation, stable workloads, and consistent configuration to control external influences. Section 2.3 provides a detailed description of the experimental environment isolation.

Importantly, the proposed framework enables the validation of power monitoring tools' accuracy under dynamic scenarios, including *(i)* dynamic CPU frequency scaling, *(ii)* dynamic Idle state transitions *(iii)* dynamic co-runner workloads on shared compute nodes. Our comprehensive methodology delivers valuable insights for academia and industry. Ultimately, the proposed framework is applied to identify limitations in Kepler's current power measurement and supports the development of more accurate, reliable cloud-native power observability tools. Table 2.3 summarizes the different scenarios under which accuracy has been evaluated.

It is important to note that the idle-pod power attribution issue in Kepler, as discussed in (Pijnacker *et al.*, 2025a), is revisited in this study. Our findings confirm that the problem has been resolved in recent Kepler releases (see section 2.4.5).

Additionally, unlike DVFS or C-state transitions—which continuously adapt to workload fluctuations and power demands—hyper-threading is configured at the BIOS level and remains fixed throughout execution. As a static architectural setting, it does not introduce runtime variability, making it less relevant for evaluating power attribution accuracy in dynamic validation scenarios considered in this study.

Table 2.2 Testbed Configuration Summary

Attribute	**Specification**	**Attribute**	**Specification**
CPU Model	Xeon Gold 6132	Threads/Core	1
Cores/Socket	14	Sockets	2
NUMA Nodes	2	CPU Freq (GHz)	1.0–2.6
Uncore Freq (GHz)	2.4 (fixed)	OS	Ubuntu 22.04.5
Kepler Version	7.12	L3 Cache	38.5 MB/socket

2.3 Container-Level Power Accuracy Validation Framework

Fig. 2.2 illustrates the proposed accuracy validation framework. Our validation framework consists of the following key steps:

Setup of the isolation environment

To minimize power measurement noise at the CPU socket level, we constructed a controlled testbed environment AI, as illustrated in Table II-2. We follow the experimental guidelines established by (Ournani *et al.*, 2020) to ensure stable and reproducible results. In addition, we extend the configuration to incorporate additional settings,

Table 2.3 Accuracy-Validation Scenarios

Ref.	**Target Tool(s)**	**Accuracy validation under dynamic scenario**				
		Co-Runner Workload	**DVFS**	**C-State**	**Turbo**	**HT**
(Hè *et al.*, 2024)	EnergAt	Mixed target + mixed co-runner	No	No	No	No
(Pijnacker *et al.*, 2025a)	Kepler	CPU-intensive target + idle pods	No	No	No	No
(Cadorel & Saingre, 2024)	PowerAPI, Scaphandre	CPU-intensive target + single co-runner	No	No	Yes	Yes
This Work	**Kepler**	**CPU/Mem target + dynamic co-runners**	**Yes**	**Yes**	**Yes**	**No**

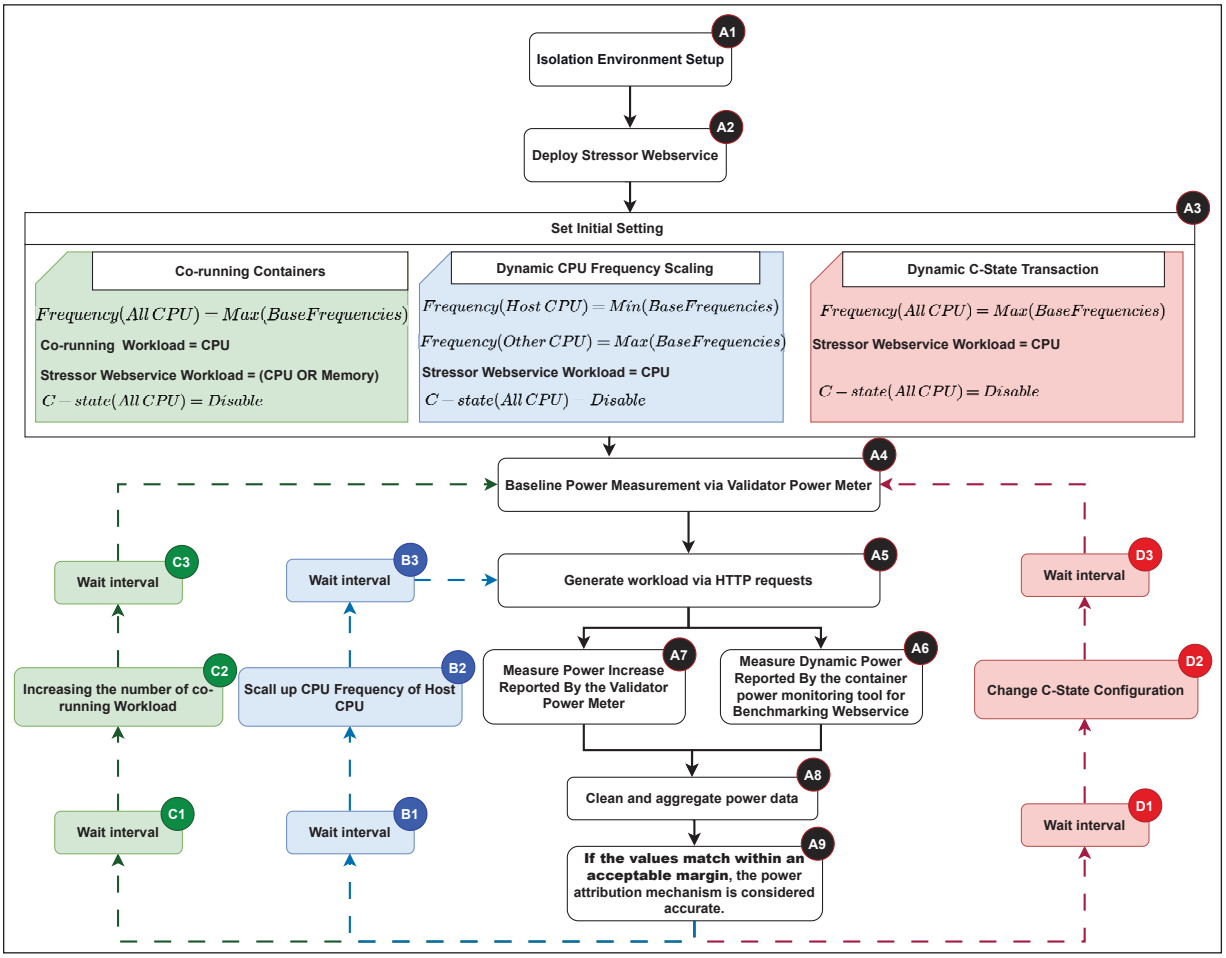


Figure 2.2 Accuracy Validation Framework for Container Power Monitoring Tool

such as CPU frequency governors, memory alignment, etc, that are known to influence power measurement reliability significantly. The proposed framework requires a multi-socket CPU architecture to enforce strict separation between experimental workloads and non-experimental processes. This hardware-level isolation prevents

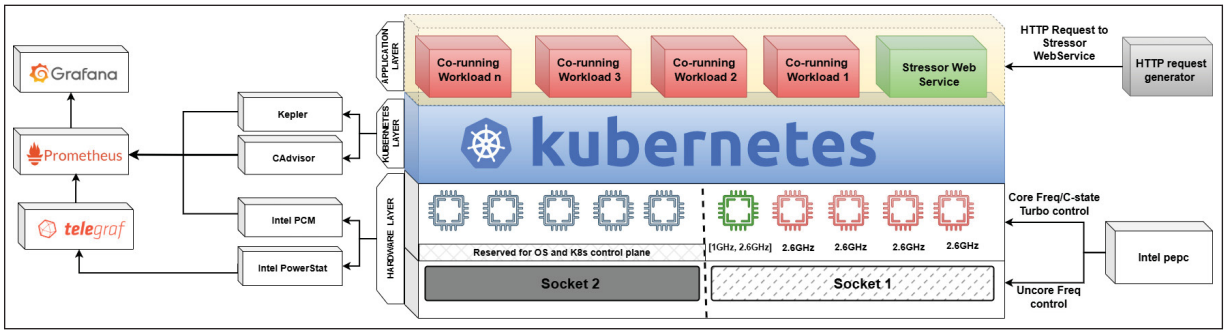


Figure 2.3 Overall architecture of testbed

background-process overhead from influencing power measurements and ensures a controlled environment for power-accuracy validation.

In this Kepler accuracy investigation, we limit our evaluation to Intel-based CPU architectures, as Kepler currently lacks equivalent hardware integration and sensor accessibility on AMD and ARM platforms as it does on Intel (IO, 2024a).

In our environment, we focus on two key objectives:

- Ensuring maximum workload isolation under test by minimizing or avoiding resource interference from other active workloads (e.g., system background processes, co-runner workloads).
- Maintain stable power measurements by controlling key factors, such as CPU frequency scaling and C-state transitions, directly impacting power consistency.

To achieve these objectives, several configurations and settings are applied at multiple levels:

Hardware-Level Configurations

Several hardware-level configurations were applied to ensure stable and reliable power measurements, eliminating power fluctuations, resource contention, and unpredictable power behaviors. These configurations minimize external interference and provide a controlled experimental environment.

Turbo Boost Disabled: Turbo Boost was disabled to ensure consistent CPU frequency and eliminate power variability caused by dynamic frequency scaling above the processor's base frequency. Turbo mode may trigger temporary power spikes exceeding the Thermal Design Power (TDP), activating thermal throttling and causing unpredictable frequency reductions. Disabling Turbo Boost thus establishes a stable environment, ensuring accurate, reliable, and repeatable power measurements.

C-States Disabled: All CPU C-states (e.g., C1, C1E, C6) were disabled to eliminate power variability caused by transitions between active and idle states, which introduce fluctuations in power consumption.

Hyper-Threading Disabled: Hyper-Threading (HT) was disabled to ensure exclusive allocation of containers and pods to physical CPU cores. This prevents intra-core resource contention caused by logical sibling threads sharing internal microarchitectural components such as execution pipelines, instruction decoders, and L1 caches, thereby reducing runtime power variability and enhancing measurement consistency.

Userspace Frequency Governor: The power governor was configured to userspace mode to allow manual frequency scaling, thereby enabling explicit control over CPU frequency throughout the experiments. This configuration ensures that CPU frequency remains fixed or follows predefined values, preventing dynamic frequency scaling driven by power management policies.

Fixed Uncore Frequency: For all experiments, the uncore frequency was fixed at the maximum supported frequency (2.4 GHz), which prevents fluctuations in uncore and DRAM power consumption.

Memory Swap Disabled: Memory swapping and page caching were disabled to ensure all experimental workloads remained strictly in physical memory. This configuration prevents paging activities such as disk I/O and page faults, thereby eliminating performance variability and associated power fluctuations. Therefore, the reliability and reproducibility of power measurements are improved.

Kubernetes-Level Configurations

To further isolate experimental workloads and eliminate background process interference, specific K8s configurations were applied.

NUMA Alignment Memory allocations were configured to follow NUMA-aware placement, binding workloads to CPU cores and memory banks within the same socket. This configuration eliminates cross-socket memory traffic, which would otherwise introduce latency overhead and variability in power consumption due to increased interconnect activity.

Static CPU Management in Multi-Socket Architectures: A static CPU allocation strategy was employed using K8s' static CPU manager policy to ensure strict workload isolation across CPU sockets in multi-socket systems:

- *Socket-under-test:* the dedicated CPU package (Socket 1) used exclusively for experimental workloads and monitoring tools (Kepler, Prometheus, and Grafana). These tools exhibited stable CPU usage and consistent overhead throughout the experiments, ensuring no interference with the accuracy of power attribution.

- *Non-Experimental Socket (Socket 2):* Reserved for operating system and K8s control-plane processes. This isolation prevented background activities such as OS maintenance, kernel threads from affecting workload performance during tests.

Such hardware-level separation eliminates cross-interference that could bias power measurements. While software-based isolation methods (e.g., CPU pinning, QoS classes) can reduce interference, they cannot ensure complete separation of power domains (Sohal, Bechtel, Mancuso, Yun & Krieger, 2022). Therefore, a dual-socket configuration remains the most reliable setup for reproducible and accurate validation. Table 2.4 summarizes the background-process isolation settings and power-feature control across different accuracy validation studies.

Table 2.4 Summary of background-process isolation settings and power-feature control across different accuracy validation studies.

	Background Process Isolation				Power-Feature Control				
Work	Socket Isolation.	HT (disable)	Core Pinning	NUMA Alignment	Fixed core Frequency	TurboBoost (disable)	C-States (disable)	Memory Swap (disable)	Fixed Uncore Frequency
(Hè *et al.*, 2024)	No	No	No	No	No	No	No	No	No
(Pijnacker *et al.*, 2025a)	No	No	No	No	No	No	No	No	No
(Cadorel & Saingre, 2024)	No	Yes	Yes	No	Yes	Yes	No	No	No
This Work	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Power Monitoring and Control Tools

- **PEPC (Power, Energy, and Performance Configurator)**: Configures CPU power management settings, including core/uncore frequencies, C-states, and other power-related parameters.
- **cAdvisor**: Monitors container- and pod-level resource usage, including CPU and memory utilization.
- **Intel PCM (Performance Counter Monitor)**: Tracks core- and socket-level hardware performance counters (e.g., instructions per cycle, L3 cache misses).
- **Intel PowerState**: Measures real-time power consumption at the CPU socket level using Intel RAPL, serving as a baseline for power validation.
- **Kepler**: Estimates power consumption at the container and pod levels.

Visualization and Data Collection Tools

- **Telegraf, Prometheus, and Grafana**: This stack enables data collection, storage, and visualization, facilitating real-time monitoring and in-depth analysis of power metrics.

Stressor Web Service for Workload Generator

To generate stress workloads, we deployed a web service–based workload generator within a K8s cluster A2. The service enables on-demand execution of predefined stress-ng stressors via HTTP-triggered requests.

This approach enables remote experiment control and decouples the control process overhead from the experimental environment, preventing it from affecting power measurement accuracy.

Resource Allocation and Isolation

To enforce strict application-level isolation, the stressor web service is deployed with a guaranteed Quality of Service (QoS) class in K8s, ensuring exclusive resource allocation and preventing power measurement inconsistencies due to dynamic resource allocation. Using Kubernetes resource specifications, we allocate 2 GB of memory and bind the web service to a dedicated physical CPU core on the host machine. This core, referred to as the *host CPU core*.

2.3.1 Experimental Methodology

Initial Configuration Setup

Each validation scenario requires specific initial configurations A3, including: *(i)* Workload definition for the stressor web service. *(ii)* Workload definition for co-runner workloads (i.e., CPU-bound or memory-bound stressors). *(iii)* Initial CPU frequency and C-state configuration for the host CPU core, defined according to the scenario. For all *other CPU cores* within the socket-under-test, the CPU frequency is fixed at the maximum base level and C-states are disabled.

These configurations provide flexibility for scenario-specific accuracy testing while eliminating all potential sources of power fluctuation that may result from other CPU cores (e.g., background processes, frequency scaling, or idle-state transitions).

Baseline Power Measurement

Although the socket-under-test is fully isolated and no background processes are running, its power consumption is never zero, as the *non-host CPU cores* inherently draw a minimal static amount of power even in the absence of active workloads in adaptation to monitoring overhead.

To establish a reference power level, we record the power consumption of the socket-under-test (i.e., socket 1) over a fixed interval using a validator power meter (A4). The collected samples are averaged to mitigate the effects of transient fluctuations and ensure a stable baseline measurement. The duration of this measurement must be sufficient to ensure statistical reliability by keeping the variance of the collected samples within an acceptable threshold. In this work, we record power for 120 seconds before initiating workload generation from the stressor web service. This baseline captures non-experimental power consumption, including background system processes, monitoring overhead, and co-runner workloads. Consequently, any increase in power observed during the validation test is attributed exclusively to the workload under test (e.g., the stressor web service pod).

Workload Generation via HTTP Requests

The benchmarking workload is generated by sending a sequence of HTTP requests to the stressor web service (A5). Each HTTP request triggers the execution of a predefined stress-ng command. Requests are processed sequentially, eliminating variations introduced by concurrent execution and ensuring uniform workload distribution across all executions.

Power Data Collection

During workload execution, dynamic power consumption is recorded at 1-second intervals from two sources:

- **Container Power Monitoring Tool (Kepler) (A6):**
  - Kepler container idle PKG & DRAM
  - Kepler container dynamic PKG & DRAM
- **Validator Power Meter (Intel RAPL) (A7):**
  - RAPL container dynamic PKG
  - RAPL container dynamic DRAM

Where RAPL container dynamic PKG is computed as $\text{RAPL}_{\text{TotalPKG}} - \text{RAPL}_{\text{BaselinePKG}}$, and RAPL container dynamic DRAM is computed as $\text{RAPL}_{\text{TotalDRAM}} - \text{RAPL}_{\text{BaselineDRAM}}$.

It is important to clarify that RAPL itself is not estimated or modeled by the proposed framework. RAPL is a hardware-provided power/energy measurement interface that reports power at processor domains such as package

(PKG) and DRAM. However, RAPL does not directly provide container-level power. Therefore, in this work, the container-level RAPL reference is derived through controlled baseline subtraction. Specifically, the baseline power of the isolated socket is measured before workload execution, and the dynamic power of the workload under test is obtained by subtracting this baseline from the total RAPL power measured during execution. Under the proposed isolation setup, where the stressor web-service container is pinned to a dedicated physical core and non-experimental sources of power variation are controlled, the resulting power difference is attributed to the container under test.

Data Cleaning and Comparison

To ensure statistical reliability, outliers are removed from power measurements, and power data are aggregated using the mean to a single value to establish a baseline power. The container-level power values reported power monitoring tool (Kepler) is compared against the power recorded by the Validator Power Meter (Intel RAPL). If the measured values fall within a predefined margin of error (e.g., ±5%), the monitoring tool is considered accurate. These steps allow the framework to validate container-level power monitoring accuracy under static and constant conditions.

2.3.2 Accuracy Validation Tests

2.3.2.1 Test 1: Accuracy Validation Under Dynamic CPU Frequency Scaling

This test evaluates how CPU frequency scaling impacts Kepler's power measurement accuracy.

Initial Setting

Initially, all CPU cores within the socket-under-test are configured to operate at 2.6 GHz, except the host CPU core (i.e., the dedicated physical core pinned to the stressor web service container), which is set to 1.0 GHz. The stressor web service runs a CPU-bound workload from the stress-ng benchmark:

```
stress-ng -c 1 --cpu-ops 800 --cpu-method prime
```

Experimental Procedure

The host CPU core frequency is scaled incrementally from 1.0 GHz to 2.6 GHz in 0.2 GHz steps. At each frequency, 100 sequential HTTP requests are sent to the stressor web service, while simultaneously collecting power consumption data. Fig.2.4 illustrates Test1's workflow.

Since Kepler's default sampling interval is 30 seconds, a waiting interval of 30 seconds is introduced both before and after each frequency scaling step. This ensures distinct, non-overlapping measurements across successive frequency scaling.

Although Turbo Boost is disabled in our setup to ensure repeatability and eliminate runtime variability, the framework implicitly captures its effects through controlled CPU frequency scaling. Turbo Boost is essentially based on the same principle: it raises the CPU frequency above the base level when sufficient thermal and power headroom are available. By systematically varying the CPU frequency across the DVFS range, the framework provides a more comprehensive and systematic assessment of power-reporting accuracy than simply toggling Turbo Boost on or off.

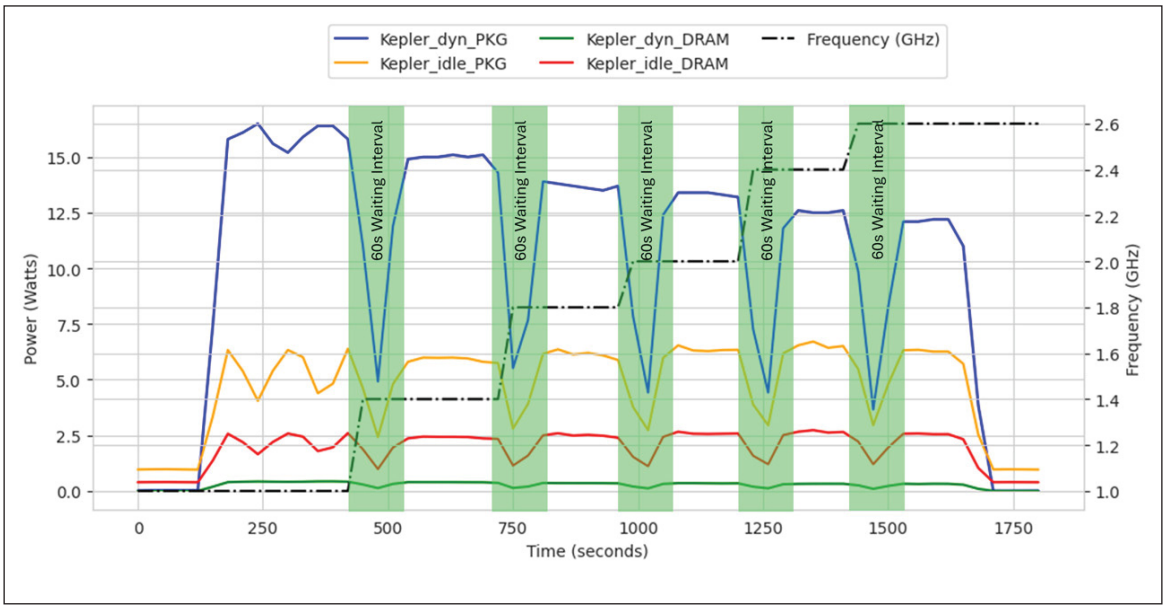


Figure 2.4 The progression of the test1 and its different measurement phases.

Expected Behavior

Since idle power is unaffected by CPU frequency scaling, and other CPU cores remain at fixed frequencies, the only co-hosted workload—the monitoring service— remains stable and does not introduce power fluctuations. Thus, any observed increase in power consumption is attributable exclusively to the dynamic power of the stressor web service. Although uncore power (from shared resources like L3 cache and memory) is difficult to fully isolate, its impact here is negligible due to the CPU-bound nature of the workload. Given that the workload is constant and runs in isolation on a dedicated physical CPU core, any observed power variations are solely a result of frequency scaling, independent of external system influences.

2.3.2.2 Test 2: Accuracy Validation in Multi-Tenant Environments

This test evaluates how co-runner workloads affect Kepler's power measurement accuracy. To cover different power domains (i.e., PKG and DRAM), the test comprises two steps:

Step 1: Accuracy Validation of Dynamic PKG Power Attribution in Multi-Tenant Environments

Initial Setting: For this validation test, all CPU cores on the socket-under-test are set to the maximum frequency (2.6 GHz). The stressor web service runs the same CPU-bound workload as in Test 1. The same workload is selected as the workload to be run by the co-runners. However, this workload will be executed by the co-runners continuously for 1 hour to maintain a constant overhead throughout the experiment. Co-runner workloads run natively via the stress-ng command-line tool rather than inside pods or containers. Each co-runner is pinned to a dedicated physical CPU core with a fixed memory limit of 1000 MB as follows:

```
systemd-run --scope -p MemoryMax=1000M taskset -c CPU_ID \
  stress-ng -c 1 -t 60m --cpu-method prime -q
```

Experimental Procedure: The number of co-runner workloads is gradually increased by two at each step, ranging from 0 to 12 co-runners. At each step, baseline power is measured. Similar to the first validation test, 100 sequential HTTP requests are sent to the stressor web service, while power data is collected throughout execution. A 30-second stabilization interval is introduced before and after increasing the number of co-runners to prevent measurement overlap and ensure that power fluctuations do not influence power readings due to co-runner cold start. As the number of co-runner workloads increases, baseline power measurements are continuously updated to account for the power contribution of co-runner workloads, ensuring that any additional power consumption is solely attributed to the stressor web service.

Expected Behavior: Since all CPU cores on the socket-under-test (i.e., socket 1) operate at a fixed maximum frequency, power fluctuations due to frequency scaling are eliminated. The workload is strictly CPU-bound, with minimal memory or cache usage, making uncore power interference negligible. By isolating the stressor web service on a dedicated core and processing one request at a time, we ensure a consistent workload regardless of the number of co-runner workloads. Therefore, any significant variation in power consumption results from misattribution rather than workload variability.

Step 2: Accuracy Validation of Dynamic DRAM Power Attribution in Multi-Tenant Environments

The same initial settings and procedure from Step 1 are retained, with one change: the stressor web service now runs a memory-bound workload. This validation test evaluates Kepler's accuracy in attributing DRAM power

consumption when memory, rather than compute, is the dominant resource. For the memory-bound workload, we use the STREAM benchmark from stress-ng, which generates sustained memory operations:

```
stress-ng --stream 1 --stream-ops 200 -q --stream-l3-size 4m
```

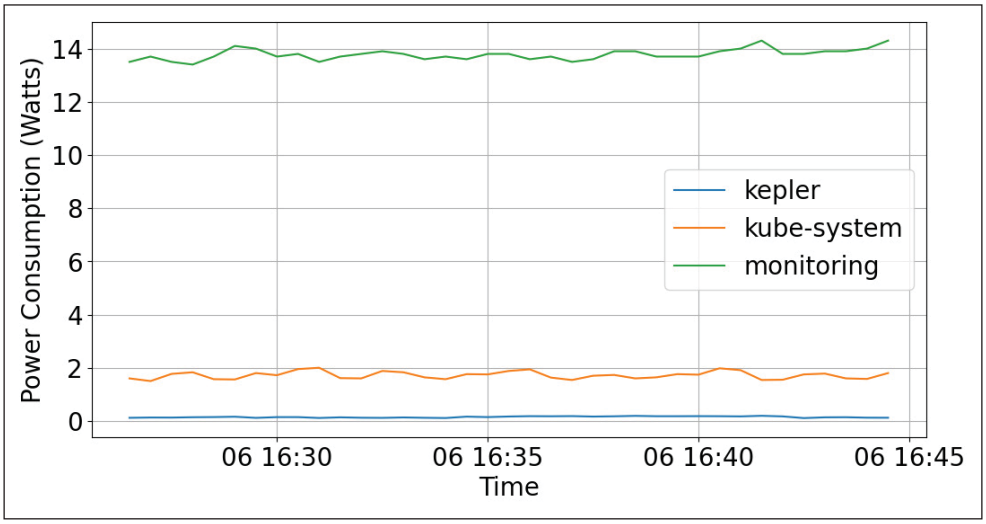


Figure 2.5 Power consumption of the monitoring subsystem and K8s system during the test.

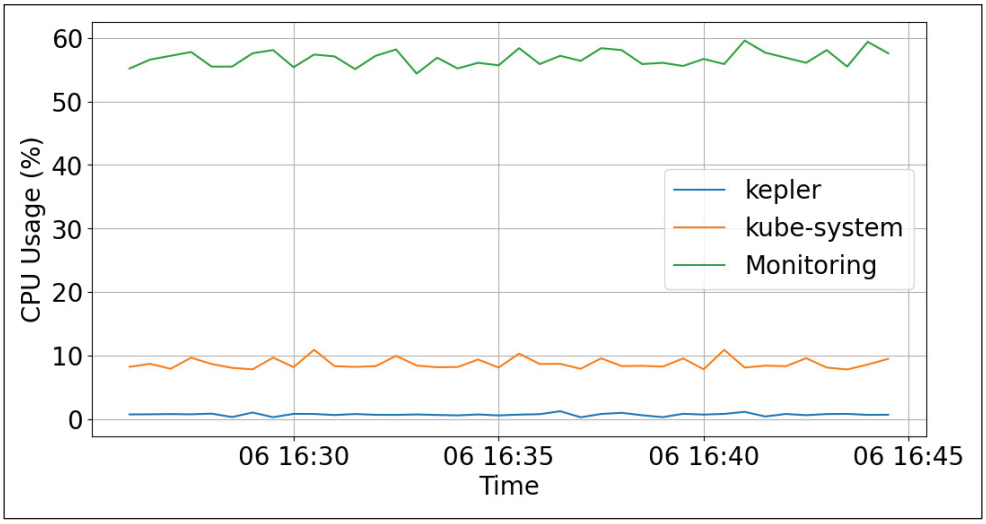


Figure 2.6 CPU usage of the monitoring subsystem and K8s system during the test.

Expected Behavior: Although the monitoring system (Grafana, Prometheus, and Kepler) is hosted within the same CPU socket as the stressor web service, the monitoring overhead remains stable throughout all conducted tests, ensuring consistent resource usage. Fig. 2.5 and Fig. 2.6 illustrate the monitoring subsystem's power consumption and CPU usage throughout the test, confirming that resource utilization does not fluctuate significantly. Since the co-runner workload exclusively utilizes computation resources with minimal memory access, it does not impose

additional demand on DRAM, ensuring that any observed changes in DRAM power consumption are solely due to the stressor web service.

2.3.3 Test 3: Accuracy Validation Under CPU Idle State Transitions

This validation test examines how the C-state transition of different CPU cores impacts Kepler's power attribution accuracy.

Initial Setting

All CPU cores on the socket-under-test are fixed at 2.6GHz, with C-states initially disabled. The stressor web service continues to run the same CPU-bound workload used previously in test 1.

Experimental Setup

The experiment is divided into two steps:

Step 1: C-states Disabled (Baseline Measurement) 100 sequential HTTP requests are sent to the stressor web service, and power data is collected throughout execution.

Step 2: C-states Enabled Following Step 1, C1, C3, and C6 C-states are enabled on all CPU cores except the Host CPU core, which continues to run the stressor web service and remains locked in the active C0 state. This configuration isolates the Host CPU core from idle-state transitions. A new baseline measurement is then recorded to reflect the lower package power associated with C-state activation. This step ensures that any subsequent power variations observed for the stressor web service can be exclusively attributed to its workload execution. The procedure from Step 1 is repeated under this new configuration. To ensure system stability, a 30-second pause interval is applied before and after the C-state transition, allowing the system to reach a steady-state condition before data collection.

Expected Behavior

The only change between Step1 and Step2 is enabling C-states for idle CPU cores, while the core running the stressor workload remains unchanged. Thus, any significant difference in power consumption is primarily due to misattribution errors rather than actual workload changes.

Table 2.5 Coefficient of Variation (CV%)

	Intel Container Pkg (W)			Intel Container Dram (W)			Kepler Container Pkg (W)			Kepler Container Dram (W)		
Validation Tests	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
Test 1	0.15%	0.32%	0.22%	0.30%	0.36%	0.33%	3.06%	12.37%	6.77%	3.23%	12.04%	7.83%
Test 2 (Step 1)	0.05%	0.22%	0.13%	0.15%	0.35%	0.26%	1.95%	5.55%	3.73%	2.30%	6.94%	4.67%
Test 2 (Step 2)	0.09%	0.34%	0.17%	0.41%	0.47%	0.44%	2.00%	6.54%	3.28%	2.77%	4.18%	3.46%
Test 3 (Step 1)	0.23%	0.23%	0.23%	0.34%	0.34%	0.34%	3.84%	3.84%	3.84%	5.65%	5.65%	5.65%
Test 3 (Step 2)	1.32%	1.32%	1.32%	0.32%	0.32%	0.32%	3.94%	3.94%	3.94%	4.83%	4.83%	4.83%

2.4 Results and Discussion

This section presents the key results obtained and discusses the main limitations of the study

2.4.1 Discussion: Stability of Power Measurements

To ensure the reliability of the results, we first evaluate the variability of the collected power data. Measurements were sampled at 1-second intervals and grouped based on CPU frequency (Test 1), the number of co-running applications (Test 2), and C-state settings (enabled/disabled) in Test 3. The data within each group were then aggregated using the mean. Table 2.5 reports the coefficient of variation across validation tests.

The *coefficient of variation (CV)* is a measure of relative variability that indicates how stable or consistent a dataset is. It is computed as the ratio of the standard deviation denoted as σ to the mean, denoted by μ, expressed as a percentage:

$$\text{CV\%} = \frac{\sigma}{\mu} \times 100\% \tag{2.4}$$

This normalized measure allows easy comparison of variability across different metrics, regardless of their scale or units. Intel's RAPL measurement exhibits strong stability across all validation tests. Package power shows minimal variation, with CVs between 0.15% and 0.34%, while DRAM power remains similarly stable, consistently below 0.36%. These low CVs confirm both the precision of RAPL-based measurements and the environmental stability of the testbed.

Kepler-based measurements show higher variability, as expected from estimation-based approaches. Package power CVs range from 2.00% to 12.37%, and DRAM power from 2.30% to 7.83%, with more pronounced fluctuations during dynamic configurations such as Test 1. Despite this variability, all CV values remain below 13%, indicating the absence of excessive fluctuations. RAPL measurements demonstrate high stability, and Kepler results—while more variable—are consistent enough for comparative analysis. These findings confirm the robustness of our experimental setup for evaluating power monitoring tools.

2.4.2 Test 1 Results: Impact of CPU Frequency Scaling on Kepler's Accuracy

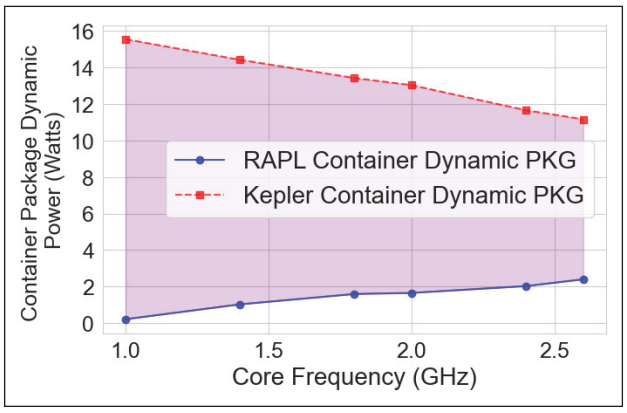


Figure 2.7 RAPL vs. Kepler – PKG Power Under Dynamic Frequency Scaling (RMSE: 11.92 Watts).

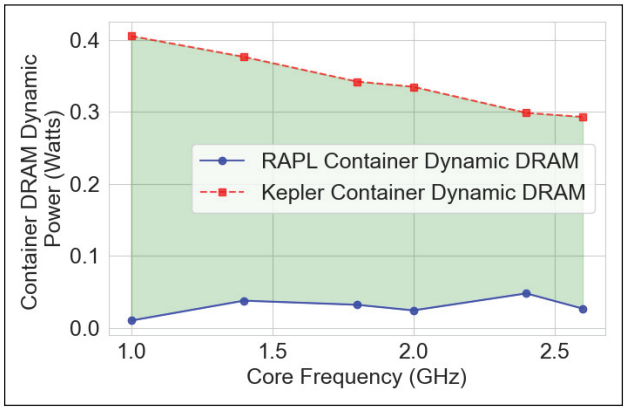


Figure 2.8 RAPL vs. Kepler – DRAM Power Under Dynamic Frequency Scaling (RMSE: 0.32 Watts).

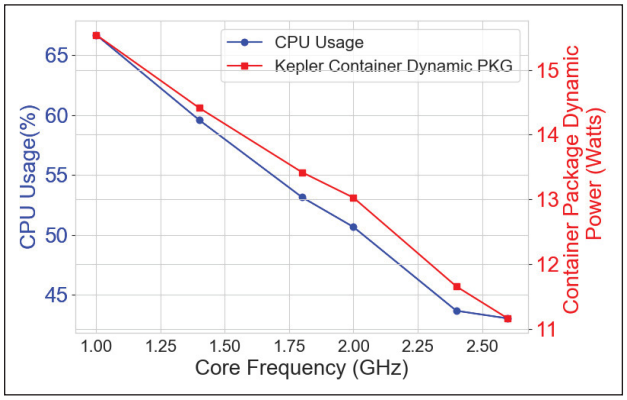


Figure 2.9 Relationship Between CPU Usage, CPU Frequency, and Kepler Dynamic Package Power.

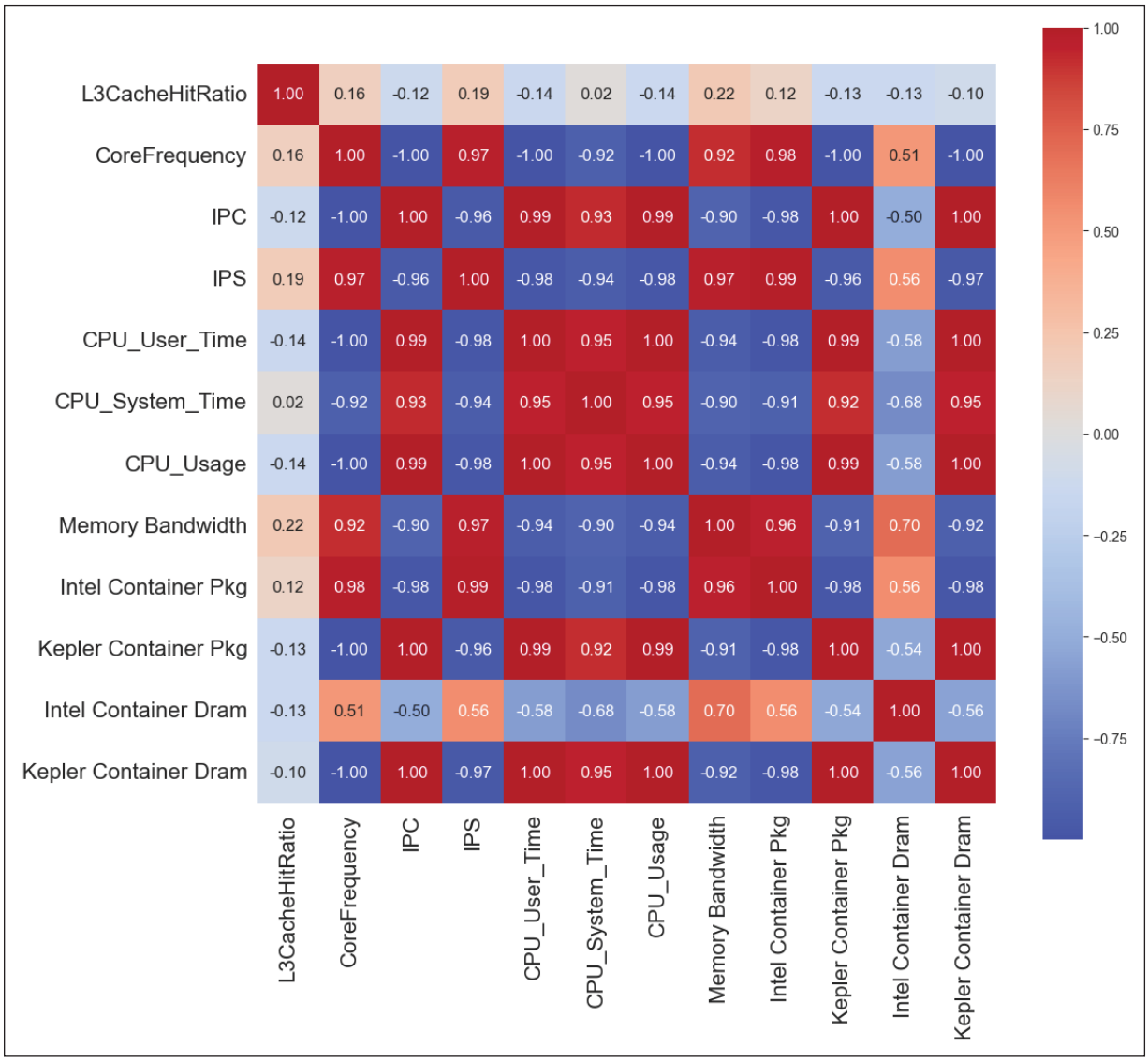


Figure 2.10 Correlation matrix of metrics and container power (Intel vs. Kepler).

Fig. 2.7 presents the container dynamic power under CPU frequency scaling as reported by Kepler and Intel RAPL. The results demonstrate a significant deviation between Intel RAPL and Kepler, particularly for package dynamic power, with Kepler's estimates showing an RMSE of 11.9 W against RAPL—equivalent to an overestimation of up to 15x. Kepler appears to attribute dynamic power based on CPU usage, assuming a linear distribution across running processes. For example, if a container or pod consumes 10% of CPU load, Kepler assigns it 10% of the total dynamic power. During this validation test, Kepler, Grafana, and Prometheus pods consistently report low CPU usage (2–5%), while the only CPU core within the socket experiencing significantly higher utilization (45–68%) is the Host CPU core running the stressor web service. As a result, Kepler misattributes a large portion

of the socket's total dynamic power to the stressor web service pod. Additionally, Kepler's initial dynamic power estimation may also be overestimated, further amplifying the discrepancy.

Beyond overestimation, Kepler reports a decrease in package dynamic power as CPU frequency increases—contradicting expected behavior. For CPU-bound workloads, higher frequencies should lead to increased power consumption (as implied by Eq 2.2), and confirmed by Intel RAPL measurements. This suggests that Kepler's power attribution model does not account for the effects of CPU frequency scaling. The inverse trend arises from Kepler's reliance on CPU usage as a proxy for power attribution. In our experiment, the workload remains constant (one request at a time), so higher frequencies yield more cycles per second, reducing CPU usage. Kepler interprets this drop as lower dynamic power, leading to incorrect attribution, as shown in Fig. 2.9. A similar issue is observed in DRAM power attribution (Fig. 2.8). Intel RAPL reports negligible DRAM power (0.01–0.05 Watts), which aligns with the workload's CPU-bound nature. In contrast, Kepler overestimates DRAM power by up to 4x, reporting 0.3–0.4 W, with an RMSE of 0.32 Watts against the RAPL ground truth. It also incorrectly indicates a decrease in DRAM power as CPU frequency increases, contradicting the expected trend, since DRAM power should remain relatively stable under low-memory-demand workloads regardless of CPU frequency scaling.

2.4.3 Test 2 Results: Impact of Multi-Tenant Workloads on Kepler's Power Attribution

Takeaway 1: Container-level power monitoring models should integrate additional performance metrics—such as CPU frequency, memory bandwidth, and cache miss rates—rather than relying exclusively on CPU usage. Figure 2.10 reports the correlation matrix between container power (Intel RAPL vs. Kepler) and the monitored metrics, showing that several hardware- and memory-level indicators exhibit non-negligible association with power consumption. Incorporating these metrics enables more accurate power attribution and enhances the overall reliability of power measurement in cloud-native environments.

2.4.3.1 PKG Dynamic Power Attribution

The stressor web service runs a stable, CPU-bound workload isolated on a dedicated physical core, ensuring no interference from co-runner workloads. As confirmed by Intel RAPL, package dynamic power remains consistent regardless of the number of co-runners on other cores. In contrast, Kepler overestimates package dynamic power when no co-runners are present (Test 1) and gradually decreases its estimates as co-runners are added, exhibiting an RMSE of 6.60 W against the RAPL ground truth. This near-linear decline occurs despite the stressor's CPU usage remaining unchanged, as shown in Fig. 2.11. These results indicate that Kepler's dynamic power attribution fluctuates with the number of co-runners, rather than as a result of changes in the workload under test. The use of a strictly isolated environment eliminates external sources of interference, confirming that the observed variation originates from Kepler's power distribution model.

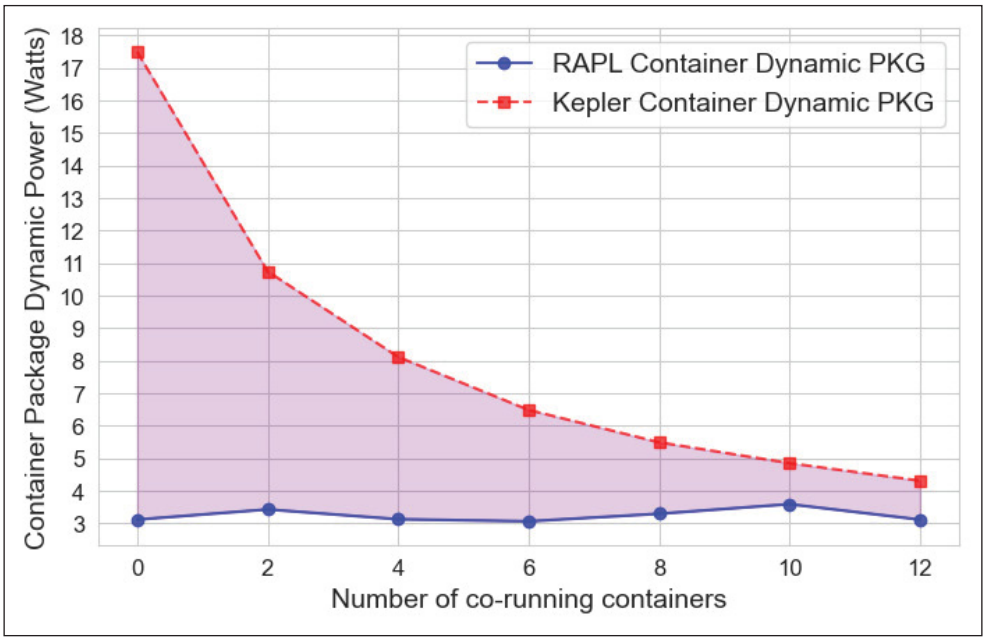


Figure 2.11 RAPL vs. Kepler PKG power under multi-co-runner workload (RMSE: 6.60 W).

Kepler's misattribution stems from a design constraint that forces the total dynamic power attributed across all containers to match the system-wide power consumption. As more co-runner workloads are introduced, Kepler redistributes dynamic power accordingly, decreasing the share assigned to the stressor web service. This rigid attribution policy, combined with the initial dynamic power overestimation, causes Kepler to rely heavily on the number of active containers rather than their actual resource usage when assigning power.

Takeaway 2: To improve accuracy, container power estimation should be independent of the number and composition of co-runners. Models that enforce a global power consistency constraint (i.e., forcing the sum of container estimates to match node-level power at each interval) make each container's attribution implicitly depend on other workloads, causing artificial power fluctuations whenever co-runners are added, removed, or change intensity. Instead, a resource-centric design should estimate each container's power from its own resource utilization profile, so that its attributed power primarily reflects its own behavior and remains stable under varying multi-tenant conditions.

2.4.3.2 DRAM Dynamic Power Attribution

The stressor web service is strictly isolated and runs a consistent memory-bound workload for each request, resulting in stable memory access patterns. Consequently, DRAM power consumption should remain constant, regardless of the number of co-runners. As shown in Fig. 2.12, Intel RAPL confirms this stability. However, Kepler initially underestimates DRAM power when no co-runners are present (i.e., an RMSE of 1.58 W against the RAPL ground

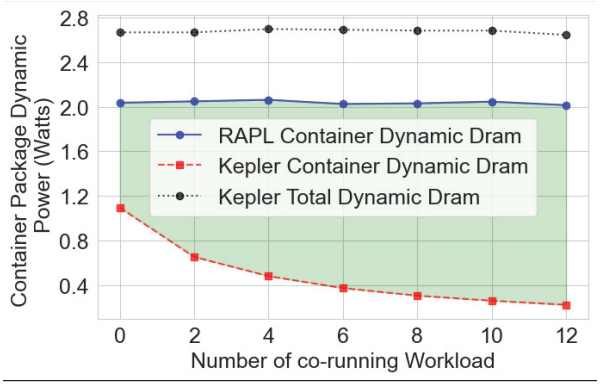


Figure 2.12 RAPL vs. Kepler DRAM power under multi-co-runner workload (RMSE: 1.58 W).

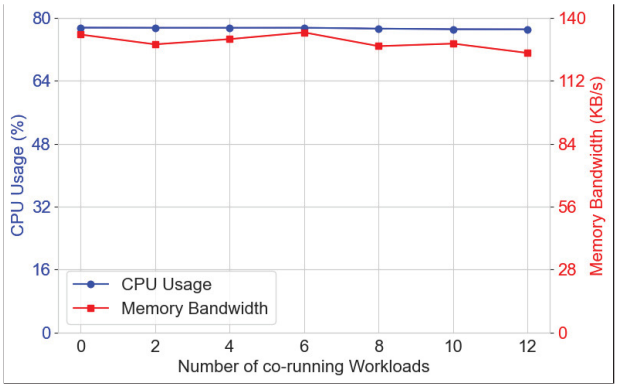


Figure 2.13 CPU usage and memory bandwidth under multi-co-runner workload.

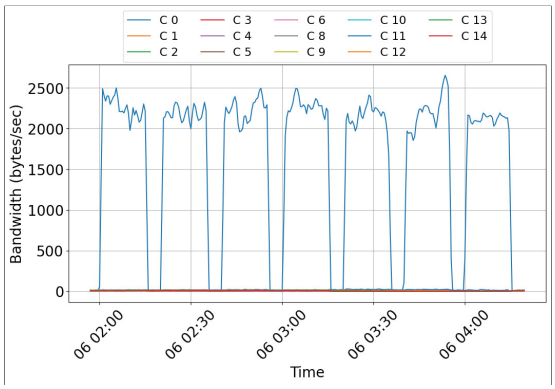


Figure 2.14 Memory bandwidth per CPU core during the test.

truth), as it distributes the total dynamic DRAM power across all active workloads, including system and monitoring services.

As the number of co-runners increases, Kepler's estimated dynamic DRAM power decreases proportionally. Fig. 2.12 illustrates that the total dynamic DRAM power estimated by Kepler remains unchanged throughout the test,

suggesting that the only factor impacting DRAM power attribution is the number of co-runners. To verify this, we analyzed the stressor web service's workload behavior. Fig. 2.13 shows its CPU usage and memory bandwidth—the primary drivers of DRAM power—remained stable throughout the test. This confirms that fluctuations in Kepler's DRAM power estimates are unrelated to stressor web service workload resource consumption. This implies that Kepler attributes DRAM power across all workloads, regardless of whether those workloads exhibit memory-related activity (e.g., memory accesses, L3 cache utilization, memory bandwidth usage). Fig. 2.14 further supports this finding, showing that only CPU core 1—the Host CPU core—consumes significant memory bandwidth. In contrast, all other cores hosting the co-runner workloads exhibit negligible memory bandwidth usage. These findings indicate that Kepler's power attribution model overlooks actual memory activity, resulting in inaccurate DRAM power estimation in multi-tenant environments. To improve accuracy, DRAM power should be attributed based on workload-specific memory access behavior—such as memory bandwidth and cache utilization—rather than relying on unrelated metrics like CPU usage.

Takeaway 3: DRAM power attribution in container-level power monitoring tools should be driven solely by memory- and uncore-related resource utilization—such as memory bandwidth, last-level cache activity, and related traffic indicators—rather than by CPU-centric or globally uniform distribution models. Aligning the DRAM domain with its actual physical drivers ensures that memory-bound containers receive a proportionally higher share of DRAM power, while CPU-bound or cache-friendly workloads are not incorrectly charged for memory consumption they do not generate. This targeted attribution approach reduces systematic misestimation across heterogeneous workloads and yields more faithful, interpretable DRAM power measurements at the container level.

2.4.4 Test 3 Results: Influence of C-State Settings on Power Attribution Accuracy

The only change between Step 1 and Step 2 is the activation of C-states for idle CPU cores, while the Host CPU core remains unchanged. As shown in Fig. 2.17, which shows consistent CPU utilization. Under these conditions, the stressor's power consumption should remain stable. Intel RAPL measurements confirm this, reporting a steady dynamic package power of 2.6W in both steps (Fig. 2.16). However, Kepler's estimated dynamic package power drops significantly —from 11.19W to 5.64W—after enabling C-states, reflecting a 49.6% reduction. This discrepancy suggests that Kepler's power attribution model is highly sensitive to global reductions in system power. Kepler attributes dynamic power by proportionally distributing the total package power across active workloads based on CPU usage. When idle cores transition into deeper C-states, the overall package power decreases. However, Kepler incorrectly reduces the attributed power uniformly to all workloads, even for those unaffected by idle-state transitions. This results in an inaccurate measurement of the stressor web service's power consumption.

Takeaway 4: Container power monitoring should incorporate per-core C-state residency as a primary feature for attributing idle-power savings. By conditioning power reductions on the residency states of the specific cores

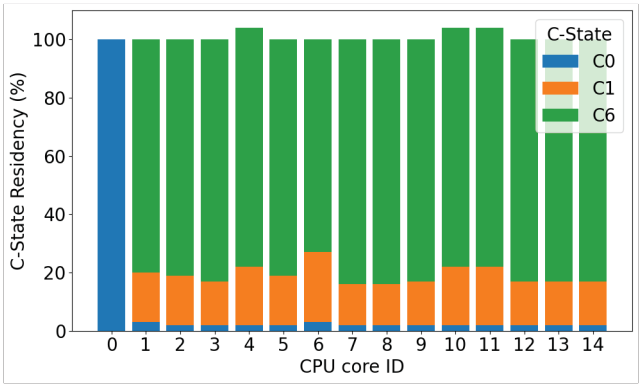


Figure 2.15 CPU C-State residency percentages per CPU core.

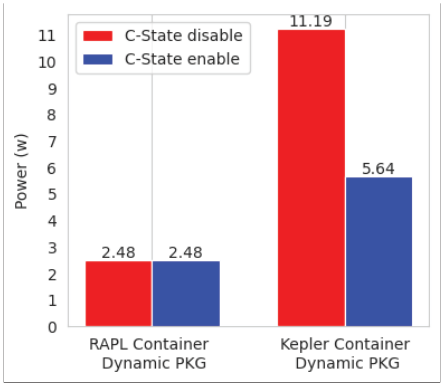


Figure 2.16 RAPL vs. Kepler PKG power under different C-state settings.

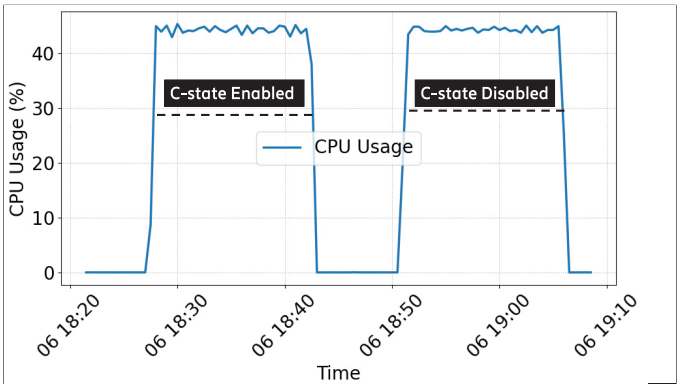


Figure 2.17 CPU usage under different C-state settings.

hosting each container, the model ensures that only workloads actually benefiting from deeper sleep states receive reduced attributed power. This per-core attribution model avoids the uniform redistribution of power savings due to C-state activation across all cores regardless of their activity level and enables more accurate and workload-specific power attribution in multi-core environments.

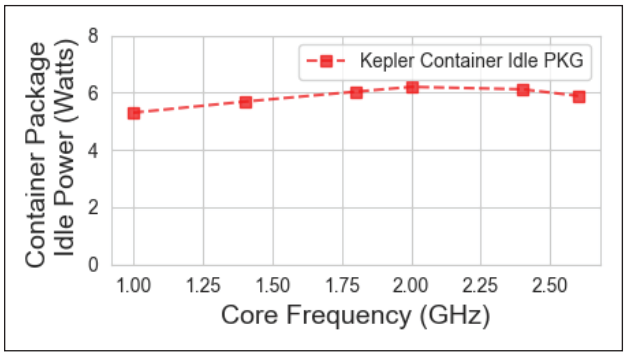


Figure 2.18 Kepler Idle PKG Power Under Dynamic Frequency Scaling

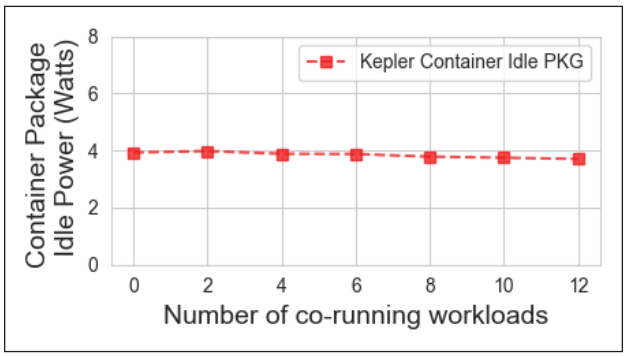


Figure 2.19 Kepler Idle PKG Power Under Multi-CoRunner Workload

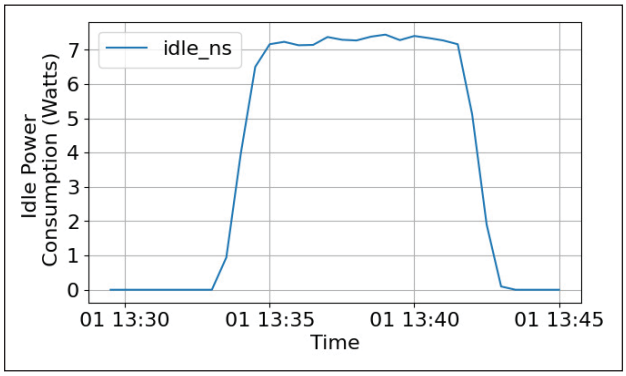


Figure 2.20 Kepler Idle PKG Power for Complete Pods

In addition to validating dynamic power attribution, we also examined Kepler's idle power distribution to assess its response to various conditions. Fig. 2.19 illustrates the effects of CPU frequency scaling and co-runner workloads on idle power reporting. Kepler allocates idle power based on container resource reservations. Since the stressor web service runs with a guaranteed QoS (2GB memory and one physical CPU), Kepler consistently reports stable

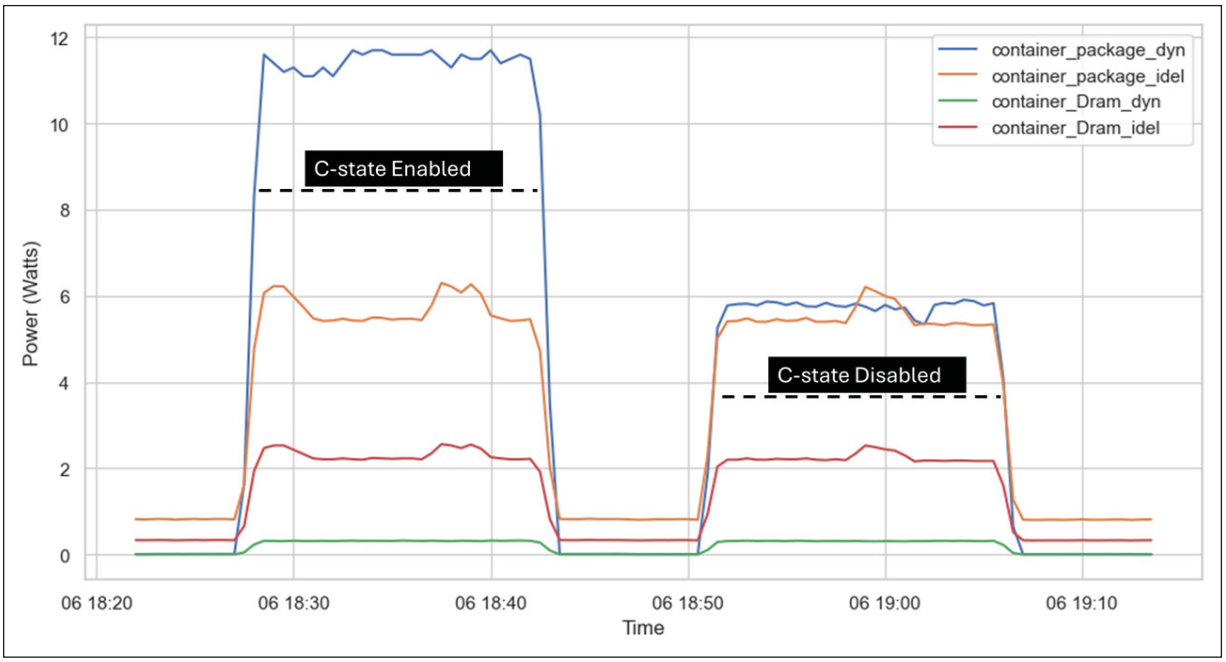


Figure 2.21 Kepler Idle PKG Power Under Different C-State Settings

idle power, independent of CPU frequency or the number of co-runners. These observations suggest that Kepler strongly decouples idle power from dynamic workload fluctuations.

Enabling C-states affects only Kepler's dynamic package power estimation rather than its idle power measurement, as illustrated in Fig 2.21. Because C-states apply exclusively to idle cores, their activation reduces only static (idle) power, not dynamic power (Bellal, Lahlou, Kara, Murphy & Nguyen, 2025a). However, Kepler incorrectly attributes these static power reductions to dynamic power. The root cause of this issue is that Kepler seems to be initializing a fixed static power estimate and does not update it dynamically during runtime, despite changes in idle cores' C-state residency.

Takeaway 5: Container power monitoring should dynamically adjust static power estimates using real-time, core-specific C-state residency, so that power savings from deeper idle states are explicitly reflected in the static component rather than being absorbed into dynamic power. By recalibrating static power downward when cores hosting a container spend more time in deeper C-states, the model preserves a clean separation between baseline (idle) and activity-driven power. This prevents idle-power reductions from being misinterpreted as changes in dynamic consumption, thereby improving the fidelity of dynamic power measurements and maintaining consistent, comparable attribution across varying idle conditions.

2.4.5 idle-pod co-runner validation scenario

Prior work (Pijnacker *et al.*, 2025a) showed that Kepler may incorrectly assign idle power to all pods listed in the K8s API, including inactive ones (i.e., pods in the completed state). We deployed 12 batch jobs to investigate this, each running a CPU-bound task for 2 minutes. The K8s garbage collector was configured to retain completed pods, creating a controlled set of inactive containers grouped under a dedicated namespace (*idle_ns*). We tracked idle power attribution to these pods after job completion. As shown in Fig. 2.20, Kepler assigned idle power only to active pods, confirming that the bug has been resolved in recent releases and that completed containers are now correctly excluded from idle power attribution.

2.5 Limitations & Future work

The experimental results show that the proposed validation framework effectively detects inaccuracies in leading container-level power monitoring tools. Nevertheless, the presented validation framework faces certain limitations, primarily regarding its coverage of more complex scenarios:

2.5.1 Unpinned CPU Scenario

Despite its effectiveness, the current validation framework requires workloads to be pinned to dedicated physical CPU cores to eliminate intra-core interference and accurately assess the effects of CPU frequency scaling and C-state transitions.

In contrast, dynamic resource allocation allows multiple containers to share CPU cores, introducing interference from context switching and cache contention across L1–L3 levels, which distorts power measurements. Accurately evaluating these effects is difficult due to the lack of per-core power measurement capabilities in existing hardware power meters. To overcome this limitation, future work will develop ML-based models that estimate per-core power consumption. These models will serve as reliable baselines for validating container-level power attribution in unpinned, dynamically scheduled environments.

2.5.2 Idle Power Validation

The current framework does not support evaluating the accuracy of idle power attribution by container-level monitoring tools. Since idle power cannot be directly measured, and built-in meters like Intel RAPL only report aggregate socket-level power (i.e., dynamic and static), validating reported idle power remains a challenge. To address this, future work will explore machine learning ML-based models capable of decoupling static (idle) from the dynamic power at runtime, such as the one presented in (Choochotkaew, Wang, Chen, Chiba, Amaral,

Lee & Eilam, 2024b). Such a model could enable accurate estimation of idle power, leveraging reliable validation of idle power distribution across containers.

2.5.3 Low-level metric accuracy validation

Power attribution models commonly rely on resource usage metrics at the container or process level. However, inaccuracies in these metrics—such as Instructions Per Cycle (IPC) and cache misses—can lead to incorrect power attribution. To ensure reliability, we plan to validate the metrics used by the power monitoring tool against ground-truth measurements obtained from trusted sources, such as Model-Specific Registers (MSRs) and performance monitoring tools like `perf`.

2.5.4 Hardware Isolation Requirement

The validation framework relies on a dual-socket architecture to achieve strict hardware isolation between experimental workloads and background system processes. This setup minimizes external interference and enables accurate socket-level measurements. Although this hardware requirement limits direct deployment on single-socket servers, it does not affect the generality of the results, since power-attribution accuracy depends on the monitoring model rather than socket count. Software-based isolation (e.g., CPU pinning, Kubernetes QoS, or bandwidth throttling) can approximate these conditions, but remains a soft isolation layer that may not fully prevent interference.

2.5.5 Reproducibility and Framework Deployment.

The current validation framework includes: (i) a manual BIOS-level configuration phase (e.g., disabling Hyper-Threading), and (ii) a Python script that parses a YAML experiment specification. This file defines per-core CPU frequency, uncore frequency, C-state settings, and other parameters. The script applies these settings through system commands to configure the testbed and run the experiment. To enhance reproducibility, we plan to deploy the framework using a Helm chart on Kubernetes. Validation scenarios will remain YAML-defined, allowing users to declaratively configure per-core settings, co-runner behavior, and dynamic control factors such as CPU frequency scaling. Helm will abstract the underlying setup and enable seamless deployment on a Kubernetes environment.

2.5.6 Uncore Interference.

When a memory-bound workload shares a host with similar workload types, interference at the uncore level—particularly in memory bandwidth and shared L3 cache—can significantly affect power consumption. For instance, cache evictions caused by co-runner workloads increase cache misses, requiring more frequent

memory access and raising power usage. Concurrent contention for memory bandwidth further complicates accurate power attribution, making validation in such scenarios extremely challenging. To address this, validation should be conducted under controlled cache and memory bandwidth conditions. Technologies such as Intel Resource Director Technology (Intel RDT) (Intel Corporation, 2020b) enable fine-grained resource partitioning per core, allowing experiments that systematically simulate uncore-level interference.

2.5.7 Cross-Architecture Accuracy Validation

Indeed, Kepler officially claims compatibility with AMD and ARM architectures ((2023), CNCF); however, this support remains under active development and does not offer the same level of hardware access or sensor integration on AMD and ARM.

The current state of Kepler's cross-architecture support is summarized as follows:

- AMD64 support: Kepler can operate on AMD-based nodes but with functional limitations. The tool attempts to use RAPL/powercap interfaces when available, but AMD's RAPL implementation remains less mature than Intel's. As a result, Kepler may not capture all energy domains (Package, DRAM) on AMD processors with the same accuracy on Intel.
- *ARM64 support:* ARM compatibility remains in an early experimental stage. While ARM images are available, power metrics on ARM rely mainly on software-based estimates rather than validated hardware sensors (Hat, 2024).

Given these constraints, the present accuracy validation focuses on Intel Xeon systems.

Once support for multi-architecture platforms in Kepler or similar tools stabilizes, we plan to integrate these architectures into the validation framework and conduct cross-platform evaluations. This will facilitate a systematic investigation into how hardware architecture influences power-attribution accuracy.

2.5.8 Cross-Platform Validation

We plan to complement this study with a dedicated container power monitoring accuracy analysis, applying our proposed validation framework to various available tools. This evaluation will examine how each tool performs under diverse runtime conditions, enabling a systematic comparison of power estimation accuracy. The study aims to identify which power-attribution models demonstrate higher precision and robustness across heterogeneous workloads and system configurations. These insights will inform the refinement or replacement of Kepler's current

attribution model, without necessarily adopting other tools directly, as some may offer better accuracy but still lack essential features required for comprehensive power observability in cloud-native environments.

2.6 Conclusion

Accurate power monitoring and attribution are essential for enabling observability in cloud environments, particularly given the dynamic and heterogeneous nature of containerized workloads. This paper outlines the key requirements for container-level power monitoring and presents a new accuracy validation framework designed to evaluate power monitoring tools under realistic conditions, including CPU frequency scaling, co-runner interference, and C-state transitions. Applying this framework to Kepler, a leading power monitoring tool, we observed that it exhibits an RMSE of 11.9 W against the RAPL ground truth (i.e., an overestimation of up to 15x), with its accuracy highly sensitive to dynamic runtime factors. These results highlight significant limitations under real-world conditions, indicating that further refinement and redesign of the Kepler power attribution model are required. Future work will focus on extending the validation framework to cover additional dynamic factors influencing power consumption, such as dynamic resource allocation. Moreover, we will explore and benchmark alternative container-level power models to select or design a model that provides higher accuracy. This advancement is critical for supporting energy-aware scheduling and achieving sustainability objectives in modern cloud infrastructures.

CHAPTER 3

K8S POWER IRRIGATION: DEEP REINFORCEMENT LEARNING FOR PERFORMANCE-AWARE POWER EFFICIENCY OF KUBERNETES CLOUD-NATIVE MICROSERVICES

Zouhir Bellal, Laaziz Lahlou, Nadjia Kara, Timothy Murphy and Tan Phat Nguyen

Department of Software Engineering and Information Technology at École de technologie supérieure,

1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Article submited in the journal

IEEE Transactions on Green Communications and Networking

on MARCH 26, 2026

Abstract: Modern cloud platforms are facing a sharp increase in power demand driven by the rapid adoption of AI-powered applications, making power optimization urgent under net-zero commitments and sustainability goals. Yet, reducing power in production remains challenging for latency-sensitive microservices (e.g., backend pipelines for autonomous driving, real-time AI inference, and AI/IoT workloads), where performance violations directly impact user experience and operational risk. These microservices exhibit heterogeneous workload characteristics (CPU-bound, memory-bound, and mixed) and diverse load patterns. In multi-tenant environments, especially for memory-intensive workloads, contention on shared uncore resources (e.g., last-level cache and memory bandwidth) can degrade performance and trigger violations of performance requirements. As a conservative safeguard, providers often pin servers to performance mode (maximum core and uncore frequencies). This occurs because existing power governors largely disregard application-level performance requirements and ignore uncore interference under contention, maintaining reliability by sacrificing energy efficiency and causing systematic power over-provisioning. To address this, we introduce K8SPI (Kubernetes Power Irrigation), a hierarchical reinforcement learning (HRL) controller that jointly optimizes CPU core and uncore frequencies for cloud-native deployments. K8SPI uses a two-stage control architecture: a coarse-grained agent rapidly mitigates performance violations, while a fine-grained agent iteratively minimizes power consumption once performance requirements are met. Guided by multi-level telemetry spanning hardware, Kubernetes, and application performance signals, K8SPI adapts to workload heterogeneity and cross-microservice interference to meet performance requirements with minimal node power. We evaluate K8SPI on a Kubernetes testbed across multiple scenarios. Experimental results show that K8SPI reduces node-level power consumption by 23–30% relative to the Linux performance governor while keeping performance requirement violations below 2–3%, even under severe uncore contention and dynamic load fluctuations.

Keywords: Power monitoring, Cloud power observability, Container-level power monitoring, Power accuracy validation framework, Kepler accuracy validation

3.1 Introduction

By 2030, the electricity demand of the IT sector is projected to reach approximately 3,200 TWh (i scoop). These projections likely underestimate future consumption due to the rapid proliferation of energy-intensive Artificial Intelligence (AI) workloads, particularly generative AI. Reports indicate that a single ChatGPT query may consume significantly more energy than a conventional web search (Saenko, a, b), with the additional global power capacity required for AI projected to reach 327 GW by 2030 (Pilz *et al.*, 2025). Even with innovations such as DeepSeek that reduce the cost of developing large language models (LLMs), overall energy demand is unlikely to slow down due to large-scale deployment and workload growth (de Kruijff, 2025).

This surge, combined with regulatory pressures such as carbon pricing schemes (Ahuja, 2024), prompts energy efficiency to be a primary operational objective for cloud providers. However, achieving efficiency in cloud-native environments, characterized by multi-tenancy, heterogeneous microservices, and dynamic load, remains a complex control problem.

Dynamic Voltage and Frequency Scaling (DVFS) is the standard mechanism for managing processor power. Modern architectures support scaling for both processing cores and the "uncore" (interconnects and Last Level Cache). While core frequency dictates computational throughput, uncore frequency governs memory subsystem performance, including L3 cache latency and bandwidth availability. Existing power governors (e.g., intel_pstate, lunix schedutil) rely on hardware-derived utilization metrics (CPU usage, instruction by cycle) to adjust these frequencies. However, these governors optimize system-level utilization rather than enforcing microservice-level performance requirements (e.g., latency constraints). Consequently, they may over-provision resources (e.g., core/uncore frequency), leading to unnecessary power waste—or under-provision them, resulting in Performance requirement violations. This limitation becomes particularly critical in performance-sensitive microservices deployed in cloud environments.

The problem is exacerbated in multi-tenant nodes where microservices exhibit diverse power–performance sensitivities. CPU-bound services rely on core frequency, while memory-bound services depend heavily on uncore frequency. Furthermore, strictly enforcing CPU core isolation (e.g., via Kubernetes QoS classes) does not isolate shared uncore resources. Consequently, uncore interference—where co-located workloads contend for memory bandwidth—can degrade performance unpredictably. In practice, to mitigate this risk, providers often default to maximum frequency (performance mode), sacrificing energy efficiency for reliability.

Several solutions have been proposed in the literature to address power–performance trade-offs, where Reinforcement Learning (RL) has emerged as a promising approach for adaptive power control. Existing solutions often lack the granularity required for modern cloud environments. Most state-of-the-art RL approaches either control frequencies in isolation, optimize for utilization rather than Performance requirements, or neglect the impact of uncore interference.

To bridge this gap, we propose K8SPI (Kubernetes Power Irrigation), a hierarchical reinforcement learning (HRL) controller that jointly optimizes CPU core and uncore frequencies. When a hosted microservice experiences either a performance requirement violation or significant resource over-provisioning, K8SPI activates a two-stage control process. First, a coarse-grained restoration stage applies large frequency adjustments to rapidly recover performance compliance or eliminate gross over-provisioning. Then, a fine-grained refinement stage performs smaller frequency adaptations to minimize power consumption without triggering new violations or unnecessary resource over-allocation. This design enables K8SPI to provision only the performance needed by each hosted microservice, improving power efficiency while maintaining strict performance requirements.

K8SPI bases its decisions on multi-level telemetry, including per-microservice hardware counters, application-level latency metrics, and socket-level memory and interconnect activity.

By synthesizing these signals, K8SPI effectively manages workload heterogeneity, dynamic load variations, and uncore interference.

We implement and evaluate K8SPI on a dedicated Kubernetes-based testbed that emulates cloud deployment conditions using benchmark-driven microservices. The evaluation covers three scenarios with increasing co-location and contention: (i) a single performance-sensitive microservice, (ii) a single performance-sensitive microservice co-located with best-effort workloads, and (iii) multiple performance-sensitive microservices sharing the same node under high contention.

Across all scenarios, K8SPI achieves a 23%–30% reduction in node-level power compared to the Linux performance governor operating at maximum frequency, while keeping latency violations below 2%–3%, even under dynamic load fluctuations and uncore interference. These results indicate that K8SPI can reduce power waste without compromising performance requirements in multi-tenant, cloud-native deployments.

In summary, the main contributions of this paper are as follows:

- We propose K8SPI (Kubernetes Power Irrigation), a novel Hierarchical Reinforcement Learning (HRL) controller that jointly optimizes CPU core and uncore frequencies to reduce power consumption in cloud-native environments without violating microservice performance requirements.

- We develop an end-to-end framework for the rapid prototyping of RL-based power optimization policies, specifically tailored for latency-sensitive microservices.
- We implement K8SPI within a Kubernetes testbed, evaluating its robustness across increasingly complex scenarios, including isolated execution, best-effort co-location, and high-contention multi-instance deployments of latency-sensitive microservices.
- We demonstrate that K8SPI achieves a 23%–30% reduction in node-level power consumption compared to the default Linux performance governor operating at maximum frequency, while strictly maintaining performance requirement violations below 2%–3% under severe uncore interference and dynamic load fluctuations.

3.2 Background & Motivation

3.2.1 Latency-Sensitive Microservices

Latency-sensitive microservices (e.g., real-time IoT analytics and interactive online services) operate under strict response-time Service Level Agreements (SLAs). In this work, latency is defined as the time elapsed between request arrival at the microservice and generation of the corresponding response. These constraints are strictly enforced because they directly impact user experience and contractual compliance.

Cloud services consume compute/memory resources (CPU, cache, memory bandwidth) in addition to storage/network; this work focuses on compute/memory only.

Microservices exhibit heterogeneous workload characteristics, which we categorize into three classes:

- CPU-intensive (compute-dominated and sensitive to core frequency),
- memory-intensive (memory/bandwidth-dominated and typically more sensitive to uncore frequency), and
- mixed (joint compute and memory demands, sensitive to both core and uncore frequencies).

3.2.2 Understanding Uncore Resource Contention

Uncore resources, including the shared last-level cache (LLC) and memory bandwidth, are shared across CPU cores within a socket. This shared design introduces contention and can degrade performance when multiple memory-intensive microservices are co-located on the same node.

Uncore frequency plays a central role in this interference because it influences the operating point of the memory subsystem and, consequently, the achievable memory bandwidth at both per-core and socket-wide levels. Since

the uncore is shared, bandwidth utilization is inherently interdependent: a core may achieve a high bandwidth in isolation at a given uncore frequency, but its effective throughput is bounded by the socket's aggregate capacity and the concurrent demand from other cores.

For illustration, consider a 16-core server at a baseline uncore frequency with an aggregate socket bandwidth limit of 120 GB/s and an achievable single-core bandwidth of 15 GB/s. A single memory-intensive microservice pinned to one core can reach close to 15 GB/s. Under full co-location, if all 16 cores execute memory-bound workloads simultaneously (co-runners), the aggregate demand exceeds 120 GB/s, and the effective bandwidth per core is reduced (e.g., to approximately 7.5 GB/s). This reduction increases memory stall time and can cause noticeable latency degradation for memory bandwidth-intensive microservices. Fig. 3.1 shows a 21% per-core bandwidth drop under co-location.

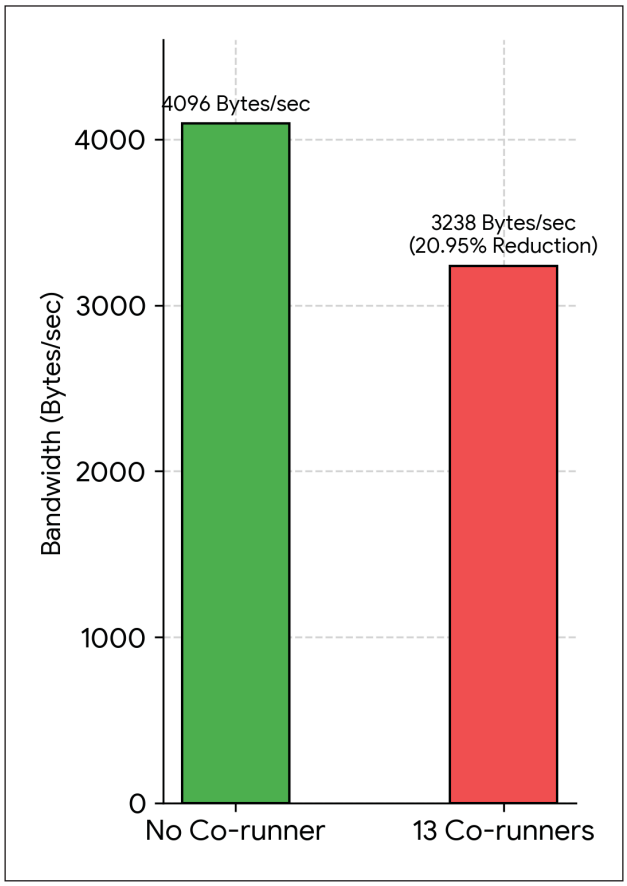


Figure 3.1 Per-core memory bandwidth of a memory-intensive microservice running on a single CPU core in isolation versus co-located with 13 memory-intensive co-runners on a 14-core socket (core/uncore fixed at 2.6/2.4 GHz). Co-runners reduce per-core bandwidth by 21%.

An efficient uncore frequency scaling policy that can scale up frequency under contention to improve bandwidth, and scale down when demand is low to avoid over-provisioning of the resources and reduce power.

3.2.3 Frequency Management in Cloud Servers

Cloud servers manage the performance–power trade-off using DVFS policies, commonly exposed as CPU frequency governors. These policies are typically categorized as static or dynamic.

Static governors (e.g., performance and powersave) operate at fixed operating points. The performance governor keeps frequencies at (or near) their maximum to prioritize performance, which helps performance-sensitive services satisfy latency requirements but often causes power waste through over-provisioning of resources. In contrast, powersave prioritizes low frequencies to reduce power, which can degrade performance and increase the risk of performance requirement violations.

Dynamic governors (e.g., Intel P-state and Linux ondemand/schedutil) adapt frequencies online using utilization-driven heuristics derived from runtime signals. In general, core frequency is increased under higher compute demand and reduced when demand subsides.

3.2.4 Core/Uncore Adjustment Policies

3.2.4.1 Core frequency control

Core DVFS policies typically adjust frequency using system-level indicators of compute demand, such as: CPU load or IPC (or related hardware indicators), where higher instruction throughput can indicate compute-intensive execution and motivate higher frequency.

3.2.4.2 Uncore frequency governor

While our solution targets Intel processors, the proposed methodology is architecture-agnostic and adaptable to other multi-core systems possessing, such as the AMD Zen architecture. The uncore is a substantial resource consumer, occupying roughly 30% of the die area and driving approximately 20% of total power consumption. Uncore frequency policies can be grouped into:

1. CPU load-based governors: On some architectures (e.g., Intel Skylake and early Sapphire Rapids generations), uncore frequency is pushed to its maximum whenever any core reaches its base frequency. This approach ignores actual memory or cache demand. Such a strategy can lead to significant energy waste for CPU-bound applications where the application performance is insensitive to the memory latency.

2. Memory operation-based governors: More recent architectures (e.g., Intel Sierra Forest) introduce activity-driven uncore scaling, adjusting uncore frequency based on memory bandwidth usage and cache access patterns.

3.2.4.3 Limitation: lack of application awareness

Despite these advances, core and uncore governors are primarily driven by system-level signals and do not explicitly incorporate microservice performance requirements (e.g., latency). Consequently, they cannot reliably distinguish between (i) conservative scaling that preserves latency targets while reducing over-provisioning and (ii) aggressive downscaling that saves power but induces performance requirement violations, especially under heterogeneous workloads and multi-tenant interference.

3.2.4.4 Reinforcement Learning

Our work employs RL to develop a dynamic, performance-aware power optimization scheme. RL works by obtaining strategy improvements through continuous interactions with the changing environment in discrete time steps. At each step, an agent receives the current state and a reward (e.g., a function of the measured energy consumption and application-layer performance). It then chooses an action from the set of available actions (e.g., core and uncore frequencies' adjustment) and uses the action to configure the environment (e.g., the hardware). The environment will then move to a new state, and the reward associated with the transition is determined. The goal of the RL agent is to learn a policy that maximizes the expected cumulative reward, i.e., the overall power saving with guaranteed application performance. In this work, we choose to use a policy network optimized via Proximal Policy Optimization (PPO). Given the discrete nature of the action space and the need for fast online adaptation, PPO directly optimizes the policy and gracefully handles noisy environments. Unlike Q-learning-based methods, PPO utilizes incremental, continuous policy updates that ensure smooth adaptation to the evolving workloads and resource contention typical in dynamic cloud environments. This property makes the controller highly robust against runtime anomalies and short-term data drifts.Furthermore, PPO is widely regarded as easier to tune and more stable in practice than many alternative RL methods, making it well suited for reliable online deployment (Eimer, Lindauer & Raileanu, 2023).

3.3 Related Work

Prior work manages the power/energy–performance trade-off using DVFS and UFS (Uncore Frequency Scaling). Early controllers are profile- or rule-based. RL has recently become a promising runtime alternative because it can adapt online under a dynamic environment.

3.3.1 Non-RL Runtime Control

A large body of prior work manages the power/energy–performance trade-off using DVFS and uncore frequency scaling (UFS) via profiling and rule-based runtime control. These methods are effective in structured settings (often HPC) where objectives are typically expressed as energy reduction under a bounded slowdown budget.

Cuttlefish (Nishtala, Petrucci, Carpenter & Sjalander, 2021) profiles MSRs to compute TIPI (memory requests per retired instruction), classifies workloads as compute- vs memory-bound, and triggers an exploration-based linear search to select core/uncore frequencies (CF/UF) minimizing joules per instruction (JPI). On a 20-core Intel Xeon Haswell E5-2650 with ten HPC benchmarks, it reports 19.4% geometric-mean energy savings with 3.6% slowdown versus the performance governor, but its deterministic classification policy can be brittle under phase changes and co-runner interference. DUFP (Guermouche, Étienne André, Trahay & Dulong, 2022) combines package power capping with UFS using runtime FLOPS/s and bandwidth signals to detect phases and adjust RAPL caps within a tolerated slowdown (0–20%), while tuning uncore using the same indicators; on NAS, HPL, and LAMMPS it reports improved energy efficiency, with strong results near a 10% slowdown budget.

Complementary UFS-focused runtimes further validate uncore as a first-order control knob. DUF (Étienne André, Dulong, Guermouche & Trahay, 2022) adapts uncore frequency online to reduce socket power/energy under bounded slowdown, and UPSCavenger (Gholkar, Mueller & Rountree, 2019b) selects uncore frequency in a phase-aware manner for HPC workloads. Collectively, these efforts show that coordinated core–uncore tuning expands the energy–performance trade-off, but the objective is typically slowdown-centric rather than latency-SLO-centric.

For latency-critical services, non-RL controllers shift the objective from slowdown to explicit QoS preservation. PEGASUS (Lo, Cheng, Govindaraju, Barroso & Kozyrakis, 2014) uses feedback control for energy proportionality under service-level constraints; SleepScale (Liu, Draper & Kim, 2014) jointly selects DVFS and sleep states under QoS constraints; TimeTrader (Vamanan, Ansari & Vijaykumar, 2014) leverages tail-latency slack by slowing non-critical requests; and Gemini (Jeong, Kim, Bae & Lee, 2020) uses learning-based prediction for per-query power/frequency decisions under tight latency targets.

3.3.2 RL-Based DVFS/UFS Control Under QoS Constraints

RL has emerged as a promising alternative for DVFS/UFS control under workload non-stationarity and co-location dynamics, because it can learn adaptive policies from telemetry rather than relying on fixed heuristics. In latency-sensitive, multi-workload environments, Twig (Li, Li, Skarlatos & Kozyrakis, 2020) applies deep RL with hardware counters to model QoS behavior and drive energy-efficient control under co-location. Hipster (Lo, Cheng, Govindaraju, Barroso & Kozyrakis, 2017) combines RL with heuristic structure to manage latency-critical

workloads alongside batch co-runners, using DVFS and runtime resource control to reduce energy while preserving QoS and improving throughput. Greeniac (Labassi, Bounour & Boumerdassi, 2019) extends QoS-aware control to fog/edge clusters, framing frequency selection as a bandit problem under response-time constraints.

In service-centric networking, RL is also applied under explicit performance constraints. RL-ADR (Wang, Shi, Lee, Sydir, Zhou, Chi & Li, 2024) selects frequency for a 5G user-plane function to minimize power under near-hard packet-drop constraints, using a policy library and conservative fallback to handle traffic distribution shifts; the authors report zero packet drops on long unseen traces while reducing power versus static baselines. GreenNFV (Nine, Kosar, Bulut & Hwang, 2023) uses actor–critic learning (DDPG) for NFV scheduling and tuning, including CPU frequency among multiple actuators to satisfy performance requirement/throughput targets while improving energy efficiency.

A second cluster shows the benefit of joint core–uncore control. Juan and Marculescu (Juan & Marculescu, 2012) study reinforcement-based policies over core and uncore DVFS under power constraints, showing coordinated policies outperform core-only or uncore-only scaling under iso-power conditions. PManager (Wang, Yang, Zhou & Wu, 2022a) co-optimizes power caps and UFS across phases using RL, and Gocht et al. (Gocht, Schöne & Bielert, 2019) explore the core×uncore space using Q-learning-style exploration with region awareness and direct energy feedback. A hierarchical RL design is also proposed in patent literature (Zhu et al., 2024), but it lacks peer-reviewed evaluation. Finally, embedded platforms study RL-based DVFS under deadline/performance constraints (e.g., Q-learning in (Biswas, Ivanov, Ayala, Benini, Macii, Jantsch & Ros, 2017) and DDQN with Linux CPUFreq in (Li, Zhou & Liao, 2023)). Despite this progress, many studies still assume isolation or whole-socket occupancy. This abstraction does not match cloud nodes where multiple services share a socket and compete for uncore resources. Our objective is to satisfy per-service latency SLAs while minimizing power by dynamically scaling core and uncore frequencies under co-location, heterogeneous workload types, and time-varying load.

3.4 K8SPI Architecture

3.4.1 System Overview

In this paper, we propose K8SPI, an online, hierarchical reinforcement learning (HRL) based power governor tailored for multi-tenant cloud environments. Specifically, K8SPI targets heterogeneous microservices deployed via Kubernetes (K8s) under the Guaranteed Quality of Service (QoS) class, a configuration that allocates dedicated CPU cores to individual microservices. The primary objective of the K8SPI framework is to minimize the CPU socket power consumption without incurring performance degradation that violates the performance requirements of the hosted microservices. As illustrated in Fig. 3.2, the K8SPI control loop consists of two primary modules: a multi-layered Metrics Monitoring module and an HRL-based Power Governor.

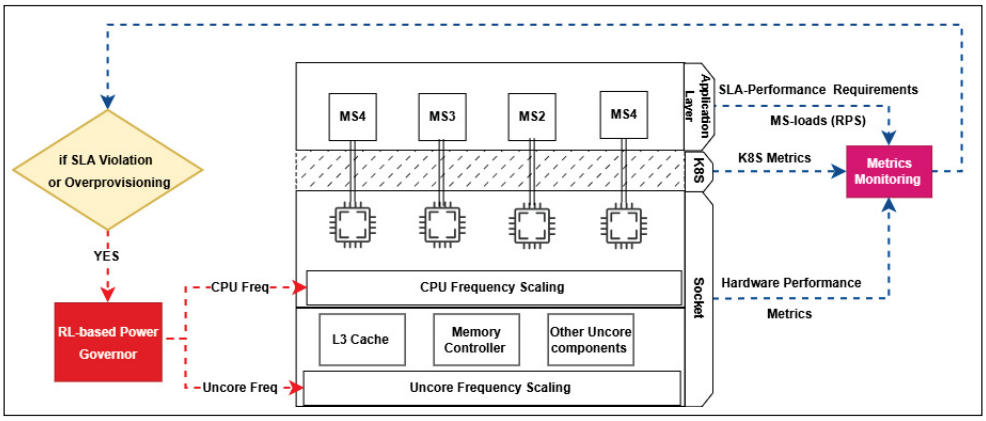


Figure 3.2 K8SPI architecture overview.

3.4.2 Multi-Layered Telemetry and Monitoring

To accurately capture the dynamic behavior of the system, the continuous monitoring module aggregates telemetry data across three distinct abstraction layers, providing both a global system view (i.e., socket-level view) and per-microservice visibility using:

- **Application Layer:** Captures high-level performance metrics (e.g., target latency) and dynamic microservice loads, measured in requests per second (RPS) for each microservice.
- **Orchestration (K8s) Layer:** Collects container- and pod-level metrics, including resource usage, request-s/limits, node placement, and CPU-core assignment, to track deployment constraints and map cores to hosted microservices.
- **Hardware (Socket) Layer:** Collects lightweight, low-level hardware performance counter measurements (e.g., instructions per cycle, cache miss rates, and memory bandwidth utilization) at both the socket and core levels.

3.4.3 Event-Triggered Hierarchical Power Governor

K8SPI operates through an event-driven control loop that continuously evaluates the current system state using the aggregated telemetry collected from the monitoring stack. At each decision point, the monitored metrics are compared against the predefined performance thresholds to determine whether corrective action is required. As illustrated in Fig. 3.2, the RL-based Power Governor is triggered only when the monitoring subsystem detects either an imminent latency violation or a condition of significant resource overprovisioning.

Once such a condition is detected, the HRL-based Power Governor adjusts two hardware control knobs: the frequency of the CPU core dedicated to the target microservice, since the microservice is pinned to a single core,

and the shared uncore frequency of the processor socket. This design allows K8SPI to react only when necessary, thereby reducing unnecessary control actions while maintaining performance requirements and improving power efficiency.

To efficiently navigate this expansive, multi-dimensional configuration space, the K8SPI governor relies on a two-stage hierarchical RL architecture:

- **Stage 1 - Coarse Agent:** This agent makes large, rapid adjustments to the frequency settings. Its primary goal is to quickly mitigate performance violations or eliminate massive overprovisioning, bringing the system into a safe operational region. In doing so, it creates an optimal starting point for the subsequent tuning phase.
- **Stage 2 - Fine Agent:** Once the system is stabilized near the safe threshold, the fine agent takes over. It performs smaller, granular frequency refinements to minimize power consumption while ensuring that the performance requirement remains strictly satisfied.

3.5 System Model and Problem Formulation

3.5.1 Socket-Level Orchestration Architecture

We consider a multi-tenant cloud compute node comprising multiple physical processor sockets; K8SPI acts at the socket level. The socket encompasses a set of physical CPU cores, denoted by C, and hosts a set of heterogeneous, latency-critical microservices, $\mathcal{M}$. To guarantee strict compute isolation and satisfy Kubernetes Guaranteed Quality of Service (QoS) requirements, we assume an exclusive deployment mapping where each microservice $m_i \in \mathcal{M}$ is pinned to a dedicated core $c_i \in C$.

The hardware exposes two primary control domains for dynamic voltage and frequency scaling (DVFS), and uncore frequency scaling (UFS). First, each core c_i operates at an independent core frequency, $f_{c,i}$, bounded by $[f_c^{min}, f_c^{max}]$. Second, all cores within the socket share a common uncore subsystem, which includes the Last Level Cache (LLC) and memory controllers. This shared domain operates at a global uncore frequency, f_u, bounded by $[f_u^{min}, f_u^{max}]$.

3.5.2 Workload and Performance Model

The system operates under an event-triggered control paradigm. Let t_k denote the timestamp of the k-th control invocation epoch, triggered by either an imminent performance violation or a state of massive resource overprovisioning.

Each microservice m_i is characterized by an architectural workload profile, ω_i (e.g., compute-bound or memory-bound), and a dynamic normalized workload intensity, $\lambda_i(t_k) \in [0, 1]$. The variable $\lambda_i(t_k)$ does not represent the request arrival rate; instead, it captures the normalized amount of application-specific work associated with each request, relative to the maximum workload profiled for microservice m_i. The performance of m_i is strictly governed by a target latency constraint, $L_{\text{target},i}$.

For example, for a CPU-bound microservice wrapping a `stress-ng` CPU stressor, the per-request workload can be parameterized by the number of bogo-operations executed by that request, where bogo-operations are a workload unit used by `stress-ng`. Let $\lambda_i^{\max}$ denote the maximum profiled workload for microservice m_i. The normalized load is then given by

$$\lambda_i(t_k) = \frac{\lambda_i^{\text{raw}}(t_k)}{\lambda_i^{\max}}, \tag{3.1}$$

where $\lambda_i^{\text{raw}}(t_k)$ is the raw request-level workload parameter. Hence, $\lambda_i(t_k) = 0$ denotes no effective work, whereas $\lambda_i(t_k) = 1$ denotes the maximum profiled workload.

The performance of m_i is strictly governed by a target latency constraint, L_i^*.

The measured tail latency at epoch t_k, denoted as $L_i(t_k)$, is modeled as the sum of its isolated execution latency and a contention time penalty:

$$L_i(t_k) = \Phi\left(f_{c,i}(t_k), f_u(t_k), \lambda_i(t_k), \omega_i\right) + \Delta L_{interf,i}(t_k) \tag{3.2}$$

where $\Phi(\cdot)$ represents the latency function of the microservice executed without noisy neighbors. The term $\Delta L_{interf,i}(t_k) \geq 0$ encapsulates the latency time penalty induced by uncore interference.

Crucially, because $\Phi(\cdot)$ and $\Delta L_{interf,i}$ depend on highly dynamic runtime behaviors, opaque microarchitectural state interactions (e.g., core pipeline stalls and cache hierarchy thrashing), and complex resource contention across the entire socket, deriving an exact analytical model is computationally difficult. Instead, these highly dynamic and complex uncore interference must be approximated and learned iteratively at runtime.

3.5.3 Power Model

The total power consumption of the socket, $P_{socket}(t_k)$, is decomposed into the dynamic power of the active cores denoted as P_{dyn}^c, the dynamic power of the shared uncore domain denoted as P_{dyn}^u, and the baseline static leakage power (P_{static}):

$$P_{socket}(t_k) = \sum_{i \in C} P^{c}_{dyn}(f_{c,i}(t_k), \lambda_i(t_k)) + P^{u}_{dyn}(f_u(t_k), \vec{\lambda}(t_k)) + P_{static} \tag{3.3}$$

Under core/uncore frequency scaling, the dynamic component changes strongly with voltage/frequency, whereas the static component is primarily governed by leakage, supply voltage, and temperature; therefore, static power is often modeled as frequency-independent or only weakly/indirectly frequency-dependent. Consequently, it is often treated as approximately constant with respect to frequency (Goel & McKee, 2016).

Similar to the latency model, explicitly formalizing the complex, non-linear coupling between highly dynamic multi-core workloads, joint frequency scaling, and holistic socket power consumption across diverse hardware is practically infeasible at runtime. Therefore, the exact power-performance manifold is treated as a highly dynamic, black-box environment whose underlying transition dynamics will be implicitly learned via the Deep Reinforcement Learning policy.

3.5.4 Problem Definition

The objective of the proposed K8SPI framework is to determine the optimal frequency scaling actions at each event epoch t_k to minimize the total socket power consumption, while strictly guaranteeing that no microservice violates its performance requirement. The constrained optimization problem is formulated as:

$$\min_{\vec{f_c}, f_u} \sum_{k=0}^{K} P_{socket}(t_k) \tag{3.4}$$

$$\text{subject to} \quad L_i(t_k) \leq L_i^* \quad \forall m_i \in \mathcal{M}, \forall k \tag{3.5}$$

$$f_c^{min} \leq f_{c,i}(t_k) \leq f_c^{max}, \quad f_u^{min} \leq f_u(t_k) \leq f_u^{max} \tag{3.6}$$

This formulation highlights the core control challenge: minimizing Equation 3.3 by lowering frequencies directly penalizes the latency function in Equation 3.2, thereby risking violations of the strict constraint in Equation 3.5 amid unpredictable uncore interference.

Because the functions governing power (Eq. 3.3) and latency (Eq. 3.2) exhibit complex, non-linear, and highly dynamic behaviors at runtime across both core and uncore domains, traditional convex optimization and static heuristic control are inadequate. Consequently, we frame this optimization as a Markov Decision Process (MDP). K8SPI leverages a Deep Reinforcement Learning agent to map real-time telemetry states to optimal frequency

actions, inherently learning a robust control policy capable of navigating the complex trade-offs between strict performance requirement enforcement and maximum energy efficiency.

3.6 Hierarchical Reinforcement Learning Controller

To navigate the high-dimensional joint frequency space efficiently, we propose a Hierarchical Reinforcement Learning (HRL) architecture. The hierarchical approach decomposes the DVFS adjustment into two sequential decisions.

By splitting the action into coarse and fine stages, the hierarchical controller reduces the search space and explicitly separates the objectives of rapid latency stabilization and fine-grained power minimization. Triggered whenever a performance requirement violation or severe resource over-provisioning crosses a predefined threshold, the control loop operates over a fixed two-step episode. Agent A1 (coarse) first applies a large frequency adjustment to promptly drive the system out of violation or mitigate gross over-provisioning, thereby moving it to a safe and feasible operating region. Agent A2 (fine) then refines A1's decision through a small frequency adjustment to safely improve power efficiency.

The time window separating the actions of A1 and A2 is fixed for all episodes. It is determined by the sampling interval of the monitoring stack, so that each agent acts on freshly updated telemetry. For example, in a Prometheus-based monitoring stack, this interval can be aligned with the configured scrape interval (e.g., 30 s in our setup).

3.6.1 State Representation and Observations

At each decision step, the active agent (A1 at s_1, A2 at s_2) receives the same structured observation (see table 3.1) composed of three components:

- **Service-level vector (x_s):** captures microservice-specific signals from both hardware and application telemetry, such as IPC, per-core bandwidth, and current request latency. This component characterizes the local behavior of the target microservice and primarily guides core-frequency scaling decisions.
- **Socket-level vector (x_u):** captures aggregated socket-wide counters that reflect shared-resource pressure and the influence of concurrent co-located microservices. This component provides a global view of uncore activity and primarily guides uncore-frequency scaling decisions.
- **Action mask (m):** Identifies infeasible actions and prevents the agent from selecting frequency settings outside the supported hardware operating range, as formulated in constraint 3.6.

Table 3.1 lists the metrics used in the observation space. These metrics capture the main hardware-level factors affecting microservice performance, enabling the RL agent to identify the runtime bottleneck (e.g., compute-bound vs. memory-bound) and adjust core and uncore frequencies accordingly.

Table 3.1 RL Agent Observation Space Metrics

Metric Name	**Description**
Service-Level Metrics (Used to guide core)	
Core_L3_Misses	L3 cache misses on the service core
Core_L3_Hits	L3 cache hits on the service core
Core_Mem_Bandwidth	Local memory bandwidth used by the service
Service_Avg_Latency	Average latency of service requests
Service_Target_Latency	Target latency of service requests
Core_MPKC	Misses per thousand instructions
Core_IPC	Instructions per cycle
Core_KIPS	Kilo Instructions per Second
CPU_Time_System	Time spent in system/kernel mode
CPU_Time_User	Time spent in user mode
CPU_Utilization	Container's total CPU usage (%)
Core_Frequency	Core frequency assigned to the service
Core_L3_Occupancy	L3 cache usage by the service core
Socket-Level Metrics (Used to guide uncore)	
Socket_L3_Misses	Aggregate L3 cache misses on the socket
Socket_L3_Hits	Aggregate L3 cache hits on the socket
Socket_Mem_Bandwidth	Total memory bandwidth on the socket
Socket_L3_Occupancy	Overall L3 cache occupancy
Socket_Power_CPU	Package-level CPU power consumption (W)
Socket_Power_DRAM	DRAM power consumption (W)
Socket_Uncore_Frequency	Current uncore frequency on the socket

Frequency-Conditioned State Normalization:

Standard global normalization fails to capture how physical hardware boundaries shift under Dynamic and Frequency Scaling. To ensure stable convergence and explicitly embed domain knowledge, we introduce a frequency-conditioned online normalization strategy. The environment maintains independent running statistics (mean μ, standard deviation σ) for each unique core and uncore frequency pair (f_c, f_u). The telemetry vector x is

normalized via an online z-score strictly within its active frequency bucket:

$$x^{(\text{norm})} = \frac{x - \mu_{(f_c, f_u)}}{\sigma_{(f_c, f_u)} + \xi} \tag{3.7}$$

This bucketed approach enables the RL agent to intelligently isolate architectural bottlenecks. If compute-related metrics (e.g., Kilo Instruction Per Second (KIPS)) yield high positive z-scores during an performance requirement violation, the agent infers core saturation and scales up f_c. Conversely, saturating memory metrics (e.g., bandwidth) prompts the agent to scale f_u. While this accurately captures regime-specific hardware limits, it introduces a minor cold-start phase where newly visited frequency pairs require a brief initialization window to stabilize their statistics.

3.6.2 Hierarchical Action Space

A flat joint controller over core and uncore frequencies induces a large discrete action space, since all supported core–uncore frequency pairs must be enumerated. In our setting, this would result in approximately 250–300 joint actions, which increases exploration complexity and slows convergence. To address this, we decompose control into two sequential stages: a coarse correction stage followed by a fine refinement stage.

Let δ_f denote the minimum hardware-supported frequency transition step. This quantity is platform dependent and captures the native granularity of the processor frequency interface. Defining the action space relative to δ_f makes the formulation portable across processors with different discrete frequency resolutions. On our platform, $\delta_f = 0.2\,\text{GHz}$.

- **Coarse stage (Agent A1):** selects a large corrective delta

$$\Delta f^{A1} \in \{-4\delta_f,\ 0,\ +4\delta_f\},$$

  which corresponds to $\{-0.8, 0, +0.8\}$ GHz on our hardware. Applied jointly to the core and uncore domains, this yields $3 \times 3 = 9$ actions.

- **Fine stage (Agent A2):** refines the coarse decision using smaller deltas

$$\Delta f^{A2} \in \{-2\delta_f,\ -\delta_f,\ 0,\ +\delta_f,\ +2\delta_f\},$$

  which corresponds to $\{-0.4, -0.2, 0, +0.2, +0.4\}$ GHz on our hardware. Applied jointly to the core and uncore domains, this yields $5 \times 5 = 25$ actions.

This hierarchy creates a clear separation between rapid correction and local refinement. Agent A1 performs large frequency moves to quickly exit severe under-provisioning or over-provisioning regions, while Agent A2 performs one- and two-step refinements to settle near the target operating point with finer control. Importantly, the combined two-stage decision yields a dense reachable set around the current state:

$$\Delta f^{\text{total}} = \Delta f^{A1} + \Delta f^{A2} \in \{-6, -5, \ldots, 5, 6\}\delta_f, \tag{3.8}$$

meaning that any net change within a $\pm 6\delta_f$ neighborhood can be realized in one two-stage episode at the hardware's native frequency resolution.

For both frequency domains, the selected action is applied as a discrete delta to the current operating point:

$$f_{\text{c}}^{\text{new}} = f_{\text{c}} + \Delta f_{\text{c}}, \qquad f_{\text{u}}^{\text{new}} = f_{\text{u}} + \Delta f_{\text{u}}. \tag{3.9}$$

3.6.3 performance Gap Quantization

To stabilize the learning process and prevent control jitter, the performance state of each microservice is evaluated via a quantized latency gap. First, the raw relative latency gap for a given microservice is defined as:

$$g_{raw} = \frac{L - L^*}{L^*} \tag{3.10}$$

where L is the current measured average latency and L^* is the strict target latency constraint. Under this formulation, a positive value indicates an performance requirement violation, whereas a negative value indicates resource overprovisioning.

Crucially, the magnitude of this raw gap serves as the primary trigger for corrective action. The control loop is only invoked when the performance deviates beyond an acceptable boundary—for instance, an acceptable latency degradation threshold (e.g., $g_{raw} > +15\%$) or a severe overprovisioning threshold (e.g., $g_{raw} < -15\%$).

Rather than feeding the continuous, noisy g_{raw} directly to the RL agent, the environment maps it to a discrete, bucketed gap $g(L, f_c, f_u)$ using a three-tiered logic:

$$g(L, f_c, f_u) = \begin{cases} 0, & \text{if hardware-saturated} \\ 0, & \text{if } |g_{raw}| < \epsilon \\ \text{sgn}(g_{raw})\delta\lceil |g_{raw}|/\delta \rceil, & \text{otherwise} \end{cases} \tag{3.11}$$

This quantization strictly enforces three operational rules to ensure predictable power transitions:

1. **Deadzone (ϵ):** A tight tolerance band (e.g., $\tau = 5\%$) masks small, transient latency fluctuations. This prevents jitter and unnecessary frequency toggling when the system is operating near the target.

2. **Step-Based Scaling (δ):** The gap is discretized into fixed intervals (e.g., $\delta = 10\%$) to group state observations into distinct buckets, rendering the optimization landscape more stable for the agent.

3. **Hardware-Aware Safety Cap:** The gap is forced to zero if the system is hardware-saturated. Specifically, this occurs if the performance requirement is satisfied but frequencies are already at their lowest supported bounds ($L \leq L^* \wedge f_c = f_{c,min} \wedge f_u = f_{u,min}$), or if the performance requirement is violated but frequencies are already maxed out ($L \geq L^* \wedge f_c = f_{c,max} \wedge f_u = f_{u,max}$). This hardware awareness stops the controller from attempting physically impossible adjustments.

3.6.4 Hierarchical Reward and Credit Assignment

The reward signals are engineered to encode both strict performance requirement constraint satisfaction and power efficiency. The hierarchical controller operates over a fixed two-step episode:

$$s_0 \xrightarrow[\Delta f_c^{(1)}, \Delta f_u^{(1)}]{\text{A1 (coarse)}} s_1 \xrightarrow[\Delta f_c^{(2)}, \Delta f_u^{(2)}]{\text{A2 (fine)}} s_2 \tag{3.12}$$

where Agent A1 performs a coarse move transitioning the system to intermediate state s_1, and Agent A2 refines it to reach the terminal state s_2. Let g_0, g_1, and g_2 represent the quantized performance gaps at the pre-A1, post-A1, and terminal stages, respectively.

3.6.4.1 Universal Performance Potential

To evaluate performance requirement compliance consistently across both hierarchical stages, we define a universal performance potential function $\Phi(g)$ based on the quantized gap:

$$\Phi(g) = \begin{cases} -(2+g), & g > \epsilon \quad \text{(violation)} \\ 1 - |g|, & g \leq \epsilon \quad \text{(safe)} \end{cases} \tag{3.13}$$

where ϵ represents the SLA safety threshold (e.g., $\epsilon = 0.05$). This piecewise function heavily penalizes performance requirement relatively to variation magnitude (g) while rewarding proximity to the target latency when operating in the safe zone.

3.6.4.2 Etiquette Penalty

To embed domain knowledge and prevent unsafe throttling, an etiquette penalty applies at both steps $s \in \{1, 2\}$. It strictly penalizes the active agent for decreasing both frequencies if the system is already in a state of performance requirement violation:

$$\psi_t = \eta_{\text{both}} \cdot \mathbb{I}\left[g_{t-1} > 0 \wedge \Delta f_c^{(s)} < 0 \wedge \Delta f_u^{(s)} < 0\right] \qquad (\text{with } \eta_{\text{both}} = 0.5) \tag{3.14}$$

3.6.4.3 Agent A2 (Fine) Reward Function

The goal of the fine agent is to refine the frequencies to minimize power once the system is near or within the safe region. Its total reward, R_{A2}, evaluates the terminal performance state g_2 and adds an efficiency bonus R^* if the SLA is satisfied, minus its step-specific etiquette penalty:

$$R_{A2} = \Phi(g_2) + \mathbb{I}[g_2 \leq \epsilon] \cdot R^* - \psi_2 \tag{3.15}$$

To explicitly favor low-power operating points, each discrete hardware frequency is mapped to a normalized index $a_c, a_u \in [0, 2]$, where 0 is the minimum supported frequency and 2 is the maximum. The efficiency bonus is defined as:

$$R^* = w_c(1 - a_{c,2}) + w_u(1 - a_{u,2}) \tag{3.16}$$

When the performance requirement is met ($g_2 \leq \epsilon$), the agent balances two competing objectives. The performance term $\Phi(g_2) = 1 - |g_2|$ encourages the agent to operate closely to the target latency, minimizing unnecessary performance slack and overproviding. Concurrently, the efficiency bonus R^* drives power reduction. Because dynamic power scales directly with operating frequency, core and uncore frequencies serve as reliable proxies for power consumption. The normalization mapping of core/uncore frequencies to $[0, 2]$ ensures that selecting the minimum supported frequency yields a maximum positive reward (e.g., $1 - 0 = 1$), whereas operating at the maximum frequency introduces a penalty (e.g., $1 - 2 = -1$). Furthermore, we configure the weights such that $w_c < w_u$; consequently, raising the core frequency is less penalized than raising the uncore frequency, accurately reflecting the higher power cost of the socket-wide uncore domain.

3.6.4.4 Agent A1 (Coarse) Reward Function

The primary goal of the coarse agent is to rapidly reduce the latency gap and create an optimal, safe starting point for A2's fine-tuning. Because A1 transitions the system to the intermediate state s_1, its immediate performance reward is $\Phi(g_1)$.

To ensure cohesive teamwork and prevent A1 from acting greedily, A1's total reward directly incorporates A2's terminal outcome, while subtracting its own step-specific etiquette penalty (ψ_1):

$$R_{A1} = \Phi(g_1) - \psi_1 + R_{A2} \tag{3.17}$$

This elegantly coupled design forces A1 to select coarse actions that not only immediately improve the SLA state ($\Phi(g_1)$) but also position the intermediate state such that A2 can successfully minimize energy without triggering a terminal violation (R_{A2}).

3.6.5 Runtime Orchestration and Action Aggregation

In a live production environment, K8SPI operates as an event-triggered control loop monitoring the performance gaps of all co-located microservices on a shared socket. When a specific microservice $m_i \in \mathcal{M}$ actively violates its performance requirement/severely over-provisioned ($g_i > +/-20\%$), the RL controller proposes a corrective joint frequency action ($f_{\mathrm{c},i}$, $f_{\mathrm{u},i}$) per microservice. However, the physical hardware imposes distinct granularity constraints for applying these independent per-service actions.

When come to the proposed core frequency is applied directly and independently to the specific core hosting m_i (as microservices are pinned to a single cpu core):

$$f_{\mathrm{c}}(c_i) = f_{\mathrm{c},i} \tag{3.18}$$

Conversely, the uncore subsystem is a shared, socket-wide resource. To resolve conflicting uncore frequency proposals among co-located services while ensuring the performance requirements of all microservices are met, the final uncore frequency f_{u}^* applied to the socket is the maximum of all individual proposals:

$$f_{\mathrm{u}}^* = \max_{m_i \in \mathcal{M}} \{f_{\mathrm{u},i}\} \tag{3.19}$$

This pessimistic aggregation guarantees that the most memory-constrained service is strictly protected against uncore throttling, while node-wide power savings are successfully captured only when all co-located services mutually tolerate a scaled-down uncore frequency.

3.7 Scalable Implementation and Training Methodology

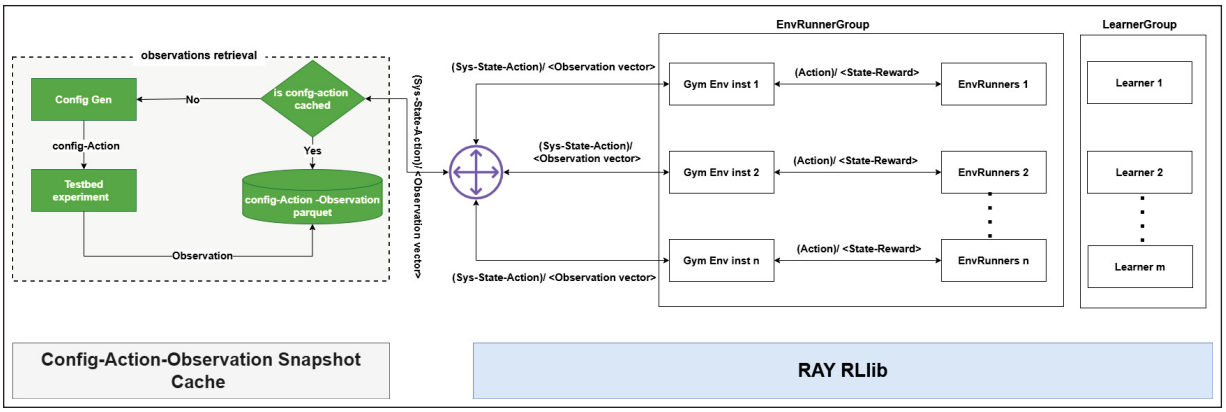


Figure 3.3 Fast RL Prototyping Framework Architecture Overview.

Hardware-in-the-loop RL is slowed by multi-second feedback: each frequency action must be applied and then observed through the monitoring stack (e.g., Prometheus) before the next step. Since RL needs thousands of trials, this makes training and iterative tuning (reward design, hyperparameters) prohibitively slow on a live Kubernetes node.

Distributed Training and Snapshot Caching:

To accelerate learning, K8SPI uses Ray RLlib to enable distributed training with a centralized learner coordinating multiple concurrent environment runners. However, running many environments is costly because each environment ultimately requires execution on the physical node (frequency actions are hardware operations and cannot be virtualized). K8SPI resolves this by inserting an offline Config–Action–Observation Snapshot Cache. Fig. 3.3 shows the architecture of the fast RL prototyping framework. When an RL agent instance encounters a new $\langle \text{state}, \text{action} \rangle$ pair, the framework performs a one-time hardware profiling run: it deploys the state (the specific microservice executing its corresponding benchmark under a defined uncore stress level), applies the selected frequency configuration on the testbed, and collects telemetry over a sustained 5-minute window to average out transient effects. The resulting high-fidelity observation vector is stored in a Parquet-backed datastore. Future visits to the same $\langle \text{state}, \text{action} \rangle$ pair are served instantly from the cache, enabling safe parallel RLlib training without live-hardware latency.

3.8 Experimental Methodology

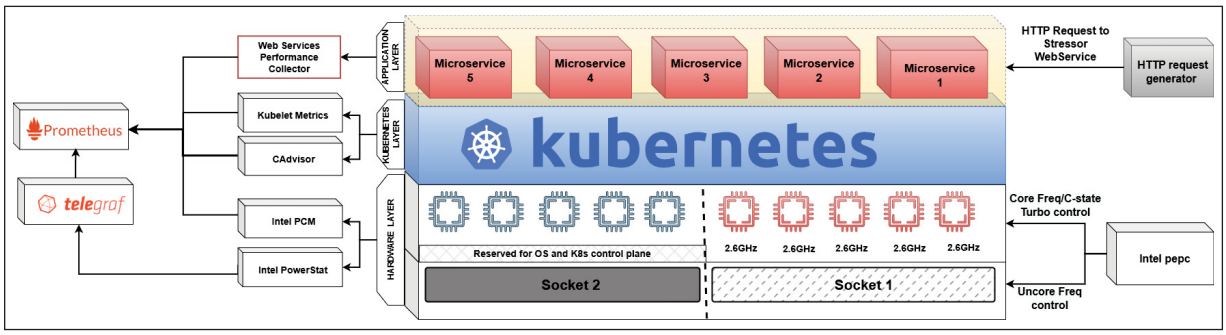


Figure 3.4 Testbed Architecture Overview.

Testbed Infrastructure and Software Stack:

Table 3.2 Testbed Hardware and Software Configuration

Attribute	Specification	Attribute	Specification
CPU Model	Intel Xeon Gold 6132	Threads/Core	1 (HT Disabled)
Cores/Socket	14	Sockets	2
NUMA Nodes	2	Core Freq (GHz)	1.0 – 2.6
L3 Cache	38.5 MB/socket	Uncore Freq (GHz)	1.2 – 2.4
OS	Ubuntu 22.04.5	Kubernetes QoS	Guaranteed
Turbo & C-States	Disabled	Memory Swap	Disabled
Core Governor	`userspace`	Uncore Governor	`userspace`

To ensure stable, reproducible, and noise-free Reinforcement Learning observations and power measurements, we constructed a strictly controlled, dual-socket Kubernetes testbed wchich it congration illustrated in Table II-2 and in Fig 3.4.

To prevent background OS noise from corrupting the RL agent's telemetry or power measurements, we enforced strict hardware-level isolation. Socket 1 was dedicated to executing the experimental microservices and the telemetry stack. Socket 2 was reserved entirely for the Kubernetes control plane and operating system background processes. Moreover, to eliminate unpredictable power fluctuations, several BIOS and kernel-level features were explicitly disabled (see Table II-2). The telemetry stack utilized Intel PCM for hardware performance counters, Intel PowerStat (i.e., Intel RAPL) for socket-level ground-truth power measurements, and cAdvisor alongside Kubelet metrics for per-microservice resource usage and core-pinning resolution. Finally, we employed custom scripts to measure the end-to-end latency per HTTP request for each hosted service, establishing the application-level performance of each microservice. Data was scraped and aggregated using a telemetry stack comprising Telegraf and Prometheus. To prevent load-generation overhead from interfering with the testbed, the traffic generators were

hosted remotely and triggered the microservices via HTTP. To control the core and uncore frequencies during the experiments, we utilized the open-source Intel `pepc` library, which provides a sophisticated power control toolkit.

3.8.1 Benchmarks and Workloads

We construct a benchmark-driven microservice corpus using two micro-benchmark suites: Stress-NG (primarily CPU-bound and mixed behaviors) and Parallel Memory Bandwidth (PMBW) (memory-bound behaviors with strong sensitivity to the shared memory subsystem and uncore frequency). Each selected benchmark is containerized and deployed as a stateless HTTP microservice with fixed resource allocations (1 CPU and 2 GB memory). Each request triggers a bounded unit of benchmark work executed at a controlled load level. For each benchmark, we define 4–5 discrete load settings (e.g., a fixed number of memory-copy operations) to expose different demand workload levels. To separate learning and evaluation from final testing, we create two disjoint datasets. Dataset

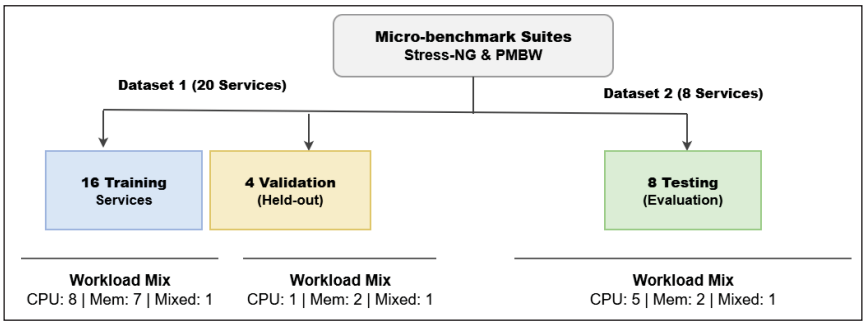


Figure 3.5 Micro-benchmark Dataset Split for Model Development and Evaluation.

1 contains 20 services, split into 16 for training and 4 held out for RL model evaluation. Dataset 2 contains 8 additional services reserved exclusively for final testing and validation. Fig. 3.5 illustrates the dataset partitioning across the training, evaluation, and test phases.

Crucially, each dataset incorporates a diverse mix of CPU-bound, memory-bound, and mixed-workload microservices. Each workload class exhibits a unique power and performance profile in response to independent core and uncore frequency scaling.

We classify benchmark services using IPC measured at the maximum core frequency as a compact microarchitectural proxy for workload behavior (Intel Corporation, 2022). The rationale is that IPC captures how effectively a workload converts processor cycles into retired instructions: compute-intensive kernels typically sustain high retirement rates, whereas memory-intensive kernels experience frequent backend stalls due to long-latency data accesses. Accordingly, we define three workload classes: *CPU-bound* for IPC ≥ 1.0, *mixed* for $0.4 \leq$ IPC < 1.0, and *memory-bound* for IPC < 0.4. This choice avoids relying solely on benchmark names and instead provides a

Table 3.3 Illustrative IPC-based classification of the development dataset.

Service	Target Latency (s)	Illustrative IPC	Class
Training Microservices			
BG_CPUFFT	0.88	2.90	CPU-bound
BG_Memcpy	1.60	2.90	CPU-bound
BG_CPUBitops	1.59	2.50	CPU-bound
BG_CPUInt64	0.77	2.50	CPU-bound
BG_Matrix	0.65	1.30	CPU-bound
BG_Vecmath	1.03	1.10	CPU-bound
BG_Lockbus	1.07	0.60	Mixed
BG_MemrateFlush	48.11	0.30	Memory-bound
BG_ScanRead64PtrSimpleLoop	9.57	0.35	Memory-bound
BG_ScanWrite64PtrSimpleLoop	9.15	0.30	Memory-bound
BG_ScanWrite128PtrSimpleLoop	9.19	0.25	Memory-bound
BG_ScanWrite256PtrSimpleLoop	9.12	0.20	Memory-bound
BG_ScanWrite128PtrUnrollLoop	9.12	0.18	Memory-bound
BG_ScanRead256PtrSimpleLoop	8.67	0.15	Memory-bound
BG_ScanRead256PtrUnrollLoop	8.63	0.15	Memory-bound
BG_ScanWrite256PtrUnrollLoop	9.15	0.15	Memory-bound
Evaluation Microservices			
BG_CPUAckermann	3.08	3.00	CPU-bound
BG_Memthrash	3.79	0.30	Mixed
BG_ScanWrite64PtrUnrollLoop	9.15	0.25	Memory-bound
BG_ScanRead128PtrUnrollLoop	8.85	0.20	Memory-bound

uniform, measurement-driven classification rule across both Stress-NG and PMBW services. Tables 3.3 and 3.4 further report the workload category of each benchmark and its allocation across the training, validation, and testing datasets.

It is important to note that the lower threshold of IPC = 0.4 was selected empirically as a conservative boundary for the *memory-bound* region in our benchmark set. Specifically, an IPC below 0.4 means that the workload achieves less than 40% of the reference throughput of one retired instruction per cycle, i.e., it loses more than 60% of that baseline execution progress. In our controlled setting, such a large loss in retirement efficiency indicates that execution is no longer primarily driven by computation, but is instead dominated by waiting effects from the memory hierarchy (Yasin, 2014). For this reason, we use IPC < 0.4 to mark clearly memory-dominated workloads, while the interval $0.4 \leq$ IPC < 1.0 is reserved for intermediate cases that still exhibit a mixed behavior.

We illustrate the impact of core and uncore frequency scaling on performance (microservice latency) and power consumption (package and DRAM power) across three representative benchmarks: a CPU-bound (`BG_CPUAckermann`) in Fig. 3.6, a memory-bound (`BG_MemrateFlush`) in Fig. 3.7, and a mixed (`BG_`–

Table 3.4 Illustrative IPC-based classification of the test dataset.

Service	Target Latency (s)	Illustrative IPC	Class
Testing Microservices			
STRESS_MEM_Vm	1.60	2.90	CPU-bound
STRESS_CPU_Longdouble	0.85	2.80	CPU-bound
STRESS_CPU_Div64	0.80	2.60	CPU-bound
STRESS_CPU_Euler	0.63	1.30	CPU-bound
STRESS_CPU_Float	1.01	1.10	CPU-bound
STRESS_MEM_Atomic	1.10	0.55	Mixed
PMBW_ScanRead64IndexSimple	9.64	0.25	Memory-bound
PMBW_ScanWrite64IndexUnroll	9.00	0.20	Memory-bound

`Memthrash`) microservice in Fig. 3.8. To isolate the specific impact of core frequency scaling, we fix the uncore frequency to its minimum. Similarly, we lock the core frequency at its minimum when varying the uncore frequency.

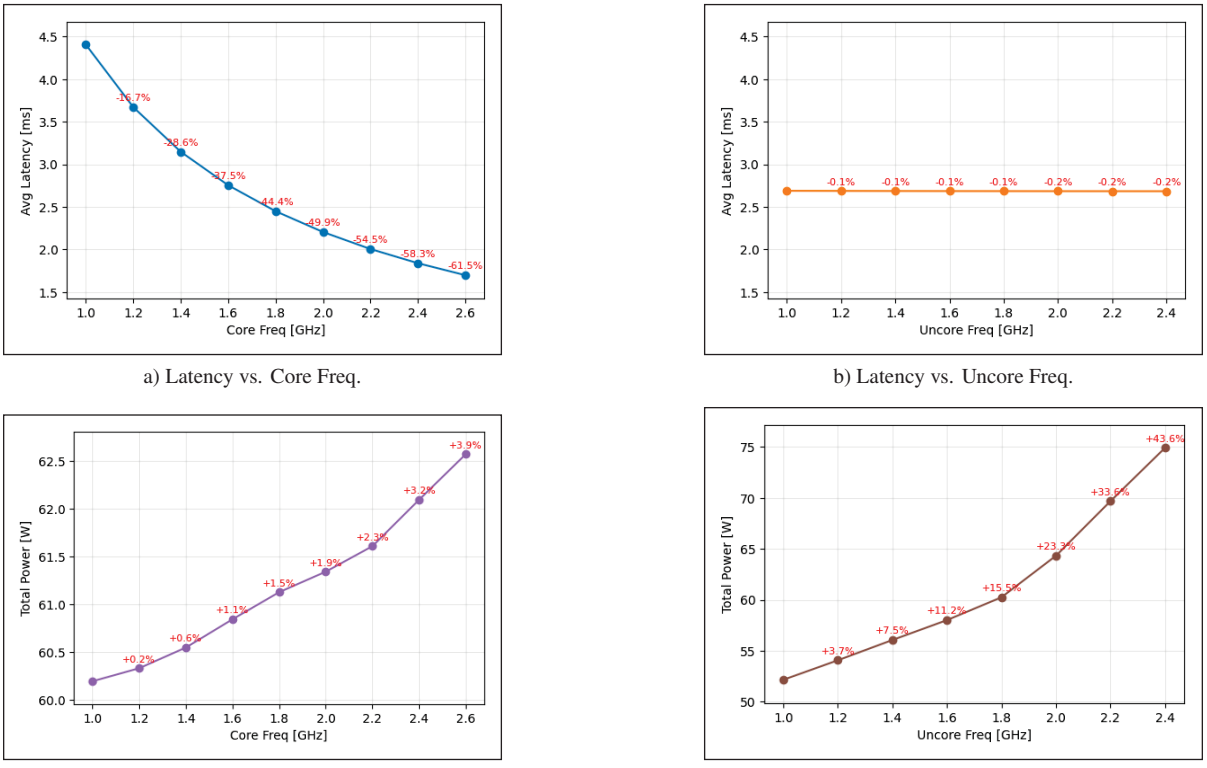
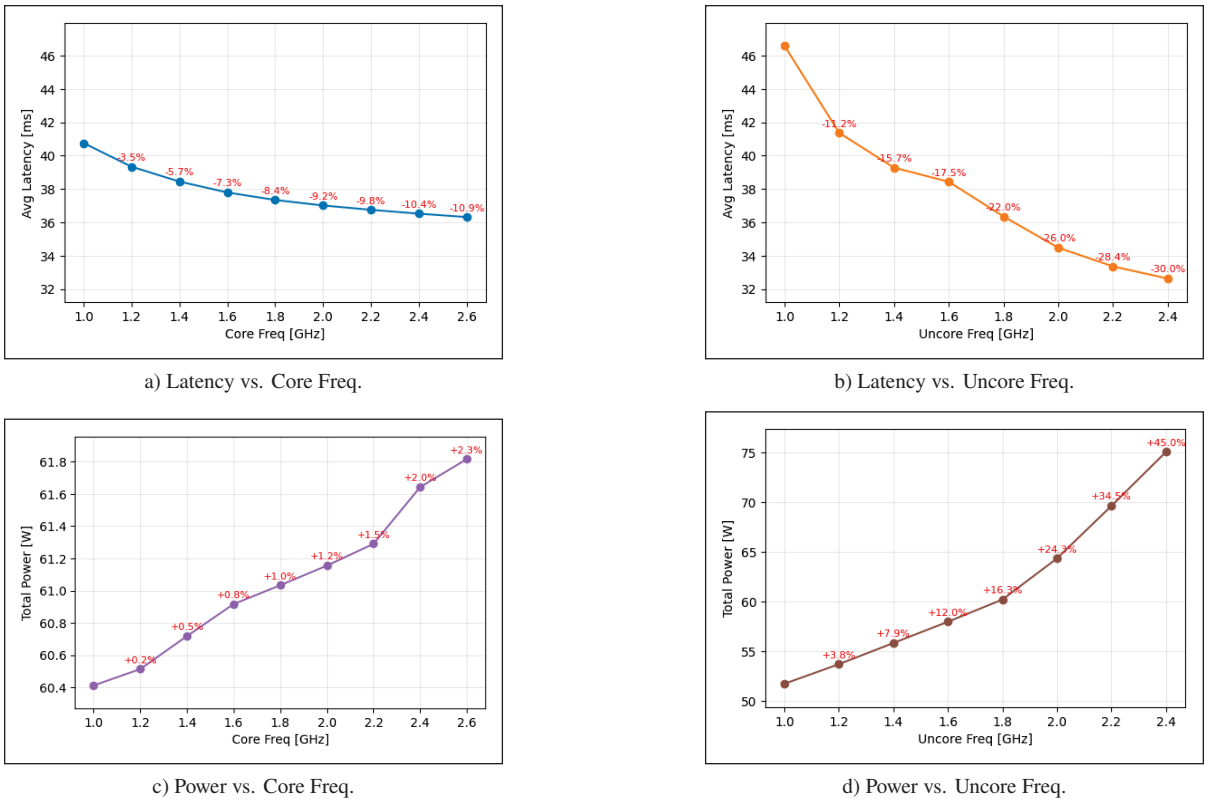


Figure 3.6 Impact of core and uncore frequency scaling on the CPU-bound `BG_CPUAckermann` microservice.

To establish a performance reference for each service, we sweep joint core/uncore frequency configurations and execute the service in isolation on the socket across the defined load levels. We record end-to-end response times



a) Latency vs. Core Freq.

b) Latency vs. Uncore Freq.

c) Power vs. Core Freq.

d) Power vs. Uncore Freq.

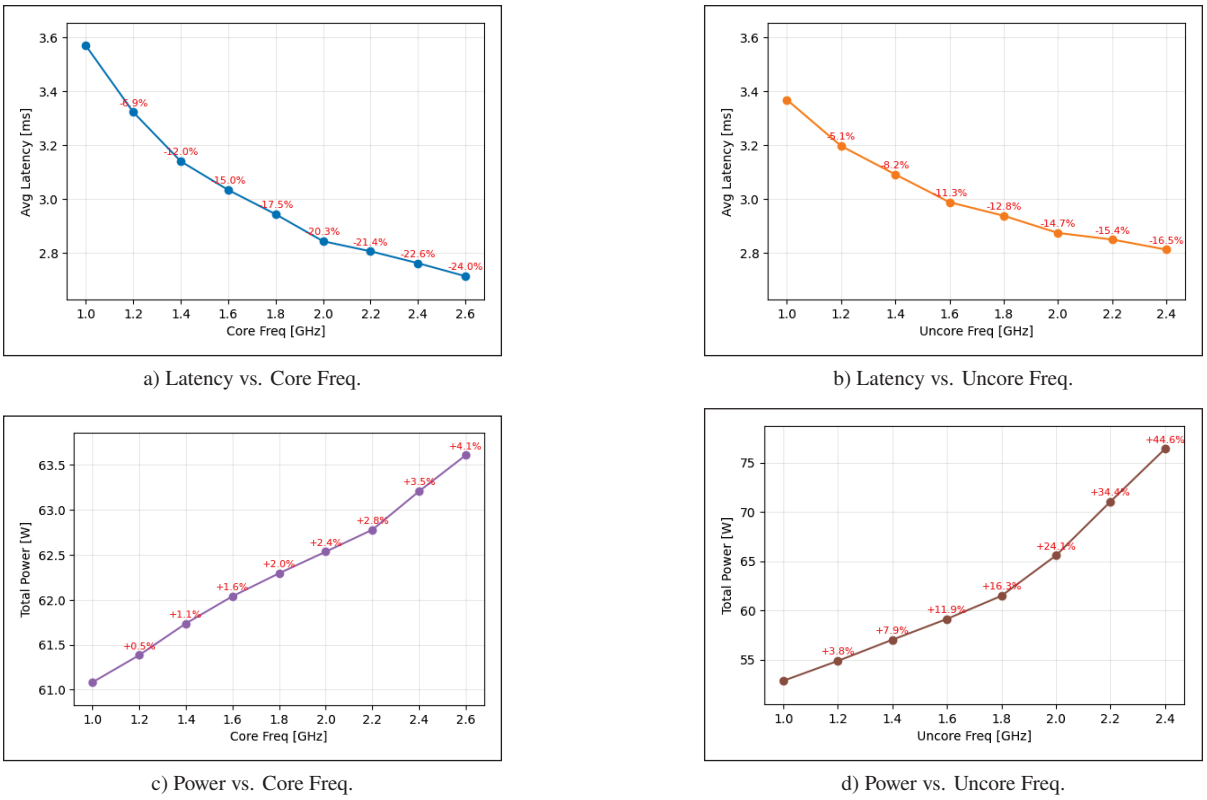
Figure 3.7 Impact of core and uncore frequency scaling on the memory-bound `BG_MemrateFlush` microservice.

and define the target latency for each microservice at each configuration as the 70th percentile of the measured samples.

Uncore Interference Generation:

To evaluate the controller's robustness against noisy neighbors, we generate controlled uncore interference using the PMBW benchmark. We employ two distinct memory-bandwidth stressor profiles: `ScanRead` (memory read pressure) and `ScanWrite` (memory write pressure).

As illustrated in Fig. 3.9, on the 14-core test socket, one physical core is strictly dedicated to the target latency-sensitive microservice deployed within the Kubernetes environment. The remaining 13 cores are utilized to host uncore stressor instances (USI) to generate varying levels of interference. These uncore stressors are deployed outside the Kubernetes boundary as standalone, non-containerized processes. To ensure precise and consistent interference scaling, each stressor instance is pinned to a distinct physical core via `taskset`, allocated 2 GB of memory, and its core frequency is statically locked at the maximum 2.6 GHz. By dynamically scaling the number of active stressor instances from 1 to 13, we accurately emulate escalating levels of background uncore contention. This process generates a detailed dataset for every benchmark across our training, evaluation, and test dataset



a) Latency vs. Core Freq.

b) Latency vs. Uncore Freq.

c) Power vs. Core Freq.

d) Power vs. Uncore Freq.

Figure 3.8 Impact of core and uncore frequency scaling on the mixed-workload `BG_Memthrash` microservice.

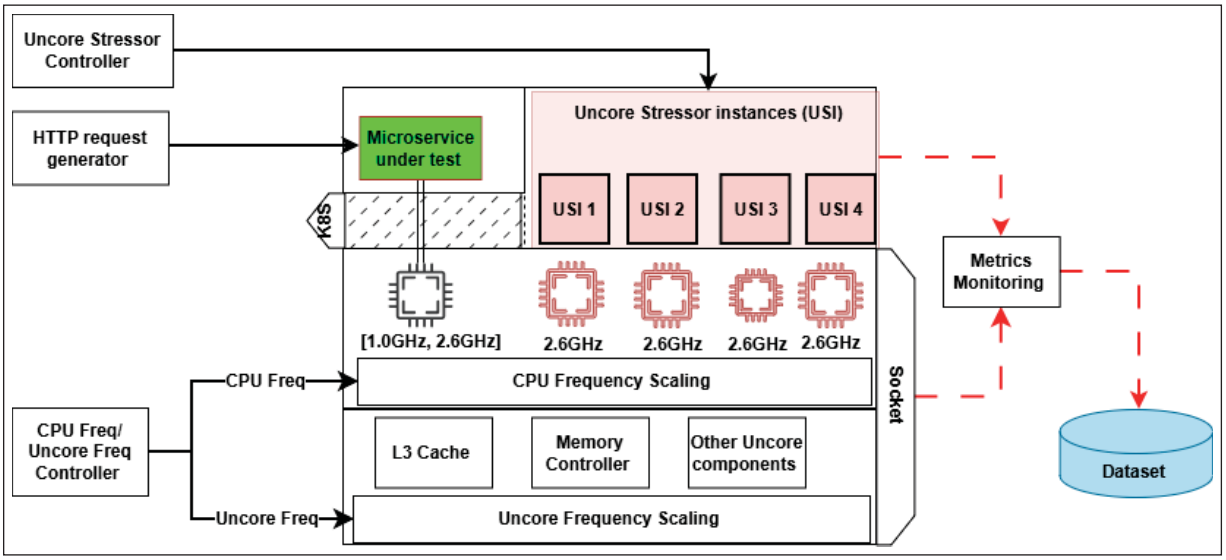


Figure 3.9 Uncore Pressure Generation and Its Real-World Impact: Trace Collection Setup

splits. Specifically, for every core and uncore frequency combination, the dataset records both the microservice's performance and the resulting socket-level metrics under varying levels of uncore stress.

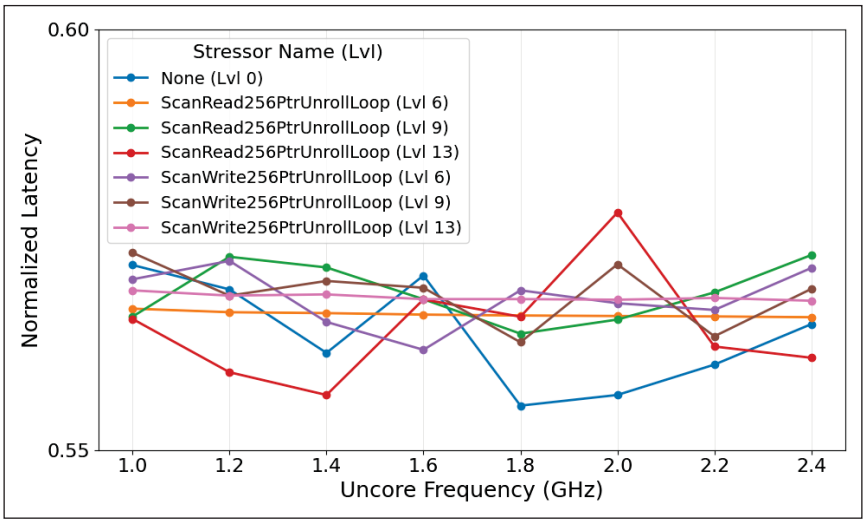
Fig. 3.10a, Fig. 3.10b, and Fig. 3.10c show the impact of escalating uncore stress levels (0, 6, 9, and 13 instances) on CPU-bound, memory-bound, and mixed workloads, respectively. For these experiments, we fix the core frequency at 1.8 GHz and the offered load at its maximum. The data reveals that CPU-bound workloads are largely immune to background uncore contention, as their latency is strictly compute-dominated. Conversely, memory-bound workloads suffer severe, yet approximately linear, performance degradation as the interference scales. Mixed workloads exhibit a moderate, hybrid impact that depends heavily on their active memory bandwidth reliance. Crucially, the latency degradation footprint is dictated not only by the intensity of the interference (i.e., the uncore stress level) but also by the specific workload nature of the uncore stressor (e.g., `ScanRead` vs. `ScanWrite`) and the target microservice workload. This demonstrates the complex landscape of real-world cloud environments, where diverse workloads running on the same socket generate highly variable patterns of uncore interference.

Crucially, the RL agent is completely blind to the explicit stressor configuration and active instance count. It must infer the presence and magnitude of external interference indirectly through socket-level telemetry (e.g., shared uncore counters and memory bandwidth saturation). This design forces the agent to dynamically detect and mitigate performance degradation caused by unseen, co-located workloads using strictly observable hardware signals.

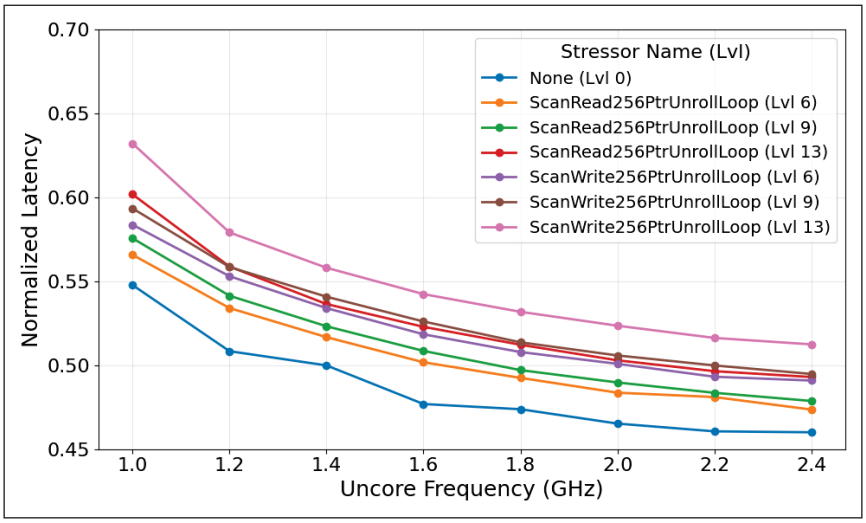
3.8.2 Offline reward shaping and weight selection.

We use historical measurements that jointly record latency outcomes and power consumption over a dense set of core/uncore frequency configurations, spanning different microservices and load levels. For each (microservice, load) condition, we define the *optimal point* as the lowest-power frequency configuration that satisfies the performance requirement (i.e., the minimum-power setting with non-positive gap, $g \leq \epsilon$). Reward shaping is calibrated to make this reference point uniquely desirable: the reward attains its maximum at the optimal point, drops sharply under performance requirement violations ($g > \epsilon$), and decreases smoothly (yet perceptibly) in the SLO-safe region when frequencies are unnecessarily high, thereby penalizing wasted power due to over-provisioning.

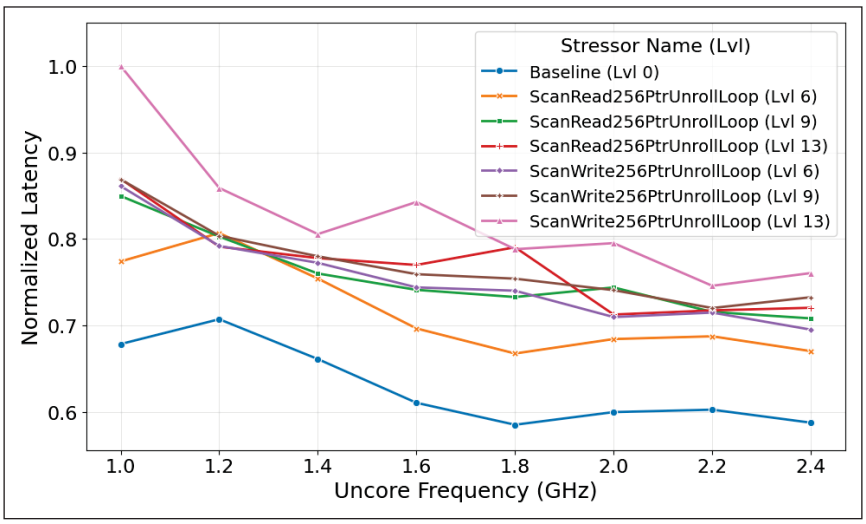
Because core and uncore frequency scaling have different marginal impacts on package power, we control their relative penalties using weights (w_c, w_u) in R. Since the uncore domain is shared and strongly influences package power, we set $w_u = 2w_c$ (a ratio commonly adopted in vendor and guidance for uncore power management (Juan & Marculescu, 2012)) and evaluate three configurations: $(w_c, w_u) \in \{(0.1, 0.2), (0.2, 0.4), (1, 2)\}$. As shown for `BG_ScanRead64PtrSimpleLoop` (see Fig 3.11) across multiple loads, $(w_c, w_u) = (1, 2)$ yields the clearest shaping: R peaks at the circled global optimum and then decreases approximately linearly as core



a) Impact on CPU-bound microservice performance (STRESS_CPU_Longdouble).



b) Impact on memory-bound microservice performance (BG_ScanWrite256PtrUnrollLoop).



c) Impact on mixed-workload microservice performance (BG_Memthrash).

Figure 3.10 Impact of escalating uncore stress levels on different microservice workload profiles.

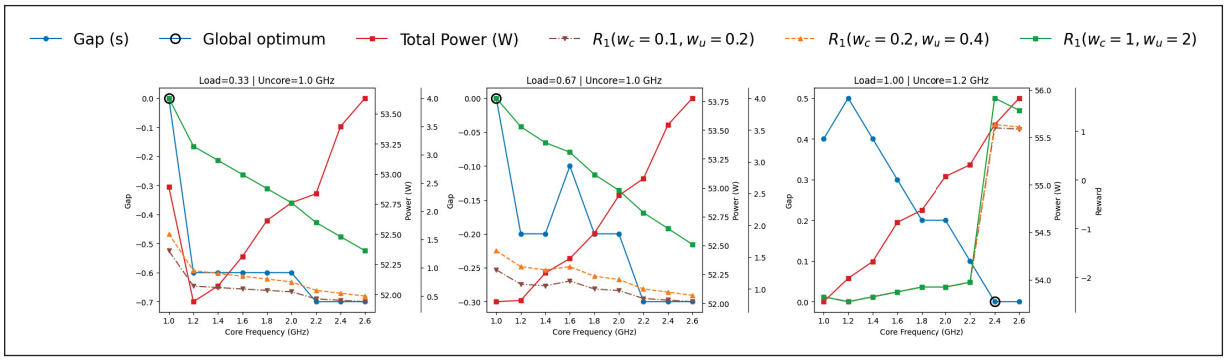


Figure 3.11 Reward function behavior under different core and uncore frequency configurations for BG_ScanRead64PtrSimpleLoop microservice .

frequency increases. The smaller-weight settings produce flatter reward curves after reaching the SLO-safe region, which weakens the incentive to downscale. We therefore adopt $(w_c, w_u) = (1, 2)$ in the remainder of the study.

3.8.3 Impact of State Normalization

We evaluate four normalization strategies:

- **No normalization.** The agent receives raw metrics directly. This introduces zero preprocessing overhead but is prone to severe scale imbalance, where high-magnitude features dominate learning.
- **Global Z-score.** A single Z-score transform is applied to all metrics. Although it standardizes scales, it can distort the physical meaning of key control and target variables (e.g., frequencies and latency).
- **Selective Z-score.** Only background/system metrics are normalized, while control fields are kept in their raw units. This mitigates scale mismatch, but remains frequency-blind by mixing statistics across heterogeneous operating points.
- **Custom (frequency-conditioned).** Separate normalization statistics are maintained for each (core, uncore) frequency pair. This captures regime-specific behavior without cross-contamination, at the cost of a brief cold-start period when a new pair is encountered.

The quantitative impact on training stability is summarized in Table 3.5. The performance is defined as the mean normalized evaluation reward after convergence. Figure 3.12 shows the normalized reward during evaluation under different normalization strategies.

Overall, the *Custom* normalization yields the best combination of sample efficiency and robustness. By isolating statistical baselines per hardware operating regime, it stabilizes learning and evaluation behavior, converging within

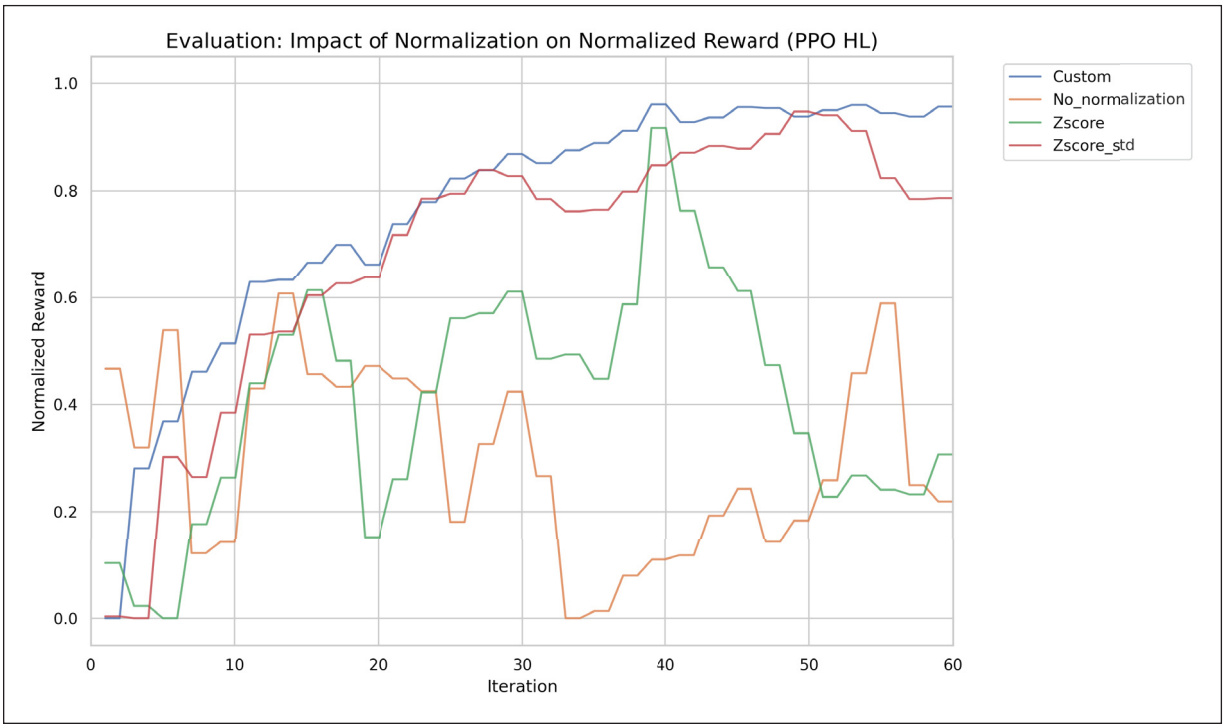


Figure 3.12 Normalized reward during evaluation under different normalization strategies.

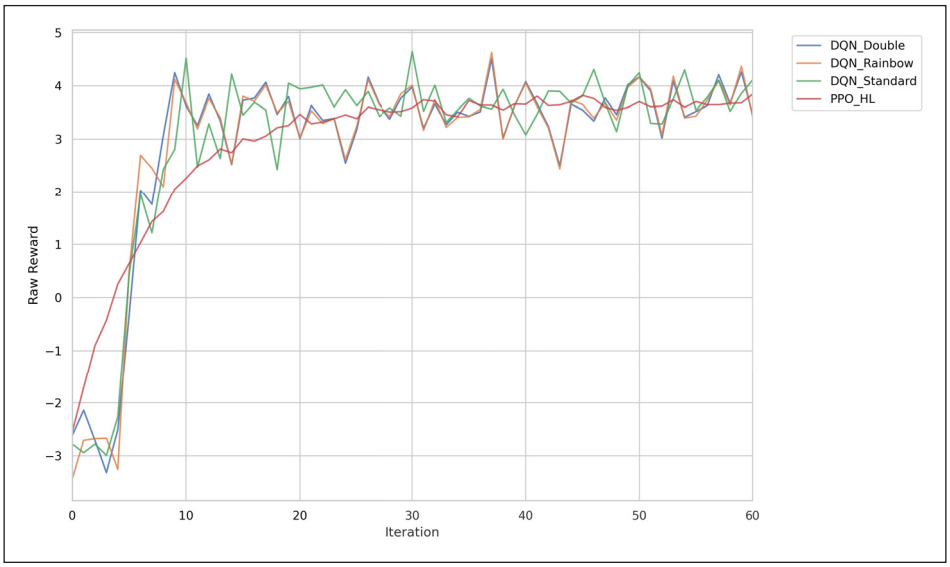
Table 3.5 Impact of normalization methods on training stability and convergence.

Method	**Convergence (iters)**	**Final performance**	**Post-conv. CoV**
Custom (freq-conditioned)	15–20	0.892	10.1%
Z-score (selective)	25–30	0.850	14.7%
Z-score (global)	30–40	0.820	18.9%
No normalization	50+ (unstable)	0.650	43.1%

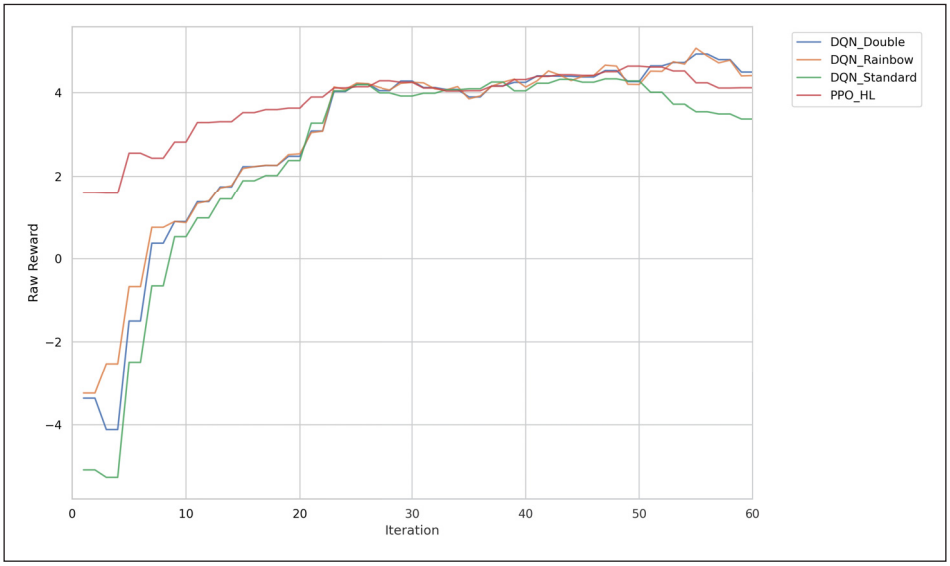
15–20 iterations while achieving the highest final performance (0.892). It also exhibits the lowest post-convergence variability (CoV 10.1%), compared to the highly unstable no-normalization baseline (CoV 43.1%).

3.8.4 RL Configuration and Training Details

We implement the hierarchical controller in Ray RLlib and adopt Proximal Policy Optimization (HL-PPO) as the default optimizer; hyperparameters and network settings are summarized in Table 3.6. We select PPO because its clipped on-policy updates are empirically more stable under the non-stationarity induced by dynamic load and co-runner interference, and it typically requires less sensitive hyperparameter tuning than value-based methods in continuous-control settings.



a) Reward function trajectories during training.



b) Reward function trajectories during evaluation.

Figure 3.13 Reward function trajectories for different RL algorithms.

We validate this choice by comparing HL-PPO with standard DQN, Double DQN, and Rainbow DQN under an identical reward definition and training. Figure 3.13a and Figure 3.13b report the reward trajectories during training and evaluation, respectively, across the different RL algorithms. After convergence (approximately iteration > 20), all methods reach comparable average reward; however, HL-PPO exhibits substantially lower post-convergence reward volatility (about ± 0.2) than Double DQN and DQN (about ± 0.5, $\approx 2.5\times$ higher) and Rainbow DQN (about ± 0.6, $\approx 3\times$ higher). This stability is important for deployment because it translates into more predictable frequency decisions and reduced control jitter.

Table 3.6 Minimal hyperparameter summary.
We keep the same training budget (N_{steps})
and shared settings across all methods;
only algorithm-specific knobs differ.

Parameter	**HL-PPO**	**HL-DQN**
Training interaction	$N_{\text{steps}} = 100$	
Optimizer / lr	Adam; 3×10^{-4}	
Discount γ	0.99	
Train batch size	4096	
Policy structure	Hierarchical (coarse+fine)	
Action space size	$(\lvert\mathcal{A}_{\text{coarse}}\rvert, \lvert\mathcal{A}_{\text{fine}}\rvert) = (9, 25)$	
PPO: clip / GAE / entropy	0.2 / 0.95 / 0.01	–
DQN: buffer / target update	–	65,536 / 500
DQN variants (if used)	–	Double, Dueling, Rainbow

A further reason for adopting PPO is its gradual on-policy adaptation under workload non-stationarity: in cloud deployments, K8SPI can optionally perform periodic online calibration using newly collected runtime samples to refine the policy. PPO's clipped updates help limit abrupt policy changes during such fine-tuning, reducing the risk of transient control spikes under distribution shift.

3.8.5 Baseline Governors

We benchmark K8SPI against the following baselines:

1. **`performance` Governor:** Default used power Governor that pinning core and uncore frequencies to maximum hardware limits, establishing upper bounds for both power and performance.

2. **Flat Single-Agent PPO (Ablation):** A non-hierarchical PPO agent managing the joint core-uncore action space. For fair algorithmic comparison, it uses K8SPI's exact observation space and a mathematically equivalent single-step reward:

$$R_{\text{flat}} = \begin{cases} -(2+g) - \psi, & g > \epsilon \\ (1 - |g|) + w_c(1 - a_{\text{c}}) + w_u(1 - a_{\text{u}}), & g \leq \epsilon \end{cases} \tag{3.20}$$

3. **Offline Optimal (Oracle):** Computed offline by scanning all frequency permutations per (microservice, load) pair. It selects the tuple $(f_{\text{c}}^{\text{opt}}, f_{\text{u}}^{\text{opt}}, P^{\text{opt}}, g^{\text{opt}})$ that strictly satisfies the performance requirement ($g \leq 0$) while minimizing total power, providing an absolute reference to quantify our policy's optimality gap.

It is important to note that the oracle is not defined as the absolute minimum-power configuration regardless of performance. Instead, it is the minimum-power configuration among only those configurations that satisfy the performance requirement. Therefore, a controller may sometimes report a lower power consumption or higher power reduction than the oracle, but such a result is not considered better if it is obtained with a positive performance gap, i.e., with latency violations.

3.8.6 Performance Metrics

During the training/evaluation phase, the agent is evaluated over entire episodes of length T. We define four primary metrics averaged over the episode steps $t \in \{1, \ldots, T\}$:

1. **Gap Percentage ($\tilde{g}$):** The mean relative deviation of the resulting latency from the target latency over the episode. Let g_t be the latency gap at step t. The mean gap is calculated as:

$$\tilde{g} = \frac{1}{T} \sum_{t=1}^{T} g_t \tag{3.21}$$

   A positive value ($\tilde{g} > 0$) strictly indicates an performance requirement violation, whereas $\tilde{g} \leq 0$ signifies compliance.

2. **95th Percentile Gap ($\tilde{g}_{p95}$):** Because the mean gap can obscure transient performance spikes, we also evaluate the 95th percentile gap. This represents the value that 95% of the observed latency gaps (g_t) fall below. It serves as a strict measure of tail latency, ensuring the system remains reliable even under worst-case interference.

3. **Power Distance Percentage (D_P):** The mean relative deviation of the controller's power consumption (P_t) from the offline optimal oracle (P_t^{opt}) across the episode. It is calculated as:

$$D_P = \frac{1}{T} \sum_{t=1}^{T} \left(\frac{P_t - P_t^{\text{opt}}}{P_t^{\text{opt}}} \right) \times 100 \tag{3.22}$$

   A lower value means better power efficiency.

4. **Normalized Reward:** The episodic reward scaled within each testing run to enable fair comparison across different algorithms and configurations.

5. **Strict Efficiency Score (SES):** A single metric that combines power efficiency and performance requirement compliance, with strong penalties for latency violations. It is defined as:

$$\text{SES} = -D_P - \text{Penalty}(\tilde{g}) \tag{3.23}$$

where Penalty($\tilde{g}$) applies a regime-specific cost based on the gap:

$$\text{Penalty}(\tilde{g}) = \begin{cases} 0, & \tilde{g} \leq -\epsilon \\ \frac{\tilde{g}+\epsilon}{\epsilon}, & -\epsilon < \tilde{g} \leq 0 \\ e^{\frac{\tilde{g}}{\epsilon}}, & \tilde{g} > 0 \end{cases} \tag{3.24}$$

3.8.7 Testing Environment

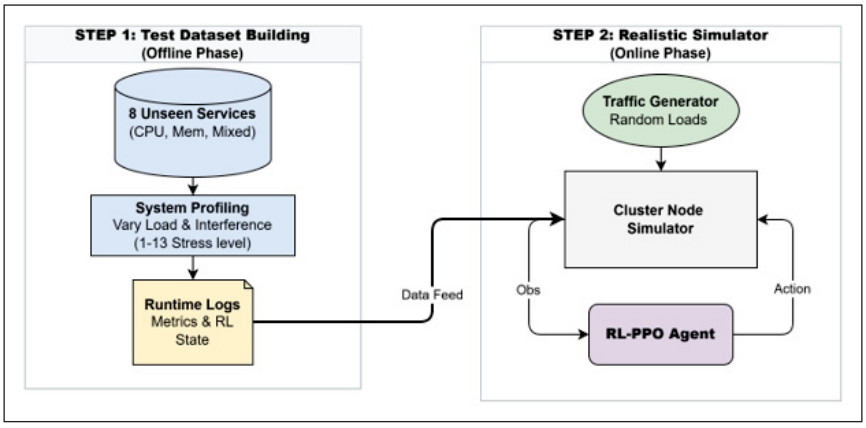


Figure 3.14 Normalized reward during evaluation under different normalization strategies.

We evaluate the trained policy using a replay-driven simulator (see Figure3.14 illustrates the testing environment simulator) in closed loop. after the offline phase that builds the test dataset as described in 3.8.1. The simulator is driven by (i) a traffic generator that determines the time-varying load profile and (ii) a simulator component that replays the system state (online phase). It emulates a non-stationary cloud node by executing microservices in fixed time windows while varying offered load and uncore interference at a fixed interval, we refer to this fixed interval as a *workload epoch* ($T_{\text{epoch}} = 60\,\text{s}$) to emulate time-varying demand in real cloud settings. At each workload epoch, the PPO agent receives the same observation representation used in the live system and outputs a joint core/uncore frequency adjustment. The simulator resolves each action by replaying the corresponding measured outcome from the dataset, conditioned on the active microservice, load level, interference setting, and selected core/uncore configuration. This design enables reproducible closed-loop testing while avoiding perturbations that would be introduced by exploratory actions on the physical node.

3.9 Evaluation and Results

To thoroughly validate K8SPI's effectiveness, we evaluate the framework across three complex, gradual, real-world scenarios.

3.9.1 Scenario 1: Isolated Latency-Critical Microservice

Setup. In this scenario, we evaluate our solution in an interference-free environment. A single latency-critical microservice is deployed on the test socket at a time. For each microservice in the test dataset, we run a 300 s trace under continuous sequential HTTP requests. The load level is updated every 60 s; (T_{epoch} = 60 s). K8SPI employs event-triggered actuation and intervenes only when the performance gap leaves the safety band, i.e., $g > +20\%$ or $g < -20\%$. To evaluate the efficiency of our RL design in reaching optimal hardware configurations with minimal action, we impose strict action budgets. Per load period, the hierarchical controller is allocated a maximum of 6 steps (equivalent to three full coarse→fine cycles), while the flat single-agent PPO baseline is strictly limited to 3 actions.

3.9.2 Results and Trace Analysis

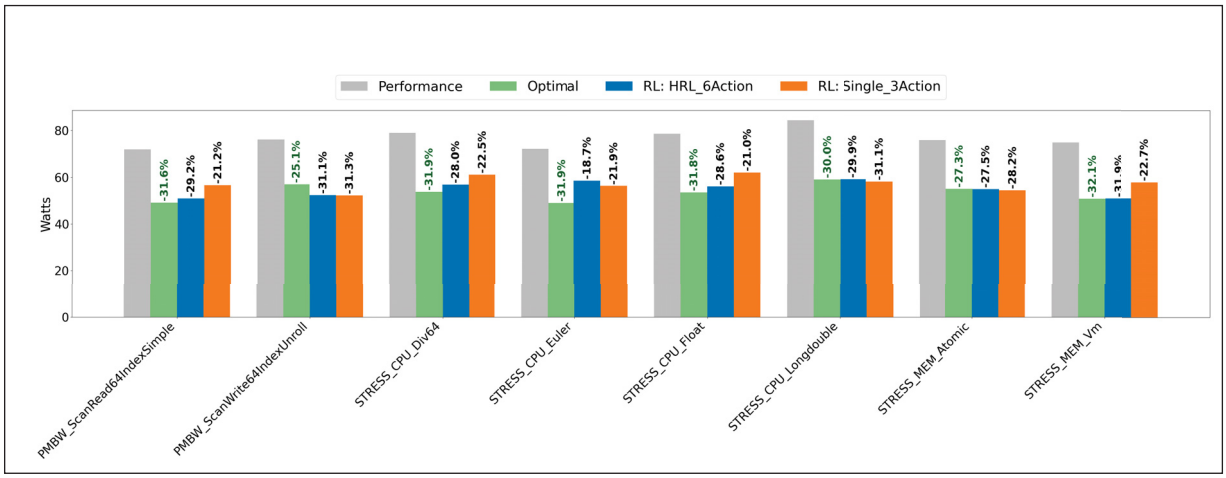


Figure 3.15 Total Power Consumption per Benchmark Under Different Power Governance Policies (Scenario 1)

Across isolated microbenchmarks, the hierarchical controller (HRL) consistently selects safer and more power-efficient operating points than the flat single-agent PPO: as summarized in Fig. 3.15 and Fig. 3.16, HRL maintains stable performace behavior (gap $\tilde{g}$ near zero or negative) while delivering high power efficiency (typically ≈30% savings) and closely tracking the offline oracle, whereas the single agent achieves only ≈20% savings on average and exhibits instability on sensitive workloads. The clearest failure case is `STRESS_CPU_Longdouble`, where

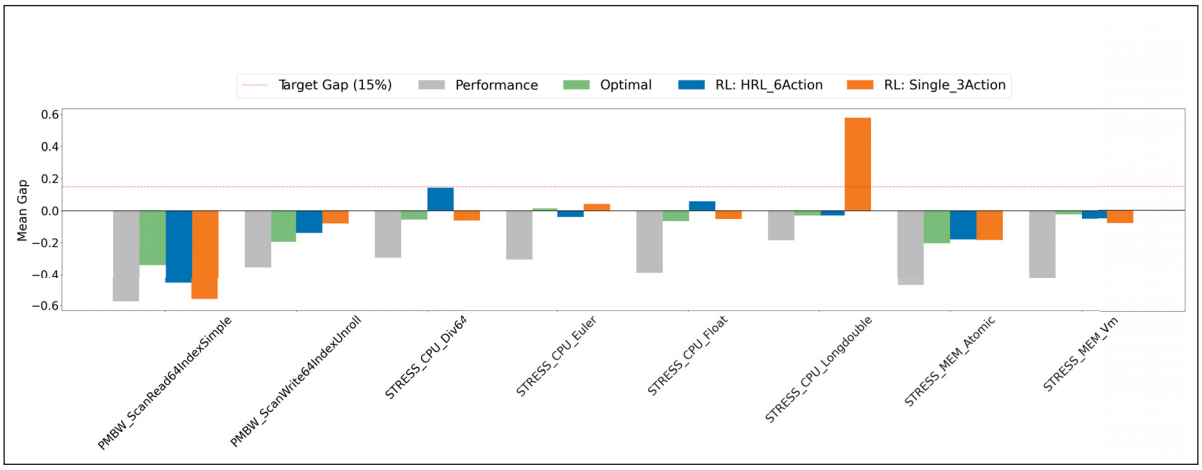


Figure 3.16 Average Performance Gap per Benchmark Under Different Power Governance Policies (Scenario 1).

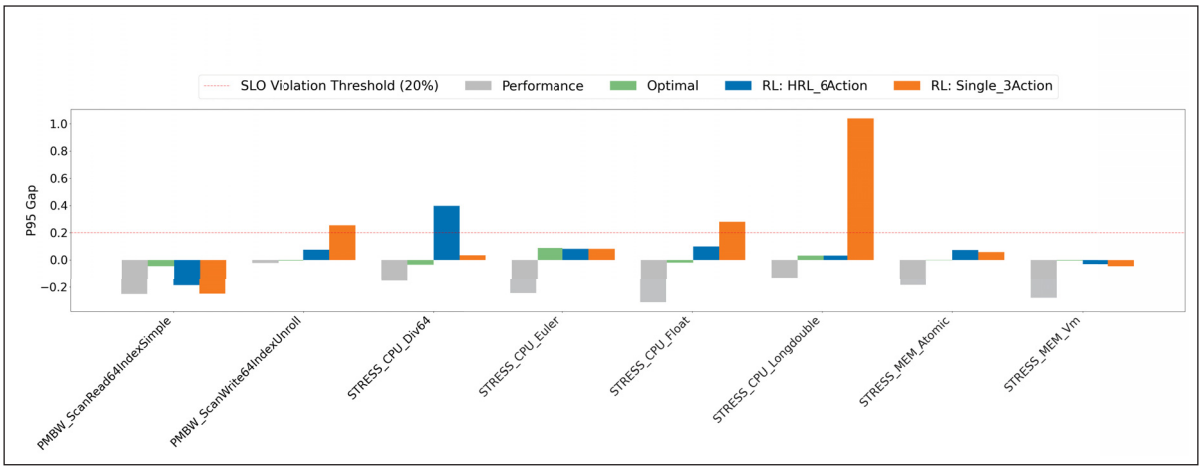


Figure 3.17 95th percentile Performance Gap per Benchmark Under Different Power Governance Policies (Scenario 1).

the single agent aggressively downscales and violates the performance requirement ($\tilde{g} = +0.58$ with 100% violation rate), while HRL preserves compliance ($\tilde{g} = -0.03$) and still attains ≈30% power savings.

This also explains why, in Fig. 4.15, the single-agent controller may appear to slightly outperform the offline oracle in terms of power reduction for `STRESS_CPU_Longdouble` (31% versus 30%). This is only a power-side comparison. The corresponding performance results show that this additional reduction is obtained at the cost of latency violations. Since the oracle is defined as the minimum-power configuration that still preserves the performance requirement, it is not actually outperformed. Rather, the single agent achieves lower power by selecting an unsafe configuration, while the oracle remains the best feasible reference under the latency-safety constraint.

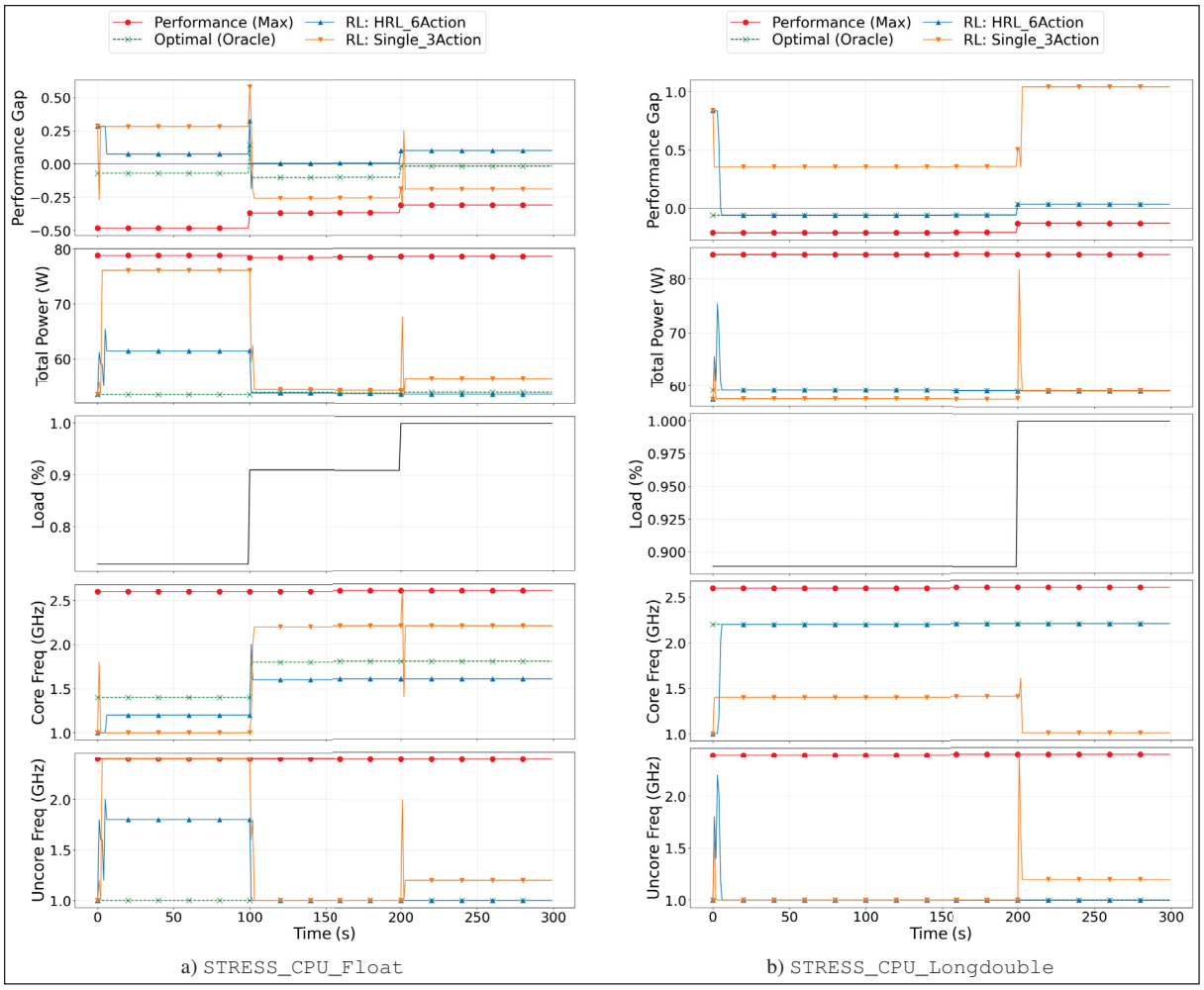


Figure 3.18 Trace analysis.

This stability is further validated by analyzing the 95th percentile gap ($\tilde{g}_{p95}$), which exposes transient tail latency spikes that the mean gap might obscure. As depicted in the Fig 3.17, the flat single-agent PPO frequently breaches the 20% performance requirement violation threshold ($\tilde{g}_{p95} > 0.2$). It exhibits severe tail latency spikes on workloads such as `STRESS_CPU_Longdouble` ($\tilde{g}_{p95} > 1.0$), `STRESS_CPU_Float` ($\tilde{g}_{p95} \approx 0.28$), and `PMBW_ScanWrite64IndexUnroll` ($\tilde{g}_{p95} \approx 0.25$). Conversely, HRL strictly suppresses these worst-case transient spikes, maintaining $\tilde{g}_{p95}$ well below the violation threshold for nearly all evaluated microbenchmarks. While HRL experiences a singular transient spike on `STRESS_CPU_Div64`, it overwhelmingly outperforms the single agent in bounding worst-case latency variations across the dataset.

While aggregate metrics demonstrate HRL's overall efficiency, a trace-level analysis further highlights the performance of HRL compared to the baselines. By examining temporal decisions across distinct microbenchmarks,

we identify how HRL successfully navigates the joint core/uncore frequency state-space to prevent destructive local optima.

For the `STRESS_CPU_Float` microservice (See the Fig3.18a), the offered load undergoes discrete step upgrades at $t = 100$ s and $t = 200$ s. During the initial phase, the flat single agent maximizes the uncore frequency to 2.4 GHz while starving the core frequency at its minimum of 1.0 GHz. Because this workload is heavily compute-bound, scaling the uncore provides zero latency benefit. Consequently, the single agent consumes 76 W while continuously violating the +0.2 performance requirement threshold (gap $\approx$ +0.3). In contrast, HRL correctly identifies the compute sensitivity, stabilizing at a core frequency of 1.2 GHz and an uncore of 1.8 GHz. This allows HRL to safely maintain compliance (gap $\approx$ +0.08) while saving 15 W compared to the single agent. As load increases, HRL dynamically readjusts to perfectly overlap the offline oracle's trajectory, pinning the uncore to its minimum and scaling the core to maintain a near-zero gap at optimal power (~54 W) without massive oscillations. Meanwhile, at maximum load (200–300 s), the single agent fails to adapt its actions to the workload's nature, unnecessarily increasing the uncore frequency from 1.0 GHz to 1.2 GHz, which needlessly increases total power consumption.

The trace of `STRESS_CPU_Longdouble` (see the Fig3.18b) further exposes the outperformance of the HRL agent compared to the baselines. HRL perfectly mimics the oracle policy throughout the execution (core 2.2 GHz, uncore 1.0 GHz), ensuring strict performance requirement compliance (gap never exceeding +0.02) at 59 W. The single agent, however, fundamentally fails to discover an adequate hardware configuration. It operates with a sustained +0.35 gap violation initially (0 to 200 s) and then suffers a catastrophic performance collapse at peak load (gap > 1.0) because it inexplicably bottoms out the core frequency to 1.0 GHz.

Ultimately, these traces demonstrate that HRL provides a highly generalizable and robust solution for joint core and uncore frequency scaling.

In memory-intensive regimes where core frequency yields diminishing latency benefit, HRL improves energy proportionality by confidently bottoming out compute resources without sacrificing performance: on `STRESS_MEM_Vm`, HRL reaches 51 W (31.9% savings) versus 58 W for the single agent (22.7% savings), effectively matching the oracle (50.9 W), and on `PMBW_ScanRead64` HRL achieves 29.2% savings versus 21.2% for the single agent; more broadly, HRL remains within 1–2% of the oracle across nearly all memory-bound cases.

Overall, under interference-free conditions, HRL exhibits oracle-level efficiency while maintaining performance-stable behavior, whereas the flat single-agent PPO is more prone to conservative local optima and occasional under-provisioning.

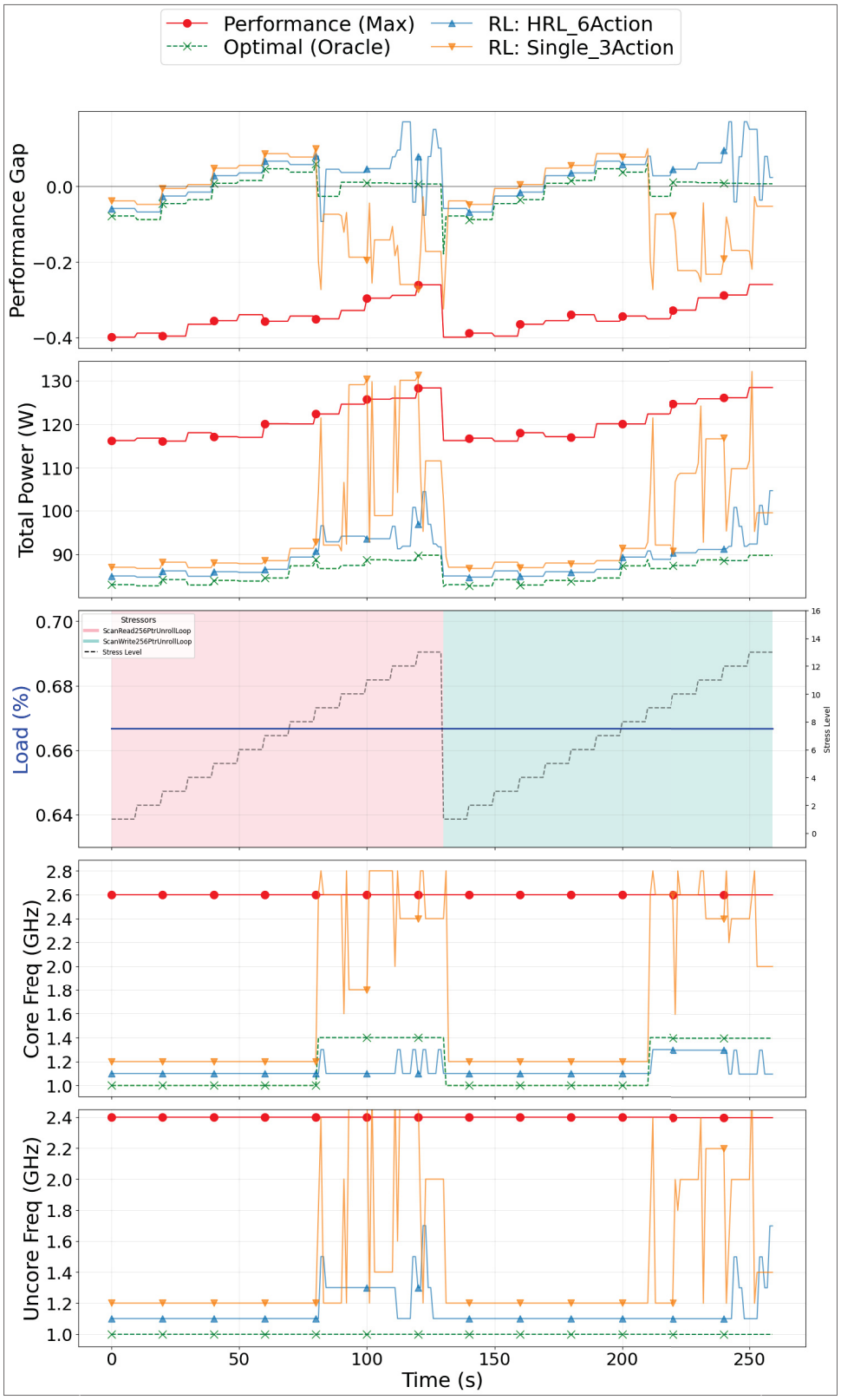
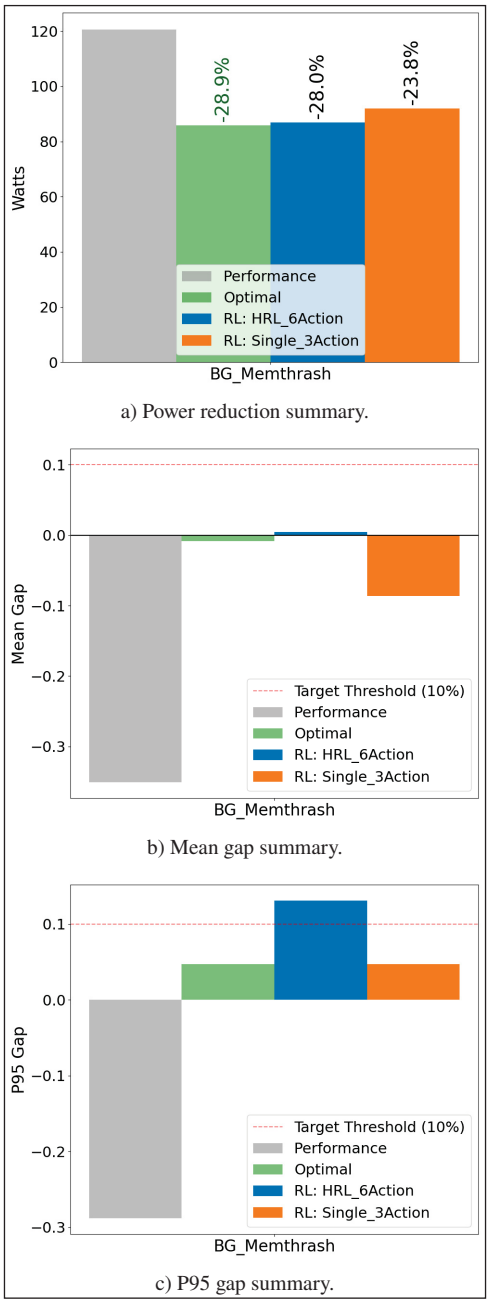


Figure 3.19 Trace analysis: `BG_Memthrash`.



a) Power reduction summary.

b) Mean gap summary.

c) P95 gap summary.

Figure 3.20 Performance summary (power reduction, mean performance gap, and P95 performance gap).

3.9.3 Scenario 2: Single Microservice with Best-Effort Interference

3.9.3.1 Step

To evaluate the controller under resource contention, we design a memory-bandwidth interference sweep. The target service is a memory-bandwidth-bounded application (`BG_Memthrash`) operating at a fixed offered load of 67%. This service is co-located with memory-intensive best-effort (BE) services (no performance requirement is considered for the BE).

We progressively increase the number of BE co-runners from 1 to 13 to emulate rising memory bandwidth contention, evaluating two BE workload profiles: `ScanRead` and `ScanWrite`. As contention intensifies, the target microservice's performance degrades, which manifests as an increased performance gap. Fig 3.19 illustrates the interference sweep setup and the resulting trace behavior as the number of BE co-runners increases. The RL controller is expected to minimize power consumption while strictly maintaining $\tilde{g}$ below the performance requirement constraint. Driven by observed interference intensity, socket-level metrics, and the real-time performance gap, the controller must dynamically adapt core and uncore frequencies to counteract the BE workload interference and its nature (scan read, scan write).

3.9.3.2 Results and Trace Analysis

As illustrate on the Figure 3.19, under low to moderate memory contention (Stress Levels 1–8), the performance gap ($\tilde{g}$) remains well within the safe boundary (-0.1 to 0.1). In this regime, both the hierarchical reinforcement learning (HRL) and single-agent controllers exhibit comparable baseline performance, maintaining $\tilde{g}$ between -4% and +3% while consuming 84 to 87 W. However, even in this low-stress environment, HRL demonstrates superior action-space efficiency by reliably pinning the uncore frequency to 1.0 GHz (note: a slight visual margin is added in the trace plots to prevent the HRL line from obscuring the optimal oracle). This strategy prevents unnecessary DRAM power consumption, whereas the single agent exhibits exploratory and inefficient scaling of uncore frequencies.

As best-effort interference amplifies (Stress Levels 9–13) and threatens the 0.10 (10%) performance gap threshold. The single agent adopts a reactive, brute-force approach, aggressively boosting core and uncore frequencies to mitigate the rising gap. This leads to severe over-provisioning; for instance, at Stress Level 10, the single agent consumes 108.9 W—wasting approximately 18 W compared to HRL—to achieve an excessive negative gap of -21.84%. Throughout these high-interference phases, the single agent exhibits spiky frequency oscillations and power bursts, indicating unstable control and the creation of an energy-expensive, conservative buffer.

Conversely, HRL demonstrates efficiency-aware control that closely aligns with the optimal oracle, avoiding the panic-driven over-provisioning of the single agent. Instead, it accepts a controlled and acceptable performance degradation (+5.01% gap at peak Stress Level 13, remaining well below the 10% threshold) to preserve energy proportionality. align with the optimal policy that accepts a marginal gap increase (≈+0.6%).

Overall summary metrics for the `BG_Memthrash` microservice confirm this efficiency. HRL achieves an average power reduction of 28.0% relative to the maximum performance baseline, nearly matching the optimal oracle's 28.9% savings, while the single agent achieves only a 23.8% reduction as depicted on Fig 3.20a. Furthermore, HRL maintains a mean gap almost perfectly at zero (see Fig 3.20b). While HRL's 95th percentile (P95) gap experiences brief transient spikes slightly above the 0.10 threshold (≈0.13) as shown in Fig 3.20c, the single agent aggressively suppresses these variations at a significant power cost. By reducing overall gap variance (σ_{gap}) despite stochastic memory stress, the hierarchical architecture successfully navigates the complex power-performance trade-offs inherent to memory subsystem contention.

Overall, for the `BG_Memthrash` microservice, HRL achieves a 28.0% power reduction, nearly matching the oracle's 28.9% and outperforming the single agent's 23.8%. HRL maintains an optimal mean gap near zero, whereas the single agent over-provisions to force a negative mean gap of ≈-0.09. While HRL's 95th percentile (P95) gap briefly exceeds the 0.10 threshold (≈0.13), the single agent strictly suppresses its P95 gap to ≈0.05 at a significant power cost (4.2% more power). By reducing the gap under stochastic stress, the hierarchical architecture successfully navigates non-linear power-performance trade-offs in the memory subsystem.

3.9.4 Scenario 3: Co-located Latency-Sensitive Microservices

3.9.4.1 Step

To evaluate the controller under complex, multi-tenant conditions, we deploy 14 latency-sensitive microservice instances co-located on the same CPU socket, as detailed in Table 3.7. These instances are instantiated exclusively from the memory-bound and mixed-workload test benchmarks. The experiment is conducted over a fixed duration of 3000 s,

To emulate a highly dynamic cloud environment, the offered load for each service changes randomly at the onset of each workload epoch (T_{epoch} = 60 s). Throughout the execution, the co-located services naturally interfere with one another by competing for shared memory bandwidth and uncore resources. Consequently, the performance requirement compliance of each individual instance is simultaneously challenged by its own stochastic load variations and the compounding effects of cross-service interference.

Table 3.7 Scenario 3 Co-located Microservices Configuration

Service Type	Instances	Workload Characteristic
PMBW_ScanWrite256IndexUnroll	5	Memory-bounded (write-intensive)
STRESS_MEM_Mmap	5	Memory-bounded (mmap operations)
PMBW_PermRead64SimpleLoop	4	Memory-bounded (read-intensive)

3.9.4.2 Results and Trace Analysis

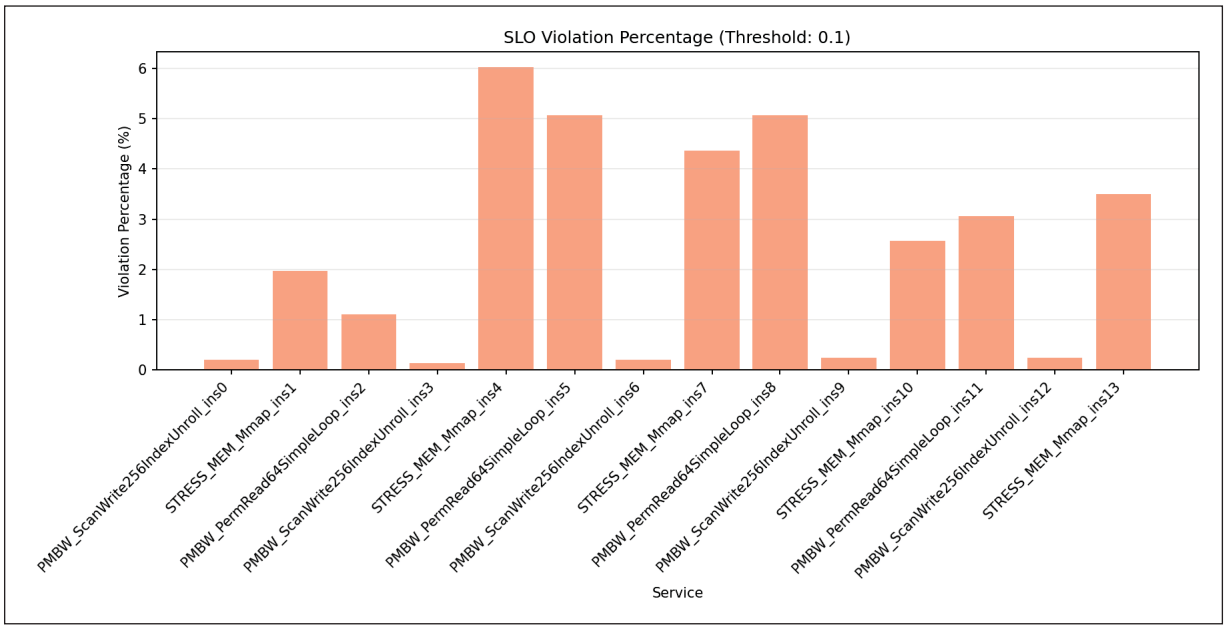


Figure 3.21 Detailed breakdown of performance requirement violation percentages across the diffrent microservices

Under realistic, multi-tenant cloud conditions where microservices demonstrate stochastic execution behavior, the HRL controller proves exceptionally resilient. The evaluation spans a 3000-second duration. The primary objectives are to enforce a strict performance requirement gap bound of +0.1 while maximizing energy savings. Acting entirely autonomously, the HRL agent explores a vast combinatorial space, concurrently managing independent core frequencies across 14 instances alongside the shared uncore frequency.

Fig. 3.25 illustrates the total power consumption reduction achieved by the proposed approach. Overall, the HRL agent achieves a 22.6% reduction in total power consumption compared to the baseline. Despite stochastic load variations and severe cross-microservice interference at the uncore level, the HRL agent successfully meets the varying performance requirements across the diverse deployed microservices. It achieves an aggregate compliance rate of 97.59%, suffering only 2.41% total violations across the entire 3000-second window.

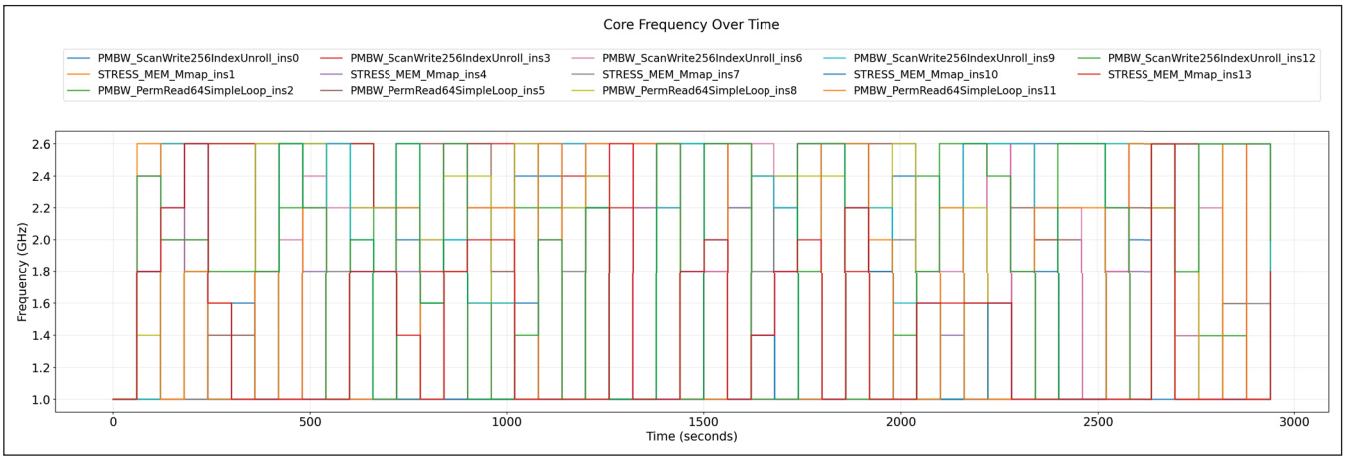


Figure 3.22 Dynamic trace of the independent adjustments made to the 14 CPU core frequencies over the 3000-second evaluation.

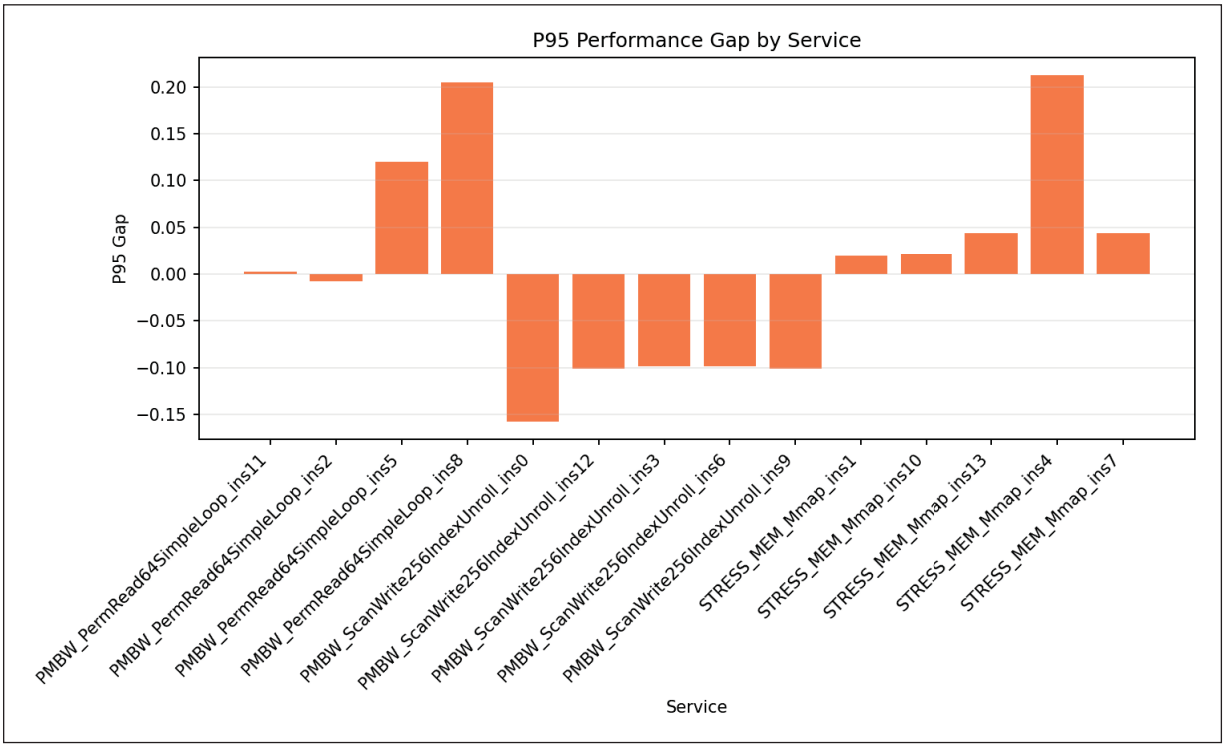


Figure 3.23 The 95th percentile (P95) performance gaps evaluated across the deployed microservices.

Fig. 3.22 and Fig. 3.24 show the dynamic trace of the independent CPU core and shared uncore frequencies over time. Trace analysis reveals the agent's highly dynamic learned policies: it rapidly and continuously adjusts all 14 independent CPU core frequencies jointly with the shared uncore frequency to maintain strict compliance under highly volatile contention.

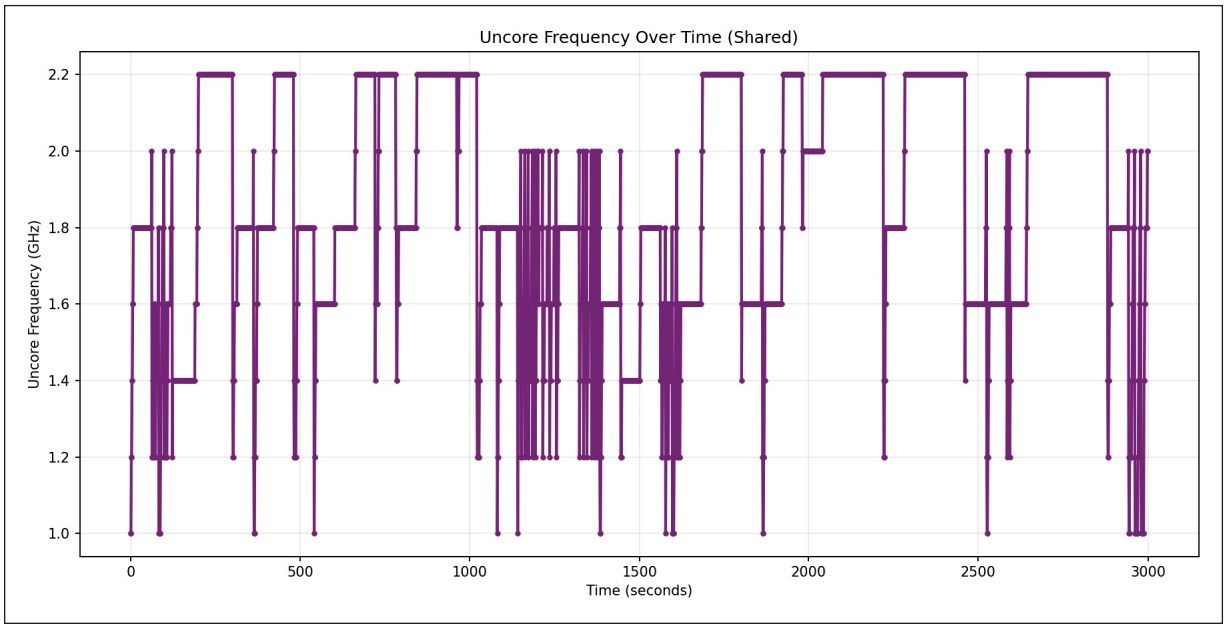


Figure 3.24 Trace of shared uncore frequency modulations executed by the HRL agent to manage cross-microservice interference.

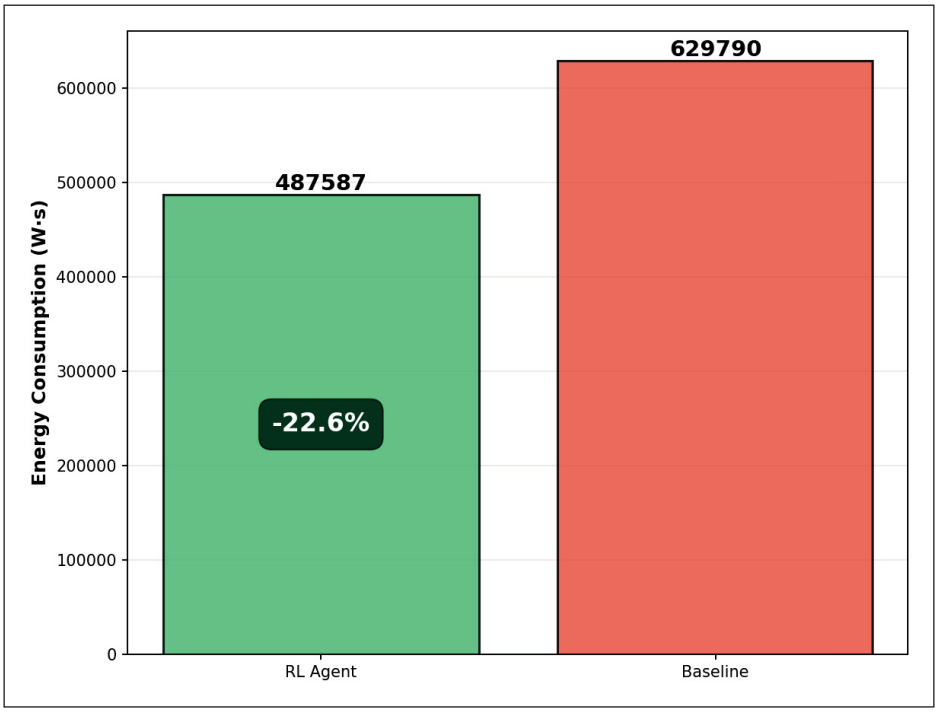


Figure 3.25 Total power consumption of the proposed HRL agent versus the default performance mode.

Fig. 3.21 details the violation percentages across the deployed microservices. A detailed breakdown of the violation percentages confirms the policy's robustness. Half of the evaluated microservices—such as `PMBW_-`

ScanWrite256IndexUnroll_ins0, PMBW_ScanWrite256IndexUnroll_ins3, and STRESS_MEM_-Mmap_ins7—achieved near-zero (≈ 1%–2%) violation rates over the testing period. Even the most sensitive and demanding microservice, STRESS_MEM_Mmap_ins4, peaked at a mere ≤ 6% violation percentage under maximal contention scenarios. This represents a highly acceptable worst-case operational bound, ensuring no single noisy neighbor leads to the system's full failure.

Fig. 3.23 illustrates the P95 performance gaps for the evaluated microservices. Operating exactly on the edge of a performance requirement threshold is risky under stochastic load. The proposed HRL avoids dangerous limit-riding via conservative optimization boundaries. The 95th percentile (P95) gaps remain in negative or near-zero alignment for the vast multitude of services. Even for the most vulnerable tasks, such as STRESS_MEM_Mmap_ins4 and PMBW_ScanWrite256IndexUnroll_ins8, the 95th percentile spikes stabilize right at the +0.2 boundary. This highlights that any latency spikes above the target are rapidly mitigated transients rather than sustained degradations.

3.10 Discussion

The evaluation reveals a consistent control pattern: K8SPI's hierarchical reinforcement learning (HRL) separates rapid mitigation of performance requirement violations from fine-grained power minimization, which prevents conservative "always-max" operating points under both isolated execution and multi-tenant contention. This section explains the mechanisms behind this behavior and the implications for cloud-native power governance with core/uncore actuation.

3.10.1 Decoupling Safety from Optimization

The failure of the single-agent baseline underscores a critical flaw in flat RL architectures when applied to latency-sensitive environments: the confounding of immediate constraint satisfaction with long-term efficiency. The single agent frequently falls into conservative local optima, aggressively over-provisioning resources at the first sign of performance degradation beyond the requirement. This behavior mimics traditional utilization-driven governors that conflate high resource allocation with system stability. By decoupling these objectives, K8SPI's hierarchical design ensures that the coarse-grained agent strictly handles rapid performance requirement mitigation, freeing the fine-grained agent to safely explore the lower bounds of power consumption.

3.10.2 Navigating the Memory Wall

A primary insight from the memory-contention scenarios is the physical limitation of independent frequency scaling. While CPU core frequency generally exhibits a linear relationship with computational throughput for CPU-bound

tasks, the memory subsystem (governed by the uncore frequency) presents a strict architectural bottleneck. When this "memory wall" is reached under high contention, further increases in core frequency result in diminishing latency benefits while exponentially increasing power consumption. The HRL agent successfully learns this non-linear boundary. It jointly scales core and uncore frequencies to adapt to a saturated memory bus. This indicates that the HRL model implicitly learns the hardware's microarchitectural constraints, avoiding the power-expensive states that trap default governor strategies.

3.10.3 Implications for Cloud Power Governance and Limitations

Two broader implications follow from the evaluation. First, application-level performance feedback (via the performance gap) is critical: without it, node-level governors cannot distinguish safe power reductions from under-provisioning, especially under interference. Second, uncore scaling governors must be integrated with core DVFS into a single policy. Because each workload has a unique footprint, joint core and uncore governance leverages multi-dimensional optimization to efficiently manage shared resources.

Despite these advantages, the current K8SPI framework has notable limitations. The system's responsiveness is bottlenecked by the observation latency of the monitoring stack. In our evaluation, telemetry aggregation via Prometheus introduces a default observation delay of up to 30 s. Consequently, making immediate reactions to highly dynamic load shifts and uncore interference remains challenging. To accommodate this, we fixed the load periods to 60 s ($T_{\text{epoch}} = 60\,\text{s}$) to allow the observation stack to reliably capture the necessary telemetry across all levels (application latency and hardware performance counters). Deploying K8SPI in highly volatile production environments will require integrating a low-latency, real-time observation system.

Furthermore, the current design is tailored for strict CPU pinning scenarios, utilizing a one-to-one mapping between microservice instances and physical CPU cores. While this isolates core-level compute, future work must extend the framework to cover broader, unpinned CPU execution scenarios and multi-core allocations to fully address diverse cloud-native deployments.

3.11 Conclusion and Future Work

This paper presented K8SPI (Kubernetes Power Irrigation), an event-triggered hierarchical reinforcement learning (HRL) controller for node-level power optimization of latency-sensitive microservices in Kubernetes. K8SPI adopts a two-stage architecture that separates rapid mitigation of performance requirement violations from fine-grained power minimization, enabling stable control over complex power–performance trade-offs. This separation is especially important in consolidated multi-tenant deployments, where co-located microservices have heterogeneous workload footprints and distinct performance requirements while contending for shared uncore resources.

To make RL-driven control practical for real systems, we also developed a rapid prototyping and training workflow. By combining RLlib's parallel training with system-state snapshots and trace-driven replay, the workflow supports efficient multi-environment training using real execution traces, reducing iteration time when reward shaping and policy design are refined.

Evaluation on a Kubernetes testbed shows that K8SPI reduces node-level power by 23%–30% relative to the Linux performance governor while keeping performance requirement violations below 2%–3%, even under dynamic load fluctuations and severe multi-tenant uncore contention. These results indicate that hierarchical RL with joint core/uncore actuation can avoid conservative performance-mode provisioning and achieve substantial power savings without compromising required performance.

Future work will proceed along two directions. First, we will extend K8SPI beyond strict CPU pinning to support unpinned execution and dynamic multi-core allocations, improving generality across cloud-native scheduling regimes. Second, we will address observability latency by integrating a low-latency telemetry path for critical signals, enabling faster reaction to micro-bursts and short-lived interference events.

CONCLUSION AND RECOMMENDATIONS

4.1 Summary

The growing energy demand of digital infrastructures has become a global concern. With the rapid expansion of cloud services, data-intensive applications, and AI-driven workloads, the energy footprint of computing systems is no longer only a technical issue for researchers and practitioners; it has become an environmental, economic, and societal challenge. In parallel, sustainability objectives and net-zero commitments are placing increasing pressure on governments and industry to improve the energy efficiency of the digital ecosystem. Since cloud computing now forms the backbone of modern digital services, enhancing its energy proportionality has become a pressing necessity.

In this context, this thesis addressed the problem of designing practical, performance-aware, and energy-efficient mechanisms for cloud-native environments, with a particular focus on containerized performance-sensitive applications across the cloud–edge continuum. Rather than treating energy reduction as an isolated objective, this work considered it together with the need to preserve application-level performance under realistic deployment constraints such as workload heterogeneity, dynamic runtime variation, multi-tenancy, and shared-resource interference.

The thesis approached this problem progressively. First, it showed that meaningful energy savings can be achieved at the edge through workload-aware DVFS control, even under constrained hardware settings. At the edge, the proposed GAS framework showed that workload-aware DVFS can reduce energy consumption by 5% to 23% under concurrent multi-microservice execution, and by up to 80% when a single microservice is deployed, while maintaining latency requirements.

Second, it demonstrated that accurate power observability remains a fundamental challenge in cloud-native systems, and that unreliable container-level power estimation can limit the effectiveness of energy-aware optimization. At the cloud observability level, the thesis showed that container-level power monitoring remains a major challenge in practice: the evaluation of Kepler revealed an RMSE of 11.9 W relative to Intel RAPL and an overestimation of up to 15× under dynamic runtime conditions, highlighting the limits of current power attribution approaches for cloud-native systems.

Third, the proposed K8SPI framework demonstrated that hierarchical reinforcement learning can reduce node-level power consumption by 23% to 30% relative to the Linux performance governor, while keeping performance requirement violations below 2% to 3%, even under dynamic load fluctuations and severe uncore contention.

Taken together, these contributions show that energy-efficient cloud-native computing cannot rely solely on generic utilization-based governors or isolated hardware tuning. Instead, it requires a broader systems perspective that connects hardware actuation, observability, workload behavior, and application-level performance guarantees. In this sense, the thesis contributes not only concrete mechanisms and experimental results, but also a coherent foundation for the design of sustainable cloud-native infrastructures.

Overall, this thesis shows that reducing the energy footprint of cloud-native systems requires performance-aware and practically deployable solutions. By addressing this challenge across both edge and cloud settings, it provides methods, insights, and design directions that support the development of next-generation sustainable digital infrastructures.

4.2 Recommendations

Several key recommendations emerge from this work:

- Power management in cloud-native systems should move beyond traditional utilization-driven policies and instead incorporate application-level performance objectives directly into runtime decision making. The results of both GAS and K8SPI show that meaningful energy savings are achievable only when the controller understands the workload's sensitivity to core and memory-system behavior, rather than relying on generic CPU usage signals alone.
- Power observability should be treated as a foundational requirement rather than as an auxiliary monitoring feature. The Kepler study clearly indicates that inaccurate container-level power attribution can mislead optimization decisions and weaken the reliability of energy-aware orchestration. For this reason, future sustainable cloud-native platforms should invest in more accurate measurement, validation, and attribution pipelines before building higher-level optimization mechanisms on top of them.
- Practical cloud-native energy optimization should be designed as an end-to-end systems problem. Effective solutions should combine hardware-level control knobs, multi-layer telemetry, orchestration-aware context, and strict performance assurance. This thesis shows that sustainable operation is best approached through

coordinated designs that bridge infrastructure control and service-level objectives, rather than through isolated optimizations at a single layer of the stack.

4.3 Future Work

Several directions remain open for future research:

- At the edge, GAS should be extended toward more heterogeneous hardware platforms. Moreover, incorporating prediction mechanisms for proactive frequency adjustment and exploring richer control capabilities beyond the current chip-wide DVFS assumptions would improve the generality of the approach.
- In cloud observability, future work should extend the proposed validation framework to additional runtime factors and evaluate alternative container-level power models capable of offering higher accuracy under realistic conditions. Since observability directly conditions the quality of optimization, improving both the fidelity and robustness of power attribution remains a critical research priority.
- For cloud runtime control, future work should broaden the applicability of K8SPI beyond the current deployment assumptions, particularly toward more general production environments where strict CPU pinning or tightly controlled allocations may not always hold. Further integration with higher-level orchestration functions, such as scheduling, placement, scaling, and consolidation, also represents a promising direction for building more holistic and deployable energy-aware cloud-native platforms.

APPENDIX I

GO GREEN, GO CHEAP (3GC): ACHIEVING ENERGY EFFICIENCY AND COST EFFECTIVENESS IN SERVERLESS EDGE COMPUTING

Zouhir Bellal, Laaziz Lahlou, Nadjia Kara, Timothy Murphy and Tan Phat Nguyen

Département de Génie Électrique, École de Technologie Supérieure,

1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Ericsson, Canada

Abstract: To align with sustainability goals and adapt to environmental regulations, such as carbon emission rights (CERs) trading where companies compensate for their CO2 emissions, Serverless providers must integrate energy consideration into their resource deployment and management models. Specifically, in edge computing, where resources are limited and latency requirements are stringent. For instance, edge computing serverless frameworks (e.g., OpenFaaS and OpenWhisk) often rely on Kubernetes scheduling schemes to allocate functions on the edge. However, these schemes typically overlook energy consumption as a decision factor, resulting in increased energy usage and, consequently, higher operational expenses (OPEX) for companies. To bridge this gap, we introduce a deadline-aware and cost-effective resource allocation approach for serverless service providers, named 3GC. Formulated as an integer nonlinear programming (INLP) problem, 3GC aims to minimize energy and execution costs. Furthermore, we propose a heuristic and an online algorithm to assess its efficiency in reducing energy and execution costs. 3GC employs per-core Dynamic Voltage and Frequency Scaling (DVFS) to optimize the balance between execution and energy costs. The results from exhaustive evaluation suggest that our methods surpass existing allocation techniques in terms of overall cost savings (up to 69.35%) while ensuring function latency requirements are largely upheld.

Keywords: DVFS, server-less, energy efficiency, resource allocation

1. Introduction

The integration of edge computing, crucial for IoT, autonomous vehicles, and AR/VR, is driven by 5G and the demand for low-latency solutions. Centralized cloud models often fall short in meeting the real-time processing needs of these modern applications, leading to a shift towards decentralized edge data centers to support latency-sensitive services. This transition is complemented by the adoption of serverless computing at the edge, which offers computing services in a more accessible, fine-grained, and cost-effective manner by offloading operational tasks like provisioning and scaling to the cloud provider (Mampage, Karunasekera & Buyya, 2022). Major cloud services

now provide serverless platforms, employing a pay-per-use billing model that charges for execution time rather than idle capacity (Raza, Matta, Akhtar, Kalavari & Isahagian, 2021). Serverless architecture, particularly Function as a Service (FaaS), allows for on-demand deployment of discrete functions. Frameworks such as OpenFaaS (Ope, 2021a) and OpenWhisk (Apa) facilitate serverless computing at the edge, fostering the growth of serverless edge computing ecosystems (Aslanpour, Toosi, Cicconetti, Javadi, Sbarski, Taibi, Assuncao, Gill, Gaire & Dustdar, 2021). Managing serverless edge computing presents unique challenges due to the heterogeneity of edge resources, complicating the development of a pricing model. This model must account for both usage duration and the specific characteristics of the resources, such as computational capacity (Li, Guo, Cheng, Chen, He & Guo, 2022b). For example, a CPU core with a higher frequency may incur greater costs, as seen with Google Cloud Functions, which bills based on CPU time and frequency (Goo). Thus, a comprehensive pricing model for edge computing must reflect the quantity and computational power of utilized resources (Wu, Buyya & Ramamohanarao, 2019). Additionally, with the growing emphasis on environmental sustainability, companies must also consider CO2 emissions in their cost assessments, influenced by mechanisms like carbon emission rights (CERs) trading (Anderson, Belay, Chowdhury, Cidon & Zhang, 2023; Kannan & Kremer, 2023). It consists of charging the companies for the amount of their emitted CO2. This necessitates serverless providers to reduce their carbon footprints, integrating both direct execution costs and energy-related expenses into the overall function cost in edge computing, aligning economic efficiency with environmental responsibility (Huber, Körber & Mock, 2019).

A sophisticated resource allocation mechanism is essential for balancing latency demands with cost minimization (Eusebio, 2022; Mampage *et al.*, 2022). While serverless providers offer some control over resource allocation, such as the number of CPU cores, optimizing CPU frequency for cost-efficiency without compromising on latency presents a challenge. High-frequency cores, although more expensive, provide the necessary computational power for latency-sensitive tasks. In contrast, lower-frequency cores, while cheaper, may not meet the deadlines of certain serverless functions due to slower processing speeds. Moreover, extending the execution time can inadvertently elevate costs, as charges are proportionate to function runtime. The nature of the workload significantly influences the optimal CPU frequency choice; CPU-bound workloads benefit from higher frequencies for better performance and energy efficiency, whereas memory-bound workloads might not require high frequencies, allowing for energy savings without performance degradation.

The Dynamic Voltage and Frequency Scaling (DVFS) technique, which adjusts CPU frequency to strike a balance between energy efficiency and performance, presents a promising extension to serverless edge computing frameworks by facilitating the selection of optimal CPU frequencies to reduce energy consumption (Tzenetopoulos *et al.*, 2023). Despite this potential, current serverless edge computing frameworks often overlook this feature, mostly employing Kubernetes for straightforward resource allocation tasks, such as assigning functions to nodes with adequate available cores, without considering energy or cost efficiency (Ope, 2021b; Rejiba & Chamanara, 2022).

Additionally, the default operation of edge nodes at maximum CPU frequency for optimal performance can lead to unnecessary energy use and higher costs.

In response, we introduce "Go Green Go Cheap (3GC)," a novel resource allocation strategy designed for serverless edge computing that prioritizes cost-effectiveness and energy efficiency. "3GC" enables serverless providers to (a) select appropriate CPU cores and their (b) corresponding operational frequencies for executing functions. The core objectives of the "3GC" scheme include: *(i)* ensuring adherence to function deadlines, *(ii)* reducing the financial burden on serverless providers and consumers, and *(iii)* enhancing energy efficiency, thereby promoting the environmental sustainability of serverless edge computing operations. We designed and evaluated the 3GC approach and compared it with various baselines. Our evaluation demonstrates that 3GC outperforms these baseline strategies, achieving significant reductions in overall costs by up to 69.39% and ensuring adherence to function deadlines, thereby fostering sustainable serverless edge computing.

2. Background & Related works

Serverless computing offers various advantages, including a pay-as-you-go pricing model, seamless auto-scaling, and automation capabilities. However, it is crucial to thoroughly investigate the energy implications of this technology to leverage its potential fully.

A technical study focused on enhancing the energy efficiency of serverless computing was discussed in (Sharma, 2023). The author investigated the energy-efficiency challenges and proposed avenues for overcoming them. The study highlighted that FaaS virtualization overheads can significantly increase energy consumption. Despite this, the literature on serverless computing largely lacks coverage of energy considerations, emphasizing the need for further thorough examination in this domain. DVFS, a long-standing solution for minimizing energy consumption, has recently experienced renewed interest in serverless computing when combined with other mechanisms. In (Rastegar *et al.*, 2023), authors introduced EneX, an energy-aware scheduler aiming to reduce energy usage during function execution.

A proposed solution, DVFaaS, utilizes per-core DVFS to optimize serverless workflows (Tzenetopoulos *et al.*, 2023). It adjusts core frequency through a parameterized linear function based on frequency and latency considerations for optimal performance. In (Jia & Zhao, 2021), the authors introduced RAEF, a runtime system that optimizes energy efficiency without breaching the latency-related Service Level Agreement (SLA) for serverless functions. RAEF employs a binary search approach to find the optimal balance between latency and energy consumption by exploring two-dimensional resources, namely frequency and CPU core count. The configuration with the lowest energy is then selected. In (Alzahrani *et al.*, 2016), the EBAS auto-scaling approach for cloud data centers was

introduced, employing a time-series model to predict CPU utilization and optimize core number and frequency based on SLA considerations. Additionally, DVFS is utilized to adjust all CPU core frequencies.

In (Herzog, Hügel, Reif, Hönig & Schröder-Preikschat, 2021), automatically utilizes various machine learning techniques to select the most energy-efficient configuration in dynamic systems.

Exploiting renewable energy sources is an excellent way to achieving sustainability and helping reduce CO2 emissions. (Aslanpour, Toosi, Cheema & Gaire, 2022) introduces energy-aware resource scheduling in serverless edge computing using a Raspberry Pi cluster powered by renewable energy. Their approach employs a greedy scheduling technique, dynamically adjusting function placement based on energy status for increased node availability. However, it overlooks function operating frequency, focusing solely on CPU resource allocation. In contrast, our solution, 3GC, utilizes a tailored heuristic technique and online algorithm, jointly optimizing CPU frequency and core allocation to minimize energy consumption and meet SLAs.

The pay-as-you-go model, a staple in cloud computing, now applies to serverless computing (Das, Leaf, Varela & Patterson, 2020; Eismann, Grohmann, Van Eyk, Herbst & Kounev, 2020; Elgamal, Sandur, Nahrstedt & Agha, 2018; Lin & Khazaei, 2020; Lin, Mahmoudi, Fan & Khazaei, 2022; Sedefoğlu & Sözer, 2021), billing users based on the resources consumed during function execution rather than idle time. The cost is influenced by the characteristics of the resources used and the duration of the function. For instance, Google Cloud Functions bills for CPU time and the frequency of the CPU core running the function (Goo). In contrast, using an ARM processor with AWS Lambda can be about 20% less expensive than using an x86 processor (Marquez, 2024).

In contrast to existing work, this study introduces 3GC, a resource allocation scheme that adopts a "pay-as-you-go" model, incorporating both energy consumption and execution costs incurred by serverless functions. 3GC uses CPU frequency control mechanisms to strike a balance between energy and execution costs. It aims to minimize the total cost while adhering to the latency requirements of serverless functions, offering a novel approach to resource allocation in serverless edge computing environments. Table I-1 summarizes the key distinctions between 3GC and existing approaches in terms of latency awareness, energy efficiency, and cost-effectiveness.

3. System Model and problem formulation

We consider an edge infrastructure with serverless-enabled nodes, each supporting various CPU frequencies, function instances are hosted based on node capacity for low-latency applications (see Figure I-1). The serverless framework, upon a function request, selects a node, allocates CPU cores, and adjusts CPU frequencies for optimal performance and energy use. Costs follow a pay-as-you-go model, combining execution costs, tied to the computational capacity (notably CPU frequency), and energy costs are derived from the energy consumption during function execution.

Table-A I-1 Summary of existing works

References	Energy-aware	latency-aware	Cost efficiency
EneX (Sharma, 2023)	✓	✓	X
DVFaaS (Tzenetopoulos *et al.*, 2023)	✓	✓	X
REAF (Jia & Zhao, 2021)	✓	✓	X
EBAS (Alzahrani *et al.*, 2016)	✓	✓	X
POLAR (Herzog *et al.*, 2021)	✓	✓	X
BEARS (Herzog, Reif, Hügel, Schröder-Preikschat & Hönig, 2022)	✓	✓	X
(Eismann *et al.*, 2020; Elgamal *et al.*, 2018; Lin & Khazaei, 2020)	X	✓	✓
(Sedefoğlu & Sözer, 2021)	X	X	✓
(Das *et al.*, 2020; Lin *et al.*, 2022)	X	✓	✓
This work 3GC	✓	✓	✓

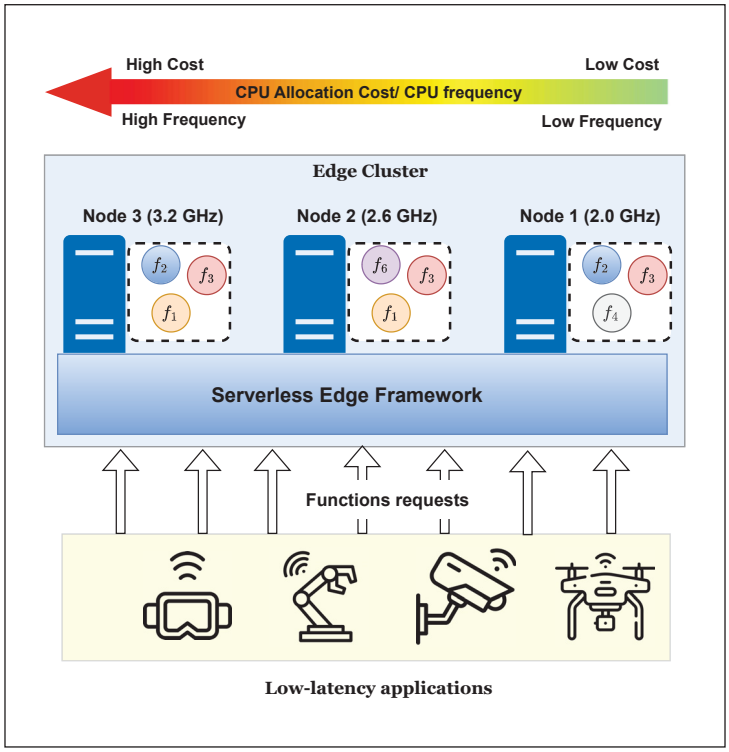


Figure-A I-1 Reference model of 3̃gc

Edge computing environment. We consider a scenario where a serverless provider manages multiple heterogeneous DVFS-capable edge nodes to allocate serverless functions. We denote these edge nodes as $\mathcal{E} = \{e_1, \ldots, e_n\}$. Each edge node e_j supports a range of CPU frequencies, represented by $\Gamma_j = \{\gamma_1^j, \ldots, \gamma_m^j\}$, with the node's computational capability being directly tied to its high frequency, $\Delta_j = \max\{\Gamma_j\}$. Extending this notion to the entire edge cluster, we define $\widetilde{\mathcal{T}} = \cup\{\Gamma_j \mid \forall e_j \in \mathcal{E}\}$ as the aggregate of all frequency sets Γ_j across $\mathcal{E}$, with $\widetilde{\Delta} = \max\{\widetilde{\mathcal{T}}\}$ representing the cluster's highest achievable CPU frequency. The pricing model for resource allocation follows a pay-as-you-go pricing model, with p_j capturing the cost per CPU core at node e_j. This cost is correlated with the

Table-A I-2 Summary of the used symbols

Model Parameters	
$\mathbb{F}$	The set of all functions instances hosted on the edge cluster.
$\mathcal{F}_j$	The set of functions instances hosted on the edge e_j.
$\mathcal{E}$	The set of heterogeneous edge nodes.
Γ_j	The set of CPU frequencies supported at edge node e_j.
γ_m^j	A CPU frequency supported on edge e_j.
Δ_j	The peak CPU frequency of edge node e_j.
$\widetilde{\mathcal{T}}$	The set of all CPU frequencies supported within the edge cluster.
$\widetilde{\Delta}$	The cluster's maximum CPU frequency.
α_i	The average CPU utilization of function f_i.
β_i	The workload nature of function f_i.
δ_i	The execution time of function f_i at maximum frequency.
p_j	The cost per core at node e_j.
$\mathcal{U}_j$	The set of CPU cores at node e_j.
$\sqcap_j$	The set of available cores for allocation at node e_j.
R_i	The set CPU cores reserved for function f_i.
σ_i	The execution time of the function f_i operating under the CPU frequency $\widetilde{\Delta}$.
$\epsilon_j^{i,m}$	The execution time of function f_i at the CPU frequency γ_m^j.
v	The energy price per Kilo Joules in USD.
$\Psi_j^{i,m}$	The energy cost for running function f_i on edge node e_j, with CPU frequency γ_m^j.
$E_j^{i,m}$	The energy consumption of the function f_i with CPU frequency γ_m^j
$\mathcal{D}_j^{i,m}$	The execution cost for function f_i using frequency γ_m^j on edge node e_j.
$\epsilon_j^{i,m}$	The duration of function f_i on edge node e_j using frequency γ_m^j.
ζ^i	The deadline of function f_i.
$\mathcal{P}_j^{i,m}$	The power consumption of function f_i using frequency γ_m^j on edge node e_j.
κ_j	The average switched capacitance per clock cycle at node e_j.
Θ_i	The required CPU cores for deploying the function f_i.
$\mathcal{C}_j$	set of CPU core on edge node e_j.
z_j^i	a decision variable indicating if edge node e_j is a candidate node to host serverless function f_i, 0 otherwise.
$y_m^{n,i}$	a decision variable indicating if frequency γ_m^n is chosen at edge node e_n for serverless function f_i, 0 otherwise.

node's maximal computational power, defined by $p_j = a \cdot \Delta_j + b$, where a and b are flexible parameters set by the provider to tailor the execution costs in line with its business model. Each edge node is equipped with a number of CPU cores to host serverless functions. We represent the set of CPU cores at node e_j as $\mathcal{U}_j = \{u_1^j, \ldots, u_y^j\}$, and use $\sqcap_j \subseteq \mathcal{U}_j$ to denote the set of cores that are currently available for allocation at the same node. With support

for Per-core DVFS, each CPU core within edge node e_j can independently operate at a distinct CPU frequency, separate from other cores within the same node.

Serverless functions. Serverless functions can be deployed across various edge nodes using multiple instances. We define the set of serverless functions instances as $\mathbb{F} = \{f_1, \ldots, f_s\}$ and $\mathcal{F}_j \subseteq \mathbb{F}$ is the set of instances hosted on the edge e_j. In our scenario, serverless function requests are received without prior knowledge of their patterns, requiring the serverless provider to make *on-the-fly* resource allocation decisions. Hence, upon a new request, a new instance of the associated function is created. The resources needed to deploy a function f_i are predetermined by the function's holder. Platforms like AWS Lambda, for example, permit the specification of necessary CPU cores for a function through memory resource settings (AWS). Θ_i denotes the required CPU cores to deploy the function f_i. It is important to note that all CPU cores allocated for the same function should operate with the same CPU frequency. Since these functions are hosted in the edge to serve low-latency applications, meeting the latency requirements is crucial; therefore, ζ_i represents the required latency, or the deadline by which the system must handle a request for function f_i (measured in seconds). Furthermore, σ_i is introduced to denote the execution duration of the function f_i leveraging the peak CPU frequency over the cluster $\widetilde{\Delta}$. Many serverless frameworks allocate resources on a per-CPU core basis to mitigate issues like noise and interference that arise when multiple functions share the same CPU core. However, this approach often leads to over-provisioning; thus, we use α_i to denote the average CPU utilization for the function f_i.

Function Execution Cost. In serverless architectures, the execution cost of functions is closely associated with the characteristics of the allocated resources, such as the maximum supported CPU frequency, and the execution duration of the function. We represent the execution cost for the function f_i on edge node e_j, with allocated CPU cores operating at CPU frequency $\gamma_m^j \in \Gamma_j$, as $\mathcal{D}_j^{i,m}$. This cost is formulated as

$$\mathcal{D}_j^{i,m} = (p_j \cdot \Theta_i) \cdot \epsilon_j^{i,m} \tag{A I-1}$$

where the term $(p_j \cdot \Theta_i)$ indicates the cost per second of using Θ_i CPU core at edge e_j, and $\epsilon_j^{i,m}$ represents the execution time of function f_i at the specified frequency γ_m^j. However, in computational environments, reducing the frequency will extend the task's execution time, but this increase in execution time is not directly proportional to the frequency reduction. The extent to which an application's performance is affected by frequency scaling depends on its CPU-intensive characteristics. Hsu and Kremer (Hsu & Kremer, 2003) introduced the β metric, which quantifies the application slowdown relative to the CPU slowdown. Therefore, we use β_i to express the sensitivity of the function's workload to CPU frequency scaling. In this context, β_i is employed to denote the sensitivity level of a function's workload to changes in CPU frequency. A function with $\beta_i = 1$ is entirely reliant on the CPU, indicating that any alterations in frequency will directly influence its execution time. On the other hand, a $\beta_i = 0$ suggests that the function's execution time is unaffected by frequency scaling. Drawing from equation (1) in (Etinski *et al.*, 2012),

the formula for determining the execution time of function f_i at a specific frequency γ_m^j at node e_j is given by

$$\epsilon_j^{i,m} = [\beta_i \cdot (\frac{\widetilde{\Delta}}{\gamma_m^j} - 1) + 1] \cdot \sigma_i \tag{A I-2}$$

Function Energy Consumption. The energy cost of executing a function is directly linked to the energy consumed during its execution. Thus, we use $\Psi_j^{i,m}$ to represent the energy cost for running function f_i on edge node e_j, with the allocated CPU cores set to operate at CPU frequency $\gamma_m \in \widetilde{\mathcal{T}}$. This cost is given as

$$\Psi_j^{i,m} = E_j^{i,m} \cdot v \cdot 10^{-3} \tag{A I-3}$$

where $E_j^{i,m}$ signifies the total energy consumption by the function f_i during execution, and v is the cost of energy per kilo-joule (expressed in USD). The energy consumption $E_j^{i,m}$ for function f_i is calculated as:

$$E_j^{i,m} = \mathcal{P}_j^{i,m} \cdot \epsilon_j^{i,m} \tag{A I-4}$$

where $\mathcal{P}_j^{i,m}$ refers to the power consumption for function f_i. The power consumption of an edge node comprises both static and dynamic components. The static power, which is constant, is expended even when the CPU is idle. In contrast, our focus primarily lies on the dynamic power, which is relative to the execution of workloads. The dynamic power of node e_j, when all it CPU cores operating at frequency γ_m^j, is expressed as

$$W_m^j = \kappa_j \cdot (\frac{\gamma_m^j}{\Delta_j})^3 \tag{A I-5}$$

Here, κ_j is the average switched capacitance per clock cycle at node e_j. This approach to calculating power usage is commonly used to calculate the power usage of tasks executed on virtual machines (Zhou, Li, Abawajy, Shojafar, Chowdhury, Li & Li, 2022), as well as for applications (Tang, Zhu, Wu, Li & Rodrigues, 2021) deployed in containers such as serverless functions (Rastegar *et al.*, 2023). Consequently, the dynamic power consumed by an individual CPU core on the edge node e_j operating at the CPU frequency γ_m^j is then derived as

$$w_m^j = \frac{W_m^j}{|\mathcal{U}_j|} \tag{A I-6}$$

Note that w_m^j corresponds to the power consumption under the condition of full CPU core utilization. Thus, $\mathcal{P}_j^{i,m}$ is determined by

$$\mathcal{P}_j^{i,m} = (w_m^j \cdot \Theta_i) \cdot \alpha_i \tag{A I-7}$$

Thus, taking into account the number of CPU cores allocated to function f_i and its average CPU utilization.

Problem Formulation and Mathematical Model. In this work, we tackle the online problem of serverless edge computing without prior knowledge of request patterns, aiming to minimize overall costs while meeting function deadlines. Our approach involves *(i)* selecting suitable edge nodes and *(ii)* CPU frequencies for hosting and running serverless functions, respectively. We define an objective function as O_{opt} to minimize the total cost incurred for each serverless function request while ensuring its deadline compliance, based on the following assumptions: *(i)* All CPU cores allocated to a function f_i on edge node e_j operate at the same frequency. *(ii)* Each CPU core on an edge node can be allocated to only one function simultaneously. *(iii)* The execution time of a serverless function stays consistent across different edge nodes when operating at the same CPU frequency. *(iv)* Similar to the AWS Lambda functions operational model (Avalkar, 2022), we assume that each function instance is designed to handle one request at a time.

Objective function and constraints.

$$O_{opt}: \min_{y,z} \sum_{e_j \in \mathcal{E}} \sum_{\gamma_m^j \in \Gamma_j} \sum_{f_i \in \mathcal{F}_k} y_{j,i}^m \cdot z_j^i \cdot (\mathcal{D}_j^{i,m} + \Psi_j^{i,m}).$$

Subject to:

$$\sum_{f_i \in \mathcal{F}_j} z_j^i \cdot \Theta_i \leq \mathcal{U}_j \quad \forall e_j \in \mathcal{E} \tag{C1}$$

$$\sum_{e_j \in \mathcal{E}} z_j^i = 1 \quad \forall f_i \in \mathbb{F} \tag{C2}$$

$$\sum_{\gamma_m^j \in \Gamma_j} y_j^{i,m} = 1 \quad \forall f_i \in \mathbb{F}\,,\ \forall e_j \in \mathcal{E} \tag{C3}$$

$$y_j^{i,m} \cdot \epsilon_j^{i,m} \leq \zeta_i \quad \forall f_i \in \mathbb{F}, \forall e_j \in \mathcal{E}, \forall \gamma_m^j \in \Gamma_j \tag{C4}$$

$$y_j^{i,m} \leq z_j^i \quad \forall f_i \in \mathbb{F}, \forall e_j \in \mathcal{E}, \forall \gamma_m^j \in \Gamma_j \tag{C5}$$

$$y_j^{i,m} \in \{0,1\}, z_j^i \in \{0,1\}, \forall f_i \in \mathcal{F}_j, \forall e_j \in \mathcal{E}, \forall \gamma_m^j \in \Gamma_j \tag{C6}$$

The total CPU cores allocated on any edge node e_j cannot surpass its capacity **(C1)**. Each function is assigned to exactly one edge node **(C2)**. A function f_i on edge node e_j must operate at a CPU frequency within e_j's available set Γ_j **(C3)**. The execution time of a function at its assigned node and frequency must not exceed its deadline **(C4)**. Allocation decisions are binary, ensuring functions are placed on adequate nodes with suitable frequencies, respectively. These are expressed by **(C5 and C6)**.

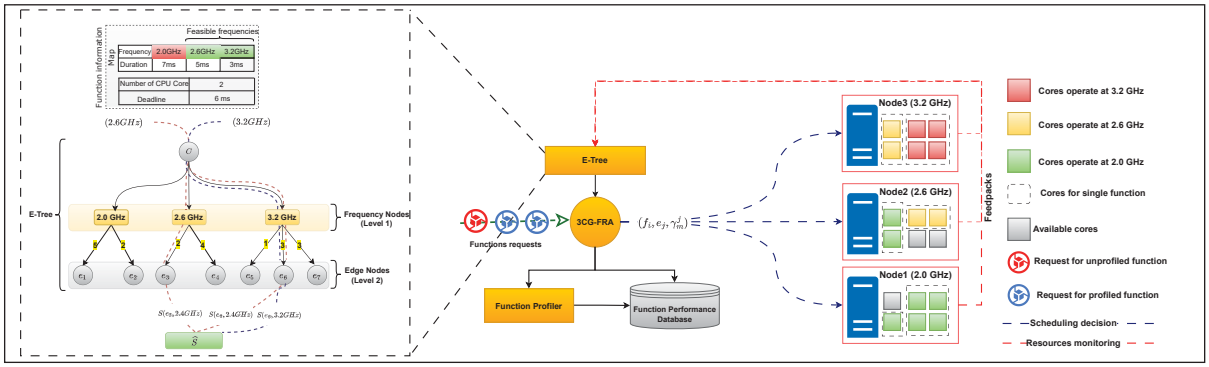


Figure 1.1 Overall architecture of *3CG*.

4. Algorithm design

To address the presented challenges, we introduce a heuristic-based algorithm namely 'Go Green Go Cheap (3GC). 3GC presents a resource allocation scheme for serverless edge computing, facilitating strategic (i) allocation of CPU cores on specific edge nodes and (ii) selection of their operational CPU frequencies to minimize overall costs while meeting functions latency requirements. As illustrated in Figure 1.1, 3GC comprises three primary components: the *Function Profiler* and the *Function Resource Allocation (3GC-FRA)*. The former is responsible for executing requests for non-profiled functions (Section 4). Once a function is profiled, all subsequent requests are handled by the 3GC-FRA (Section 4). We define 'profiled functions' as those for which we can estimate their execution duration at different CPU frequencies. Finally, E-Tree (Section 4), which is a tree structure to obtain a comprehensive view of the edge resource infrastructure.

E-Tree structure. 3GC relies on a tree structure, namely E-Tree, to have a global view of edge resource infrastructure. This tree structure as illustrated in Figure 1.1 serves as a roadmap for the resource allocation algorithm, providing it with a clear hierarchy and order of resources based on their capabilities and current availability. With the help of this tree, 3GC can effectively explore different solutions (i.e., selected edge node, selected CPU frequency) and select the optimal one. E-tree consists of several layers, each representing different aspects of the infrastructure:

1. **Root Node:** Represents the entire edge infrastructure, encapsulating the global view.

2. **Frequency Nodes (Level 1):** Direct descendants of the root node, each representing a unique CPU frequency level. It is ordered from left to right, starting with the highest frequency to the lowest, reflecting the computational power of children's edge nodes.

3. **Edge Nodes (Level 2):** Children of frequency nodes, representing edge computing nodes whose maximum CPU frequency matches that of the parent frequency node. The nodes are sorted from left to right based on their capacities, with nodes of lower average switched capacitance placed on the left. This arrangement guides the scheduling algorithm towards energy-efficient choices, as nodes with lower capacities are typically associated with low energy footprints compared with an edge node with the same maximum CPU frequency but with higher capacity.

4. **Arc Weights:** The weight on an arc connecting a frequency node to an edge node is dynamic and indicates the current number of available cores in that edge node.

Function Profiler. As depicted in Algorithm 3, upon receiving a function request, 3GC determines whether the function has been previously profiled (Line 3). Functions that have not been profiled are executed in profiling mode to determine the nature of the workload, which helps in predicting the execution times of the function at any given CPU frequency. Profiling involves executing the function at two distinct frequencies: the peak frequency of the cluster and an alternative, lower frequency. The duration of the function at these distinct CPU frequencies is used to infer the nature of the workload, which can be derived from Eq (A I-2) and given as:

$$\beta_i = \frac{\gamma_m^j(\epsilon_j^{i,m} - \sigma_i)}{\sigma_i(\widetilde{\Delta} - \gamma_m^j)} \tag{A I-8}$$

To minimize the overhead associated with function profiling, the first execution is repurposed to serve both the function request and the second for profiling requirements. Ideally, the function should be executed at the peak frequency initially to prevent any risk of missing its deadline. However, in scenarios where all CPU cores operating at peak frequency are occupied, 3GC executes the function at the highest available frequency at that moment, thereby mitigating the risk of deadline violation. If the function's first execution (to handle the request) is done at the peak frequency, the second profiling run is conducted at the lowest supported frequency. If not, the function is executed again at the maximum frequency. Executing at the lowest frequency during the second run ensures that CPU cores capable of higher frequencies remain available for other critical tasks. Once the function workload nature is determined, the profiler forecasts the function's execution duration across all frequencies supported by the edge infrastructure. The resulting CPU frequency-duration mapping is stored in the Function Performance Database (FPDB). A function is considered 'profiled' once it secures an entry in the FPDB.

3GC Function Resource Allocation (3CG-FRA). When a function request is already profiled, 3GC retrieves the corresponding frequency-duration map from the FPDB database to identify CPU frequencies that can meet

the function's deadline (Line 4), termed feasible frequencies. An illustrative example is provided in Figure 1.1, showcasing a function that necessitates 2 CPU cores and must adhere to a 6ms deadline. Within the E-Tree, edge nodes are grouped by their maximum supportable frequency, allowing the function scheduler to efficiently navigate through edge nodes within each frequency group. The 3CG-FRA leverages this structure to find the optimal <edge node, CPU frequency> pair that results in the minimum cost. To do so, It starts by setting the initial optimal cost as infinity, and then for each feasible frequency, it calculates the overall cost for edge nodes supporting that frequency. To achieve that it explores the edge nodes per group starting from the group which support that given frequency and given that edge nodes within the group are sorted (from left to right) based capacitance (edge with lower average switched capacitance first). Thus rather than evaluating all edge nodes within the group the 3CG-FRA only calculates the overall cost of the first available node from the left (i.e., the node with enough CPU cores for the function, indicated by the arc's weight) (see Line 6-12). As illustrated in the example presented in Figure 1.1 with $input_1$, node e_6 is the first available node in the group of edge nodes that supports 3.2GHz frequency. The overall cost of allocating a CPU core on edge node e_j and operating at CPU frequency γ_m^j for the function f_i is denoted by $S(e_j, \gamma_m^j)$ and is calculated as:

$$S(e_j, \gamma_m^j) = \mathcal{D}_j^{i,m} + \Psi_j^{i,m} \tag{A I-9}$$

If this resulting cost is lower than the current optimal cost, the scheduler updates the optimal cost and records the edge node and frequency pair as the best solution (Lines 13-16). This procedure is repeated across all edge groups supporting a higher frequency than the given frequency. for example, for a frequency 2.6GHz (see Figure 1.1, $input_2$) , the relevant groups are $G1 : \{e_3, e_4\}$, and $G2 : \{e_5, e_6, e_7\}$. The scheduler evaluates the cost on the first available nodes in each group, e_3 in $G1$ and e_6 in $G2$. This approach ensures that every possible node supporting the function at the required frequency is evaluated.

5. Evaluation

In this section, the performance of 3GC is evaluated in a resource-rich edge computing environment to meet serverless functions' demands. Figure 1.4 illustrates the resource setup for each edge node, detailing CPU cores and their maximum frequencies (label indicates the number of edges that share the same configuration, default is one). Evaluation parameters are listed in Table I-3. 3GC's effectiveness is compared against the following strategies.

Optimal Solution: This is an online algorithm based on the mathematical model, described in Section 3, which exploits Gurobi solver to find i) the optimal frequency leading to energy cost minimization while meeting the deadline of the functions ii) and the cost-effective edge node to schedule its execution.

Kubernetes Default (Default-K8S): This strategy follows Kubernetes' default allocation, assigning function to the first available node with sufficient CPU cores. Nodes are operating at maximum supported frequency by default.

Algorithm 3: 3GC Function Allocation.

Input: f_i: New function instance to deploy.
Global Var: $ETree$: Tree structure encapsulating edge infrastructure.
Output: $Solution$: Optimal node and frequency for deploying f_i.

$\widehat{S} \leftarrow \infty$;
$Solution \leftarrow (\varnothing, \varnothing)$;
// Check if function has been profiled for feasible CPU frequencies
if *Profiled*(f_i) **then**
  // Determine feasible CPU frequencies for f_i
  $\Upsilon_i = \{\gamma_m^* | \forall \gamma_m^* \in \widetilde{\mathcal{T}}, \epsilon_*^{i,m} \leq \zeta_i\}$;
  // the symbol * used to indicate any edge node
  foreach $\gamma_m^* \in \Upsilon_i$ **do**
    foreach $Freq\_Node \in ETree.getChildren()$ **do**
      if $Freq\_Node.Value() \geq \gamma_m^*$ **then**
        foreach $e_j \in Freq\_Node.getChildren()$ **do**
          if $|\sqcap_j| \geq \theta_i$ **then**
            $\gamma_k^j \leftarrow Freq\_Node.Value()$;
            $S(e_j, \gamma_k^j) \leftarrow \mathcal{D}_j^{i,k} + \Psi_j^{i,k}$;
            if $\widehat{S} > S(e_j, \gamma_k^j)$ **then**
              $\widehat{S} \leftarrow S(e_j, \gamma_k^j)$;
              $Solution \leftarrow (e_j, \gamma_k^j)$;
            end
          end
        end
      end
    end
  end
end

Table-A I-3 Evaluation Parameters

Parameter	**Value**
Coefficient a in cost formula	0.000005
Coefficient b in cost formula	0.000005
Energy price per KJ (ν)	0.001$
Number of edge nodes	7
Workload nature (β_i) range	[0–1]
Supported CPU frequencies	[1.0 GHz – 3.6 GHz]
Number of cores per edge	[8–16]
Number of deployed functions	50
Number of requests per function	100
Number of core requests per function	[1–4]
Function deadline range	[5–20] seconds
Average CPU utilization range	[10% – 100%]
Average switched capacitance range	[2 – 10]

Greedy (Execution Cost) (GEC): Aiming to minimize execution costs without sacrificing performance, this method selects the lowest-execution cost node with enough CPU cores, using the node's maximum frequency.

Greedy (Performance) (GP): Focused on maximizing performance within cost constraints. It filters for nodes with enough available cores and sorts them by maximum supported frequency (descending) and then by average

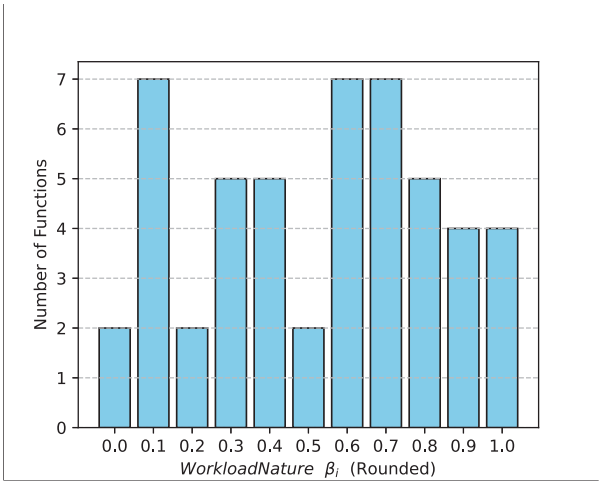


Figure 1.2 Distribution of Workload Nature

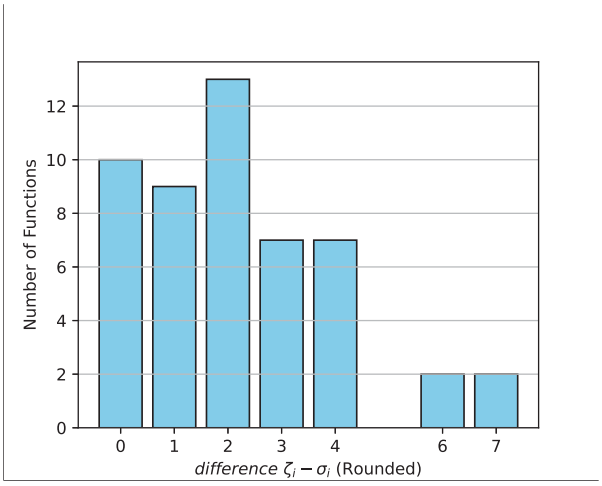


Figure 1.3 Distribution of Deadline

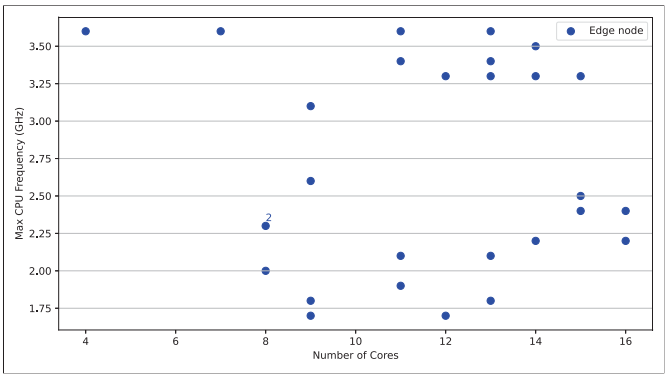


Figure 1.4 Edge node resource configuration

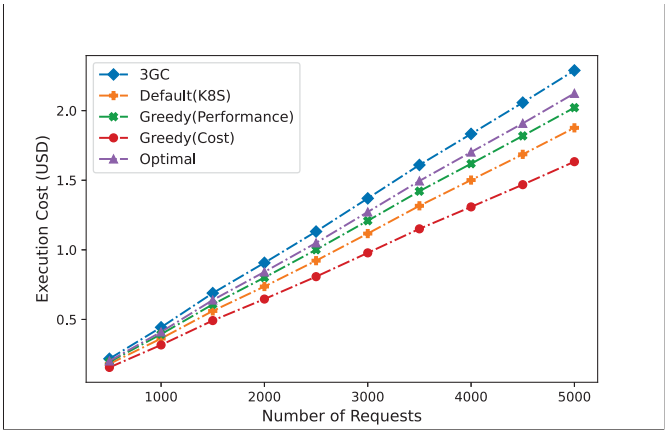


Figure 1.5 Execution Costs Across different Strategies.

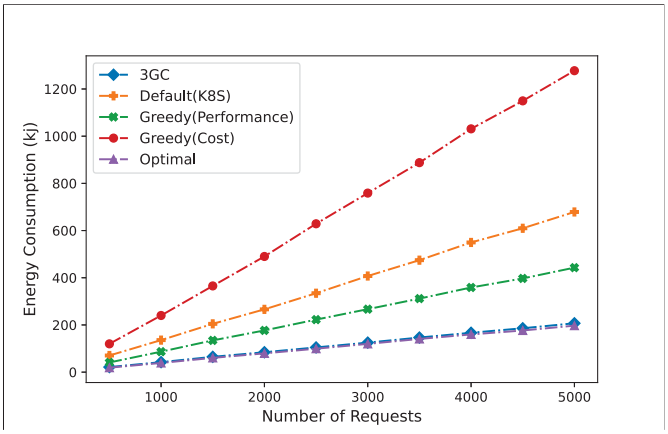


Figure 1.6 Energy Consumption Across Various Strategies.

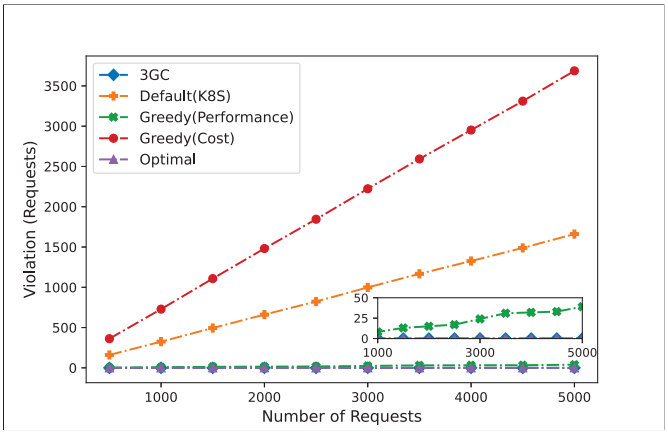


Figure 1.7 Request violation under different Strategies.

switched capacitance (ascending). The function is scheduled on the node with the highest frequency and lowest average switched capacitance among the eligible nodes.

To gauge the performance of 3GC, we select the following evaluation metrics, which are commonly used for the evaluation of scheduling algorithms and resource allocation policies.

Energy Consumption: Energy consumed during function execution.

Execution Cost: The cost (in US dollars) associated with a function execution.

Overall Cost: The total cost (in US dollars), encompassing execution and energy costs related to function operation.

Violation: This measures the incidence of deadline violations, quantifying the number of requests that fail to meet their deadline.

5.1 Results & Discussion

Figure 1.5 shows the execution costs of various baselines versus our strategy. While 3GC results in higher execution costs compared to others, it balances execution expenses with energy savings and latency compliance. The slight increase in 3GC's execution cost over the optimal is due to the profiling requirement of executing each function twice. GEC and Default-K8S have the lowest execution costs, whereas GP's costs are higher because it prioritizes the use of the most computationally powerful—and thus most expensive—CPU cores.

Figure 1.7 illustrates the number of requests that fail to meet their deadlines under each strategy. Our solution, alongside the optimal one, results in no violations, demonstrating that all requests meet their deadlines. Although GP opts for the maximum available frequency to maximize performance, not all functions benefit equally from higher frequencies. For instance, memory-bound workloads benefit less from high frequencies compared to CPU-bound workloads, and some functions have relaxed deadlines, allowing for more flexible CPU frequency adjustments. Hence, running a non-CPU-bound workload or a function with relaxed deadlines on a higher CPU frequency is not only considered over-provisioning of CPU cycles but also may prevent forthcoming functions that could require this high-frequency CPU core. This explains the 2.14% of request violation (214 request) observed with GP. Default-K8S and GEC led to 14.32% and 17.09% violations of the total requests received during the evaluation, respectively. This was expected since GEC selects low-cost CPU cores with lower computational capabilities, extending execution times and causing deadline misses. Conversely, Default-K8S does not account for low-level features such as CPU frequency of the allocated core, leading to indiscriminate function distribution without considering computational performance.

Figure 1.6 highlights the energy consumption across various baselines compared to 3GC. Our solution outperforms others in terms of energy efficiency, achieving minimal consumption that aligns with the optimal solution. Despite GEC's strategy to host functions on low-priced cores with lower computational power and thus lower CPU frequency, leading to reduced power consumption, the long duration of function execution results in higher energy usage, given that energy is the product of power over time. Default-K8S exhibits a similar trend in energy consumption. While GP slightly surpasses our solution in execution cost and matches the latency achievement, it incurs significantly higher energy consumption compared to 3GC. This evaluation emphasizes our focus on minimizing energy usage, given its substantial impact on overall costs, contrary to other baselines that may optimize execution costs but result in elevated energy consumption, potentially leading to increased total costs.

Figure I-2 presents the overall costs associated with different strategies, demonstrating that our solution aligns with the optimal one regarding delivered overall cost. Our approach results in a total of 24 USD throughout the evaluation, which is on par with the 22.5 USD of the optimal solution. This links to a significant cost reduction of 69.35% compared to the GEC strategy, which incurs the highest expenses 78.44 USD. Moreover, cost reductions of 39.35% and 54.77% were observed with Default-K8S and GP, respectively. In comparison to other strategies, 3GC achieves the maximum cost reduction while maintaining zero request violations, thereby matching the optimal performance (71.25% overall cost reduction). To further dissect the overall costs, we analyzed the accumulated costs over the evaluation period (spanning 10,000 requests) into execution and energy costs. Figure I-3 delineates this breakdown, spotlighting the equitable distribution between energy and execution costs in our solution, where they contribute 12.5 USD (45.7%) and 11.5 USD (54.3%) to the overall cost, respectively.

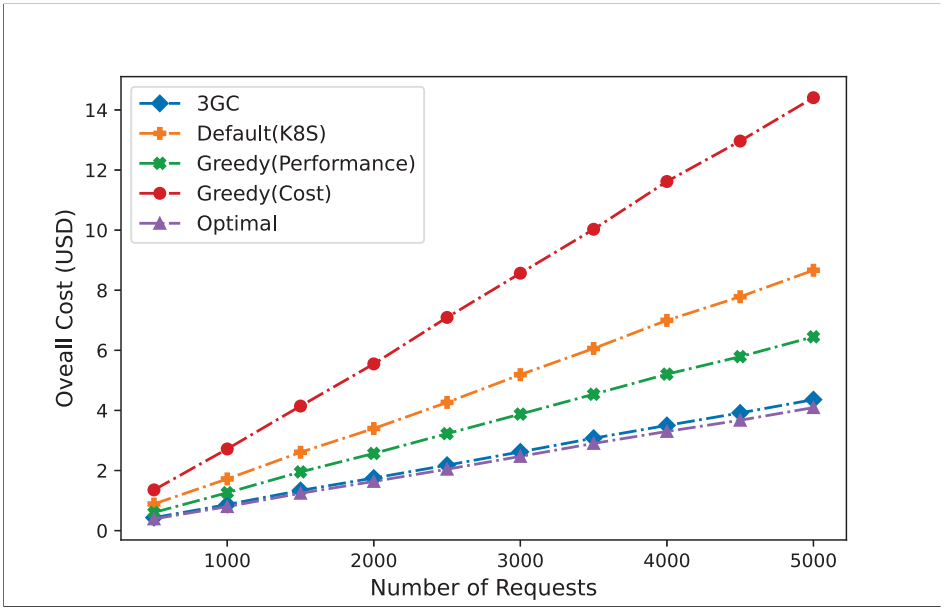


Figure-A I-2 The Overall Costs by Strategy

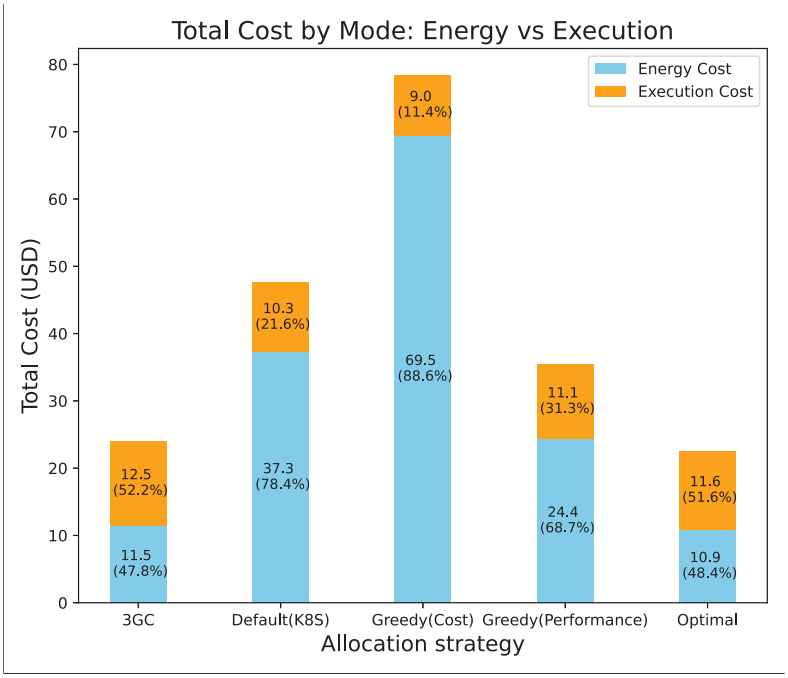


Figure-A I-3 Detailed Breakdown of Overall Costs for Each Strategy

In comparison, other solutions failed to maintain this balance, which led to an increase in their total overall costs. Specifically, only 9, 10.3, and 11.1 USD were spent on execution costs for GEC, Default-K8S, and GP respectively, but a significant portion of their costs was driven by energy expenses. The energy costs for GEC, Default-K8S, and GP were 69.5, 37.3, and 24.4 USD, respectively, representing 88.6%, 78.4%, and 68.7% of their total costs, respectively. These findings highlight the effectiveness of our solution not only in meeting latency requirements but also in optimizing costs. Our strategy prioritizes energy efficiency, acknowledging its profound impact on overall expenses, in contrast to other approaches that might focus on optimizing execution costs at the risk of elevating energy consumption, thereby potentially escalating total costs. Our solution stands out for its ability to fulfill latency demands efficiently while maintaining cost-effectiveness.

6. Conclusion

In conclusion, our 3GC solution has demonstrated high efficiency in reducing the overall cost while meeting strict latency requirements. it reduce the overall cost up to 64% This work underscores the importance of considering workload characteristics and deadlines in devising resource allocation strategies. It highlights the crucial need to extend beyond mere execution cost and performance optimization, aiming instead to enhance energy efficiency. For future work, we strive to validate within real-world scenarios, employing serverless benchmarks and leveraging open-source serverless frameworks like OpenWhisk. We believe that our approach not only offers an economical choice to run serverless functions but also lays the groundwork for sustainable serverless computing practices.

APPENDIX II

EC6: ENHANCING ENERGY EFFICIENCY IN KUBERNETES THROUGH DYNAMIC EXTENSION OF CPU DEEP IDLE STATES (C6)

Zouhir Bellal, Laaziz Lahlou, Nadjia Kara, Timothy Murphy and Tan Phat Nguyen

Département de Génie Électrique, École de Technologie Supérieure,

1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Ericsson, Canada

Abstract:

Although the energy footprint of cloud infrastructures is becoming a critical concern, mainstream orchestration platforms such as Kubernetes (K8s) continue to prioritize performance over power efficiency. By default, K8s employs static CPU pinning for latency-sensitive services to ensure predictable performance and prevent resource contention. However, during service idle periods, residual background activities—such as runtime threads and maintenance tasks—occupy the assigned cores, preventing them from entering deep idle states (e.g., C6) and leading to unnecessary power consumption. This paper introduces EC6, a lightweight, idle-aware scheduling policy for K8s that consolidates idle services onto a reserved single CPU core, allowing other cores to transition into deep C-states during service inactivity. Upon resumption of service activity, EC6 transparently restores services to their original CPU cores with minimal latency, ensuring resource allocation constraints (*i.e.,* , number of CPU cores) are maintained. Experimental evaluations show that EC6 increases per-core C6 residency by up to 30% and reduces idle power consumption by up to 13.6% compared to K8s' default container scheduling policy, without compromising performance. This demonstrates its effectiveness in improving energy proportionality in cloud-native environments while preserving service quality.

Keywords: Kubernetes Scheduling; Energy Efficiency; CPU C-states; Cloud-Native Services

1. Introduction

Cloud data centers are consuming an increasing amount of energy worldwide, raising serious environmental and cost concerns. They currently use about 1–2% of all electricity globally. However, these numbers may be too low, especially with the rapid rise of energy-intensive AI tasks, such as generative models. For example, one ChatGPT query can use about 25 times more energy than a regular web search (Saenko, a). Experts predict that by 2027, AI could push global data center energy use to 68 GW, and up to 327 GW by 2030 (Pilz *et al.*, 2025). This fast

growth and the need to fight climate change make it crucial to focus on energy-efficient computing in the cloud for running cloud-native services. Cloud-native services are widely deployed through containerization, which enables lightweight and portable execution environments on multiple hardware platforms (Pahl, Brogi, Soldani & Jamshidi, 2019). Kubernetes (K8s), the de facto standard for container orchestration, provides robust mechanisms for dynamic scheduling, resource allocation, and high availability (Burns, Grant, Oppenheimer, Brewer & Wilkes, 2016). Although K8s scheduling policies are generally tuned for performance and fault tolerance, energy efficiency remains a largely overlooked dimension (Bellal, Lahlou, Kara, Murphy & Nguyen, 2025b). In performance-sensitive scenarios, K8s supports the Guaranteed Quality of Service (QoS) class, where dedicated CPU cores are statically assigned to the service to ensure isolation and predictable performance (Kubernetes, 2025). While this static allocation model minimizes contention and meets stringent Service Level Objectives (SLOs), it constrains the CPU's ability to enter deep power-saving idle states. Despite this shortcoming, modern processors support multiple idle states (C-states), with deeper states such as C6 and C9 offering substantial power savings (Intel Corporation, 2017). However, keeping the CPU core in deeper C-states requires sustained, uninterrupted periods of idle time for the CPU. Unfortunately, the prerequisites for staying deep C-states are rarely satisfied when CPU cores are statically pinned to services under the Guaranteed QoS class (Lefurgy, Wang & Ware, 2007). K8s's default static CPU pinning policy maintains a persistent core-to-service binding, even when the assigned service is idle (Podzimek, Bulej, Chen, Binder & Tuma, 2015). For instance, a RESTful API service may remain idle due to the absence of incoming requests but still requires periodic CPU cycles to maintain its runtime environment. We define such services as idle burner services -services that appear idle from a functional or user-facing standpoint but continue to consume CPU resources to obey some logic defined by design. A common example is the Java Virtual Machine's garbage collector, which may execute background memory management tasks regularly, even without active functional activity (e.g., handling HTTP requests for a web service) (Sharafzadeh, Kohroudi, Asyabi & Sharifi, 2019). This tiny activity prevents cores from reaching deeper C-states, inhibiting hardware-level energy optimization (Haj-Yahya, Volos, Bartolini, Antoniou, Kim, Wang, Kalaitzidis, Rollet, Chen, Geng et al., 2022).

To address this limitation, we propose EC6 (Extended C6 State), an energy-aware scheduling approach specifically designed for idle service management in K8s to mitigate this inefficiency. EC6 continuously monitors service activity metrics (e.g., incoming requests, system calls) and dynamically migrates idle pods to designated isolation cores. This enables the original cores, statically allocated for Guaranteed QoS, to transition into deeper idle states. Once activity resumes, EC6 seamlessly restores the service's pods to their original CPU cores, thereby preserving performance guarantees. Experimental evaluations on a multi-core testbed demonstrate that EC6 reduces total system power consumption by up to 13.6% without incurring performance degradation, validating its effectiveness for sustainable and performance-compliant cloud-native operations.

The remainder of this paper is organized as follows. A background and related work are briefly described in Section 2. The proposed EC6 is given in Section 3, followed by a proof-of-concept (PoC) implementation in Section 3.3.

Section 4 presents the methodology used to illustrate the usefulness of our experimental evaluation. Section 5 presents and discusses the main results. Finally, conclusions are given in Section 7.

2. Background & Related Work

2.1 CPU idle states (C-states)

Modern CPUs support multiple idle power modes, known as C-states, which shut down various core and uncore components when the processor is not actively executing instructions (Antoniou, Bartolini, Volos, Kleanthous, Wang, Kalaitzidis, Rollet, Li, Mutlu, Sazeides et al., 2024). The deeper the C-state entered, the greater the potential energy savings; however, deeper C-states incur longer wake-up latencies before returning to the active state (C0), as summarized in Table II-1.

To transition into each C-state, the CPU core must remain idle and uninterrupted for a specific duration, referred to as the target residency. For example, on Intel Skylake processors (See Table II-1), transitioning from C0 to C1 requires the core to remain idle for at least 2μs, whereas entering C6 requires an idle duration of 600μs. In Linux, the cpuidle and CPUfreq subsystems determine the appropriate idle state based on the estimated idle duration (Karpowicz, 2016). Any interrupt request (IRQ) or scheduling activity immediately triggers an exit from the idle state toward either the active (C0) mode or a shallower idle state, such as C1 or C1E. However, low-level software control over C-state selection is constrained, as transitions are primarily governed by hardware firmware policies and OS-level heuristics based on the runtime conditions.

Table-A II-1 C-States on Intel Skylake Server Cores (int)

C-State	**Wake-up latency**	**Target residency time**	**Power per Core**
C0 (High CPU freq)	N/A	N/A	4W
C0 Low CPU freq	N/A	N/A	1W
C1	2μs	2μs	1.44W
C1E	10μs	20μs	0.88W
C6	133μs	600μs	0.1W

Several research efforts have investigated using C states to improve energy efficiency, particularly in latency-sensitive services (Asyabi, Bestavros, Sharafzadeh & Zhu, 2020; Chou, Bhuyan & Wong, 2019; Fujimoto, Harasawa, Natori, Otani, Saito & Shiraga, 2022; Govindaraj, George, Kandemir, Sampson & Naryanan, 2021; Kaffes, Sbirlea, Lin, Lo & Kozyrakis, 2020; Sharafzadeh *et al.*, 2019). Sharafzadeh et al. propose Yawn (Sharafzadeh *et al.*, 2019), an OS-level idle-state governor that uses machine learning to predict CPU idle durations and mitigate tail latency without sacrificing power. The PACT framework (Kaffes *et al.*, 2020) and Peafowl (Asyabi *et al.*, 2020) consolidate best-effort workloads onto fewer cores during low load, explicitly invoking the OS C-state governor so that inactive

cores enter deep idle, thereby saving power. In contrast, PowerPrep (Govindaraj *et al.*, 2021) targets hardware, introducing a novel cache-retaining deep-sleep mode (via Non-Volatile Memory (NVM)) to meet the low wake latency, low residency power, and cache retention requirements of latency-critical workloads. Fujimoto et al. (Fujimoto *et al.*, 2022) present PWU, which 'pre-wakes' a core before assigning work, dramatically reducing recovery latency from C6 (by 84%) in virtualized RAN tests. µDPM proposes scheduling techniques tailored for ultra-low latency applications, which aggressively transition CPU cores into deep C-states immediately upon idleness (i.e., when no pending requests) and retain them until new work arrives (Chou *et al.*, 2019). However, this depends on custom schedulers and fine-grained per-core power control mechanisms, which are not available in standard Linux distributions. Additionally, even idle microservices often perform essential background tasks, such as heartbeat signaling or security certificate renewal. Forcing deep sleep immediately after request execution may prematurely terminate these background tasks, potentially disrupting necessary operations required to maintain service availability.

Hardware-level modifications have been proposed to reduce the latency penalty of deep idle states. One such approach, AgileWatts/C6A (Antoniou *et al.*, 2024), is a redesigned C6 state that preserves the CPU's microarchitectural context, which is typically discarded. This modification significantly reduces wake-up latency and energy overhead associated with exiting deep sleep. Consequently, this makes C-state transitions less costly, enabling more aggressive power management policies without violating strict tail latency Service Level Agreements (SLAs). However, as a hardware-centric solution, its adoption is limited as it requires an entire hardware modification. Other work focuses on dynamically tuning C-state selection policies in software. Traditional systems use a fixed residency time threshold to determine whether to enter a deep C-state. In contrast, DynSleep (Chou, Wong & Bhuyan, 2016) adjusts this threshold based on the workload's characteristics and the predicted slack. By dynamically assessing the trade-off between power savings and wake-up latency, DynSleep attempts to select the optimal C-state for any given idle period. This approach focuses on optimizing the C-state decision on cores with fragmented idle time. Instead of tuning the policy for a busy core with fragmented idle time, EC6's consolidation strategy creates a completely empty core where the most aggressive C-state policy (i.e., always go to C6) becomes the obvious and optimal choice. Moreover, EC6 differs from these approaches by operating at the container/orchestration level: It migrates idle K8s pods to a dedicated "isolation" CPU core, allowing the original cores to enter prolonged C6 sleep and then restoring them on demand (i.e., when service activity resumes). In other words, EC6 achieves deep idle residency through scheduling policy (pod migration) and shares the common goal of saving idle power without degrading latency.

2.2 CPU Resource Management and Energy Efficiency in K8s

K8s has become the standard orchestration platform for deploying performance-sensitive services using Guaranteed QoS class pods, where CPU resources are statically assigned at deployment (Kubernetes, 2025). K8s traditionally considers these resource assignments immutable post-deployment; thus, CPU resources allocated at startup persist

irrespective of subsequent utilization changes. For instance, a service initially requiring multiple CPU cores remains allocated those cores even after the load diminishes significantly, resulting in wasted resources and power. Pods configured with equal CPU requests and limits under the Guaranteed QoS policy gain exclusive use of assigned CPU cores through K8s' static CPU management policy. K8s uses CPU pinning via Linux CPU sets to enhance cache affinity and performance predictability, restricting container processes to specific CPU cores (Intel Corporation, 2019b). Although advantageous for performance, this fixed core assignment complicates power optimization efforts during idle periods, especially for microservices characterized by persistent minimal background workloads. Current K8s power optimization initiatives predominantly target active workload scheduling (Bellal, Lahlou, Kara & Khayat, 2024b), leaving the energy savings potential during idle phases largely unexplored. Intel's K8s Power Manager (Corporation, 2022) is an open-source operator designed for dynamic power management within Kubernetes (K8s), allowing the use of hardware-level power features such as Speed Select Technology and per-core frequency scaling. Similarly, RedHat's OpenShift Per-Pod Power Management (Hat, 2022) annotates latency-critical pods to maintain their cores in active states (C1 with turbo mode enabled), allowing non-critical pods to leverage deeper C-states and reduced frequencies. However, these solutions cannot dynamically re-assign cores during runtime or address persistent minimal-load scenarios inherent to idle burner services, highlighting a significant gap that this research aims to address.

2.3 Idle Burner Services

Cloud-native services frequently exhibit non-negligible CPU consumption even during idle periods, due to continuous background operations inherent to their runtimes or business logic. Such services, which we call *idle burner services*, maintain minimal CPU activity, preventing the core from entering a prolonged idle period. Different reasons might influence this behavior, including but not limited to:

- **Heavy Runtime Frameworks:** Java applications, such as those using Spring Boot, often incur considerable idle CPU usage from JVM-related operations like garbage collection and class loading (Hot). Python-based microservices using frameworks such as Flask or Django similarly experience idle CPU overhead due to Python's Global Interpreter Lock (GIL) and event loop polling mechanisms (Pyt).

- **Poor Code quality:** Applications employing inefficient timers, frequent polling loops, or unnecessary asynchronous tasks contribute significantly to bursts of CPU activity. For example, Python or Node.js event loops may wake periodically to check timers or handle asynchronous I/O (**?**).

- **Business Logic Constraints:**

  In microservices-based systems, some services must perform regular background tasks—such as refreshing feature flags or maintaining persistent connections—to ensure system reliability and up-to-date configurations, even in the absence of external workload. For instance, local caching of dynamic data to reduce latency, or

regularly polling for certificate updates, requires periodic background activity to maintain consistency across distributed services (Falkevych & Lisniak, 2025).

These idle burner services, especially when coupled with static K8s granted deployment, limit deep C-state power savings. Hence, the CPU may frequently stall only in shallow idle states (C1/C1E) rather than in deeper C6 states. Thus, this substantially limits potential power savings from idle periods in Kubernetes environments. As observed in (Pijnacker, 2024), containers that appear idle can still draw power unexpectedly.

3. EC6 Architecture and PoC Implementation

Fig. II-1 illustrates the concepts underlying EC6 and its core functional modules.

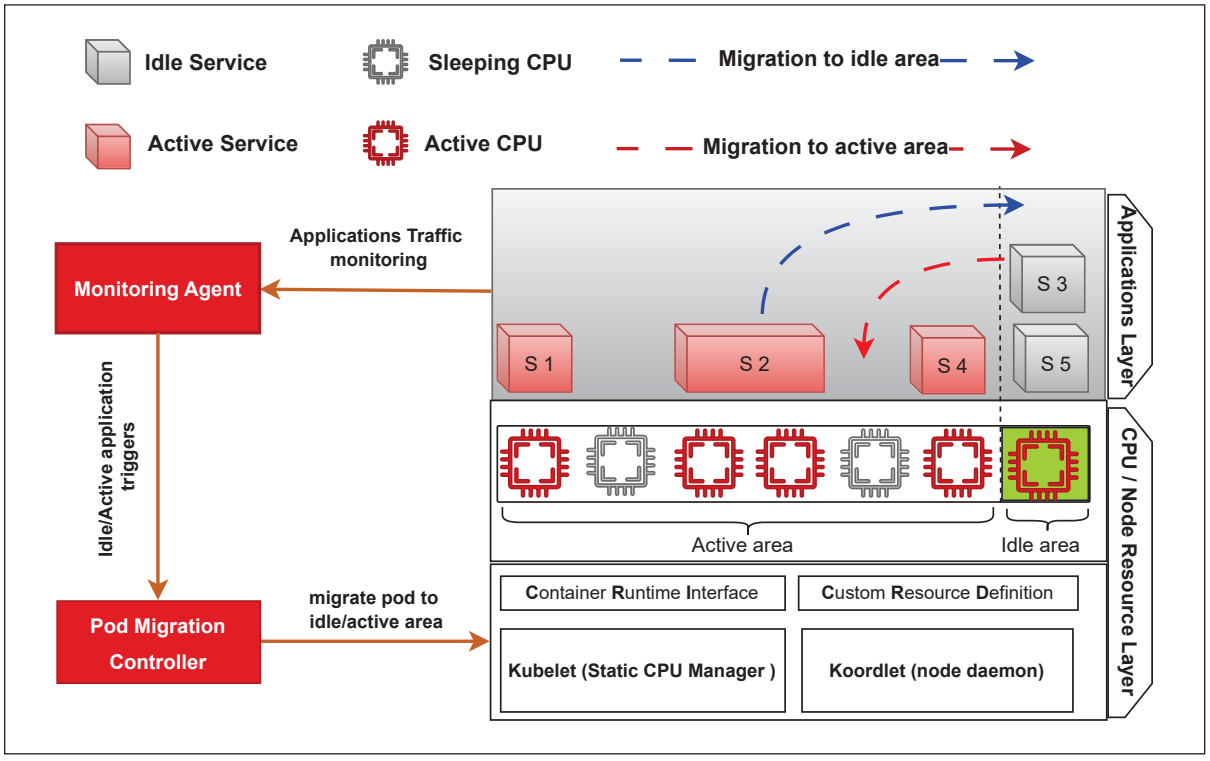


Figure-A II-1 Overall architecture of EC6

EC6 aims to improve energy efficiency by extending the duration of deep C-state residency across CPU cores when services are operationally idle (e.g., a web service waiting for incoming requests). C-state residency refers to the period when a CPU core remains in a deep C-state during the system's idle state. Directly controlling CPU C-states at the pod level is not feasible, as the operating system or CPU firmware governs C-state transitions per core. However, EC6 enables an indirect mechanism: when a service pod becomes idle, it is migrated to a CPU core designated for idle services. This allows the originally allocated cores to become completely idle, enabling them to

enter deeper sleep states. EC6 is deployed at the K8s cluster level using a `DaemonSet` across all worker nodes. Its architecture consists of two primary components: the **Service Monitoring Agent**, which detects service idleness based on application-layer activity, and the **Pod Migration Controller**, which manages CPU resource updates and pod migration actions in response to detection of idle/active state.

3.1 Service Monitoring Agent

The Service Monitoring Agent is responsible for continuously observing application-level behavior to detect idleness of the pod. Detecting idleness in microservice-based cloud-native applications is nontrivial due to background tasks such as health checks, logging, or security token refreshes, which persist even when services are not actively processing external requests. Hence, EC6 utilizes application-layer telemetry for precise and context-sensitive activity evaluation, integrating the Kubernetes Event-Driven Autoscaler (KEDA). This open-source extension enables fine-grained scaling decisions based on external event signals. KEDA supports a wide range of event sources, including message queues (e.g., Kafka), database queries (e.g., PostgreSQL), and telemetry systems (e.g., AWS CloudWatch, HTTP request rate).

In EC6, each monitored service is registered with a unique identifier and linked to a specific application-level metric—one that reflects operational activity. For example, an HTTP-based web service may use an HTTP scaler to track inbound request rates, while REST APIs can use invocation frequency as a proxy for activity. This design enables accurate detection of the service's operational state (*i.e.,* , Active/Idle).

3.2 Pod Migration Controller

CPU-aware pod migration is a core mechanism in EC6. EC6 splits the CPU pool of a socket into two logical zones:

- **Active Area:** This zone includes CPU cores dedicated to active services. These cores remain visible to the default Kubernetes scheduler.
- **Idle Area:** This zone typically consists of a single shared CPU core allocated to idle services. During idle periods, resource requests—such as the minimum number of CPU cores required for guaranteed performance—are relaxed, allowing multiple services to share a single core.

When the Service Monitoring Agent detects that an active service has become idle for a predefined duration, customizable based on workload function burst, it triggers the Pod Migration Controller to perform a two-step migration process, which first releases the pod's CPU resource requests and clears its CPU affinity bindings (number of required cores), freeing up the original exclusive cores, and then migrates the pod to the idle core zone.

In EC6, when a service transitions to an idle state, its CPU resource requests are relaxed, allowing multiple idle pods to share a single core within the idle area. Since CPU allocation is elastic, unlike memory, reducing CPU reservations does not impact pod stability or lifecycle. The minimal CPU provision is sufficient to support minimal background activities and keep the services alive. When service activity resumes—triggered by incoming requests or application-specific events—the Service Monitoring Agent signals the Migration Controller to restore the pod to its original CPU core. The controller maintains records of each service's original core placement and promptly reinstates CPU resource requests and core affinity. This process typically incurs negligible latency, experimentally measured at less than 0.03 milliseconds (ms).

Although recent K8s versions introduced *In-Place CPU Resource Updates* (Kubernetes Authors, 2024), allowing for the dynamic resizing of CPU resources without requiring pod restarts, these native features do not fully support runtime pod migration or CPU core re-pinning. EC6 utilizes Alibaba's ACK Koordinator (Koordlet) (Alibaba Cloud Documentation, 2024), an advanced K8s add-on designed for dynamic runtime resource adjustments via Custom Resource Definitions (CRDs) and controllers to address this limitation. Koordlet can modify both CPU resources and CPU affinity of running pods without requiring pod restarts. This is accomplished through the direct manipulation of Linux `cgroup` settings (e.g., CPU quota, `cpuset`) via the Container Runtime Interface (CRI), thereby overcoming the default immutability constraints of K8s pod specifications. Therefore, using Koordlet's capabilities, EC6 efficiently manages resource allocation dynamically and seamlessly, enhancing cluster-wide energy efficiency.

3.3 EC6 PoC Implementation

While deploying EC6 on a fully operational K8s cluster with all its components is technically feasible, we opted for a simplified experimental environment to focus solely on validating the core concept. The primary objective of this PoC is to demonstrate the effectiveness of EC6's central mechanism. To abstract away the orchestration complexity inherent in full K8s deployments, we implemented the PoC using a container-based testbed. Rather than working with actual K8s-managed pods, we emulate their behavior through standalone containers. This setup enables low-level, direct control over container placement and runtime properties without requiring the full K8s control plane.

The PoC focuses exclusively on the core operational action of EC6—namely, the transition of a container from one CPU core to another. Specifically, containers are initially pinned to a designated core using the `cpuset` flag during launch, and later "migrated" by dynamically updating their CPU affinity using runtime commands (e.g., `docker update`). This emulates the pod migration process central to EC6, while abstracting away other layers of automation.

It is important to note that the event detection and monitoring subsystem, which in a full EC6 deployment would identify when a service becomes idle or active, is not implemented in this PoC. We treat such events as external triggers, i.e., outside the scope of this experiment. Instead, we assume a simplified, idealized setting in which all containers remain idle throughout the evaluation period. This allows us to isolate and precisely assess the energy impact of the migration mechanism itself, independent of the complexity introduced by real-time service activity monitoring.

4. Evaluation Methodology

To evaluate the power-saving impact of EC6, we conducted experiments on a real-world testbed. The hardware and system configurations are detailed in Table II-2. To ensure reliable and repeatable measurements, the test environment was carefully isolated and configured to minimize power measurement noise and eliminate interference from non-experimental workloads (e.g., OS background processes).

Table-A II-2 Testbed Configuration Summary

Attribute	Specification	Attribute	Specification
CPU Model	Xeon Gold 6132	Threads/Core	2
Cores/Socket	28	OS	Ubuntu 22.04.5
Uncore Freq (GHz)	2.4 (fixed)	CPU Freq (GHz)	2.6 (fixed)

4.1 Experimental Setup Configuration:

Several hardware and operating system-level settings were applied to ensure measurement stability:

- **Turbo Boost Disabled:** Intel Turbo Boost was disabled to maintain a stable core frequency and eliminate transient power fluctuations. Since Turbo Boost can cause temporary power spikes and lead to thermal throttling, disabling it ensures consistent and deterministic power measurements.
- **Performance Frequency Governor:** To ensure consistent power measurements, the CPU frequency governor was set to `performance` mode, which locks the processor at its maximum frequency and disables dynamic frequency scaling.
- **Fixed Uncore Frequency:** The uncore frequency was statically fixed at its maximum supported value (2.4 GHz) during all experiments to eliminate variability stemming from uncore subsystem behavior, including L3 cache access and memory bandwidth.

- **Swap and Page Cache Disabled:** All workloads were confined to physical memory. To ensure deterministic execution and improve measurement reproducibility, both swap memory and page caching were disabled, eliminating variability from disk I/O and paging overhead.

4.2 CPU Socket Isolation:

The testbed uses a dual-socket architecture to isolate one socket for experimental workloads by excluding its CPU cores from the Linux scheduler via the `isolcpus` boot parameter. This setup ensures that the experiments are isolated from interference caused by OS background processes at the socket level. Within the isolated socket, cores are logically divided into two operational zones:

- **Active Area:** CPU cores 0 to 26 were reserved for hosting active service containers
- **Idle Area:** CPU core 27 was designated as the idle zone. This core exclusively hosted service container is identified as idle by EC6.

4.3 Idle Burner Services

To emulate real-world Idle Burner Services, we designed three distinct services, each reflecting a different cause of residual CPU usage commonly observed in production environments.

1. **Business Logic Constraints – RefreshService:**
   The RefreshService simulates a realistic idle-burner microservice. Its primary function reflects business-layer requirements commonly found in production-grade systems: (1) maintaining always-warm downstream connections to external APIs to ensure ultra-low response latency on first use, and (2) periodically refreshing application-level configurations—such as feature flags—from a remote JSON-based source of truth. To fulfill these objectives, the service implements two scheduled tasks: one that sends a lightweight heartbeat (GET) request every 25 ms to a target endpoint, and another that simulates a 50 KB feature flag refresh, performing multiple rounds of SHA-256 hashing to validate consistency before parsing the data. Although each task is lightweight in isolation, the high frequency and deliberate CPU usage result in continuous wake-ups, interrupting the deep sleep state of the CPU. The service is deployed on a runtime-heavy platform, such as Spring Boot, with additional features including DevTools, file watchers, and garbage collection.
2. **Heavy Runtime Frameworks – Payara Server:**
   This service illustrates idle CPU consumption inherent in enterprise-grade runtimes. The Payara Server (Payara Team, 2024), a full-stack Java EE platform, activates numerous internal processes—such as autodeploy scanning, JMX monitoring, and JMS clustering—even when no external requests are served. These background

threads generate periodic activity that keeps CPU cores from entering deep sleep states. Payara thus represents an infrastructure-induced source of idle CPU utilization.

3. **Poor Code Quality – stress-ng:**
   To model idle workloads resulting from suboptimal implementation practices, we use stress-ng (Ubuntu Manpage Repository, 2024) to simulate a recurring task: 1 ms of CPU load followed by 10 ms of sleep. This behavior mimics inefficient polling loops or frequent state checks found in low-quality service code. The frequent transitions between active and idle states prevent effective C-state entry, highlighting the impact of software inefficiencies on power consumption.

These load generators collectively reflect a range of idle-time CPU patterns encountered in real systems, forming the basis for evaluating EC6 under practical, heterogeneous idle service scenarios.

This experiment aims to evaluate the power-saving efficacy of EC6 in comparison to K8s' default container scheduling policy. The evaluation focuses on two dimensions:

- **Diversity of Idle Workloads:** We examine the impact of EC6 on different types of idle services. Each service exhibits distinct CPU usage patterns (e.g., background activity, polling), which influence the CPU core's ability to enter deep sleep states (C-states).
- **Scalability with Respect to Idle Load:** For each service, we incrementally increase the number of concurrently deployed idle service instances. This enables analysis of power consumption as idle service density increases.

A controlled experiment was designed to systematically measure power consumption and CPU C-state residency under both scenarios: EC6 and the default K8s scheduler, to ensure reliable results.

4.4 Experimental Procedure

The experimental workflow, depicted in Fig. II-2, is executed iteratively for each Idle Burner Service as follows:

1. **Step 1: Initialization**
   An idle service instance is deployed and pinned to one CPU core in the active area, beginning with CPU 0 (Step ❶ in Fig. II-2). A 10-second delay is introduced to allow for system stabilization and eliminate transient startup effects.
2. **Step 2: Dual-Phase Measurement**
   To measure power consumption and C-state residency distribution for each scenario, we used turbostat. turbostat was selected due to its lightweight overhead compared to tools such as Intel PCM (Intel Corporation, 2024) and its ability to provide high accuracy with fine-grained profiling. A sampling interval of 0.1 seconds was

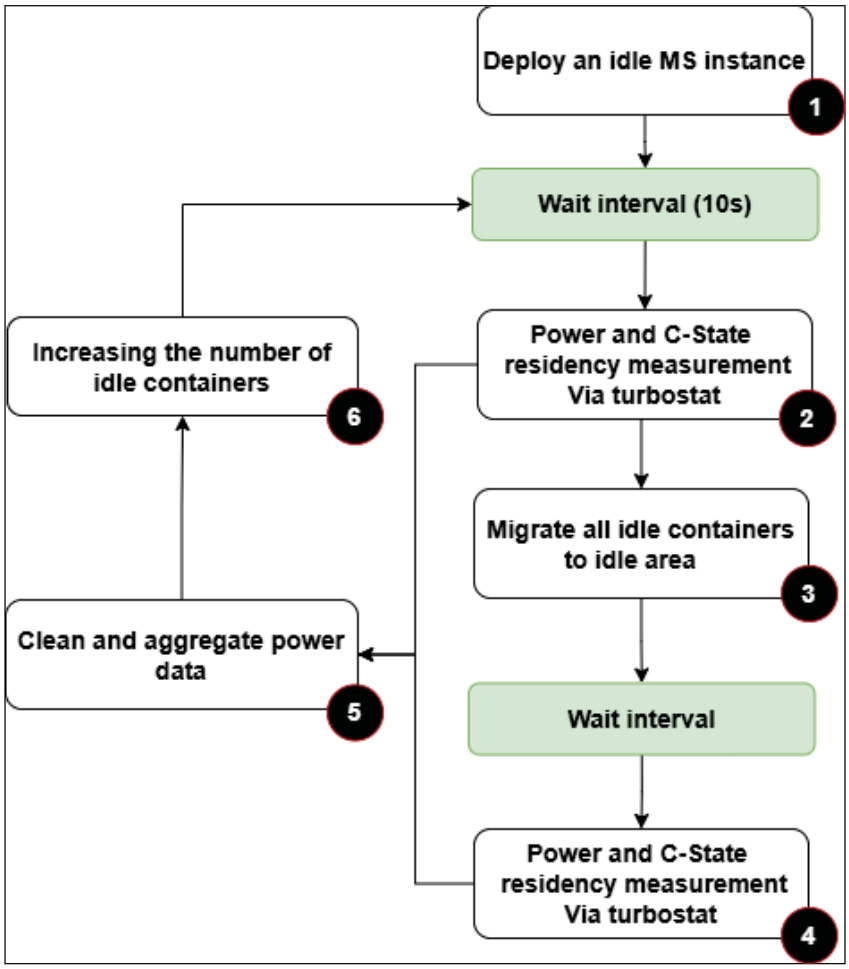


Figure-A II-2 Experimental evaluation workflow

used to ensure detailed monitoring of power usage and C-state behavior.

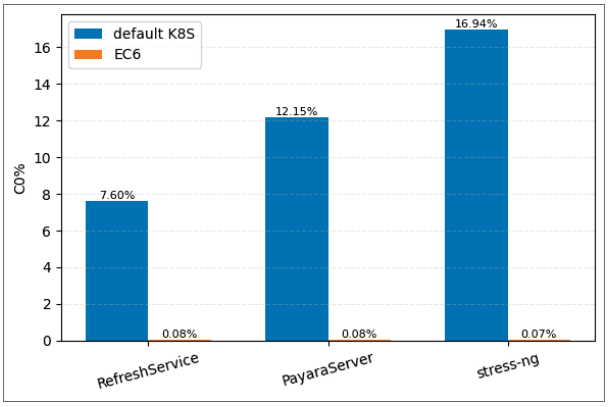
Phase A – K8S Default Policy: After stabilization, turbostat is executed for 400 seconds (Step ❷ in Fig. II-2) to collect:

a. Package-Level Power Consumption (PkgWatts): The average power usage (in watts) of the entire CPU package.

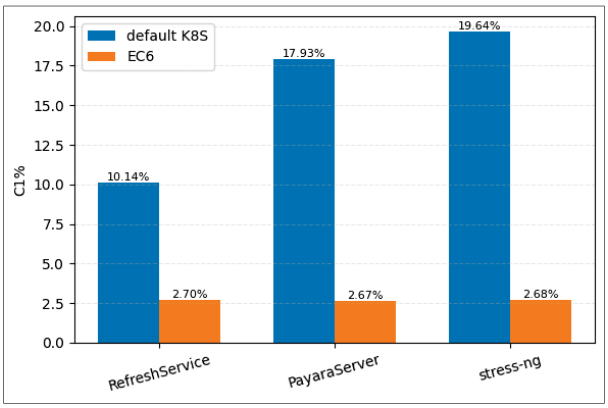
b. Core-specific C-state residency (C0%, C1%, C6%): C-state residency indicates the proportion of time a CPU core remains in a particular power state over the entire measurement period.

It is essential to note that C-state residency represents the percentage of the total measurement period during which a CPU core spends time in a given C-state. For instance, a C0% value of 50% indicates that the core was active for 200 out of 400 seconds. Higher C6% values signify longer durations in deep idle states, indicating greater energy efficiency.

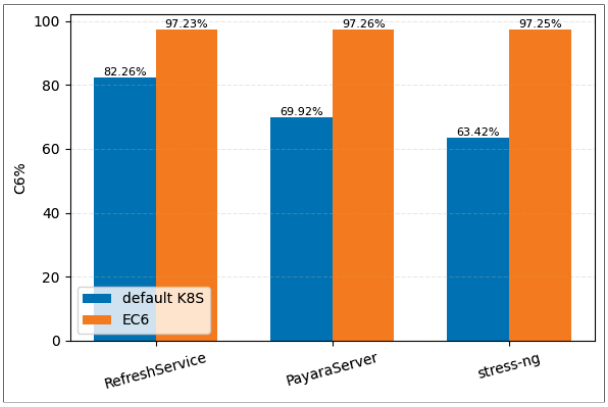
Phase B – EC6 Policy:



a) C0 residency (%)



b) C1 residency (%)



c) C6 residency (%)

Figure 2.1 Core-level C-state residency across idle burner services under default and EC6 scheduling.

The container of the idle service is migrated to the idle area (in this experiment, CPU 27 represents the idle area), replicating EC6's idle migration behavior (Step ❸ in Fig. II-2). A 10-second waiting period is then introduced to allow the system to stabilize. Afterward, turbostat is executed for 400 seconds for the second time to capture power consumption and C-state residency distribution under EC6 (Step ❹ in Fig. II-2).

3. **Step 3: Data Cleaning and Aggregation**
   The collected data is cleaned to remove outliers (Step ❺ in Fig. II-2). Values are averaged over the measurement period (*i.e.,* , 400 seconds). Power metrics are aggregated into a single representative value per phase, and C-state residency data is averaged per core.

4. **Step 4: Workload Scaling**
   The number of concurrently deployed idle service instances is incremented by one (Step ❻ in Fig. II-2), and Steps 1–3 are repeated. This process continues until all 26 active-area cores are fully utilized.

5. Results

A. Impact of Idle Burner Services on C-State Residency distributions

Figs. 2.1a, 2.1b, and 2.1c illustrate the C0, C1, and C6 state residency distributions for three idle burner services, evaluated under both default Kubernetes scheduling and the EC6 strategy. Each service was pinned to a dedicated CPU core, ensuring that C-state transitions were driven solely by the characteristics of the idle service workload. Under default Kubernetes scheduling, idle services remain statically pinned to their allocated cores, leading to heterogeneous C-state residency patterns driven by the services' workload-specific background activity. In contrast, EC6 dynamically migrates idle services, enabling unutilized cores to enter and sustain deep idle states. The following sections provide detailed analyses for RefreshService, PayaraServer, and Stress-NG.

1. RefreshService

As shown in Fig. 2.1a, RefreshService exhibits substantial background activity under default K8s scheduling, despite being in the idle phase (i.e., no external requests). In addition to the mandatory periodic flag refreshing background process, the Spring Boot runtime initiates background operations—including dependency injection scanning, hot-reload monitoring, and garbage collection—that trigger frequent CPU wake-ups. As a result, the assigned core remains in C0 for 7.6%, C1 for 10.1%, and only 82.3% in C6, as indicated in Figs. 2.1b and 2.1c, respectively. Interrupt activity averages 111 IRQs/s, primarily driven by internal refresh routines, as shown in Fig. II-3.

Under EC6, the service container is migrated to a reserved idle core, isolating background-induced interrupts. This suppresses ~90% of IRQs on the original core, enabling over 97% C6 residency.

Despite the absence of application-layer workload, the CPU exhibited a residual C0 residency of 0.08% and 2.7% in C1, primarily driven by unavoidable OS-level tasks such as thermal regulation, background monitoring, and interrupt handling.

These results confirm EC6's ability to suppress non-essential wake-up events, consolidate idle workloads, and enable prolonged deep C-state residency. Consequently, EC6 significantly improves energy efficiency by pushing the system closer to ideal idle-state behavior in containerized cloud environments.

2. PayaraServer

PayaraServer, a full-profile Java EE application server, exhibits persistent background activity due to its thread pools, class loader checks, and heartbeat daemons. Consequently, the core remains in C0 for 12.15%, C1 for 17.93%, and C6 only for 69.9%. Interrupt activity averages 241 IRQs/s. Under EC6, both C0 and C1 residency drop below 3%, while C6 increases to 97.26%. IRQs decrease by nearly an order of magnitude, effectively offloading background activity to the idle area CPU core. These results highlight that idle runtimes can still induce significant background CPU activity, and reinforce EC6's utility in reducing power overhead by consolidating idle services.

3. Stress-NG

The stress-ng workload represents a worst-case idle pattern, executing a 1 ms busy loop every 11 ms. Although this results in a theoretical 9% activity ratio, the core remains in C0 longer than the workload demands. As shown, C0 reaches 16.94%, C1 remains at 19.64%, and C6 drops to 63.42% under default conditions. This occurs because the CPU core does not transition to a sleep state (C1 or C6) immediately after completing instructions. Instead, the built-in C-state controller waits for a predefined target residency time before entering a specific state (as indicated in Table II-1).

Under EC6, the core achieves 97.25% C6 residency, with C0 and C1 reduced to 0.07% and 2.68%, respectively. IRQs also drop to 25/s. These remaining interrupts are unavoidable and stem from thermal monitoring and Read-Copy-Update (RCU) callback handling. These results demonstrate that frequent, short-duration tasks—common in inefficient polling loops—can significantly reduce the time spent in deep sleep. EC6 effectively mitigates this behavior, recovering over 30% of C6 residency and highlighting the value of consolidating such patterns on an isolated core.

B. EC6's Impact on Socket-Level Power Consumption

Figs. II-4a, II-4b, and II-4c show the average socket-level power consumption (PkgWatt) as the number of concurrently deployed idle containers increases from 2 to 26, for RefreshService, PayaraServer, and Stress-NG, respectively. Measurements were averaged over 400-second intervals, with outliers excluded. At low container counts (2 or 4), EC6 provides negligible power reductions (< 1%), as the majority of cores are already idle and capable of entering C6 autonomously. Moreover, as depicted in Fig. II-5, enforcing C6 residency on a single core

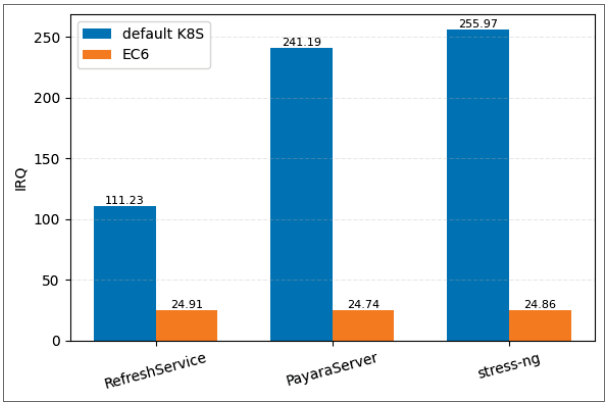


Figure-A II-3 Core-level IRQ/s across idle burner services under default and EC6 scheduling.

yields minimal influence on overall socket power. As container density approaches full socket occupancy (26 cores), EC6 demonstrates significant efficiency gains. By consolidating background workloads onto fewer cores, EC6 frees up idle cores to remain in deep sleep, thereby significantly reducing aggregate power consumption.

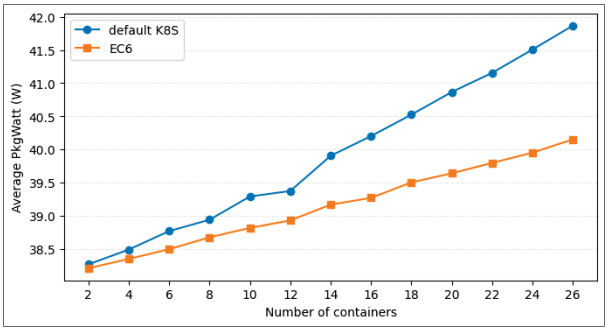
Fig. II-5 summarizes the percentage reduction in PkgWatt under EC6 relative to default K8s, showing that EC6 achieves up to ~4% savings for RefreshService, ~6% for PayaraServer, and a pronounced 13.6% for Stress-NG. These results highlight that EC6's benefits are workload-dependent, with the highest gains observed in workloads with frequent disruptive background activity. This gain is achieved with a negligible latency penalty. More specifically, container restoration to their CPUs when service activity resumes incurs only 3.0146×10^{-2} ms, regardless of container count, rendering the delay negligible in conventional cloud environments. However, even such sub-millisecond latency may be restrictive in real-time or ultra-low-latency applications requiring deterministic response times. Overall, the evaluation confirms that EC6's core consolidation strategy effectively exploits C-state mechanisms to deliver energy savings in containerized environments, particularly for idle burner services.

Discussion

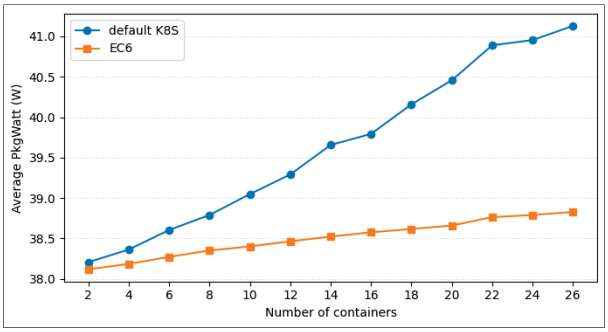
Architectural heterogeneity is a defining characteristic of cloud-native and edge-native infrastructures. This section discusses the broader applicability of the proposed power management approach by analyzing its compatibility with diverse hardware architectures and orchestration frameworks.

A. Hardware Architecture Heterogeneity

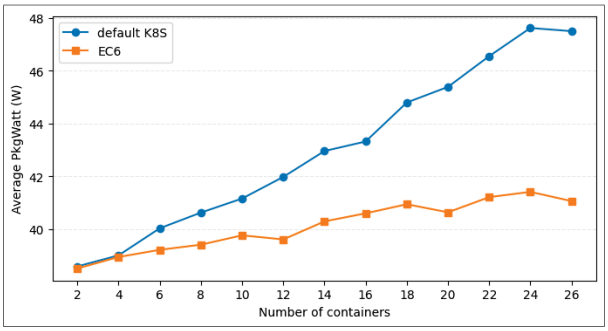
The empirical validation of this work focuses on Intel x86 architectures, which provide robust support for C-states. However, the fundamental mechanism leveraged by our approach—the dynamic manipulation of processor idle states—is not proprietary to Intel. The *Advanced Configuration and Power Interface (ACPI)* standard provides a



(a) RefreshService



(b) PayaraServer



(c) Stress-NG

Figure-A II-4 EC6 power savings (%) for different idle burner services

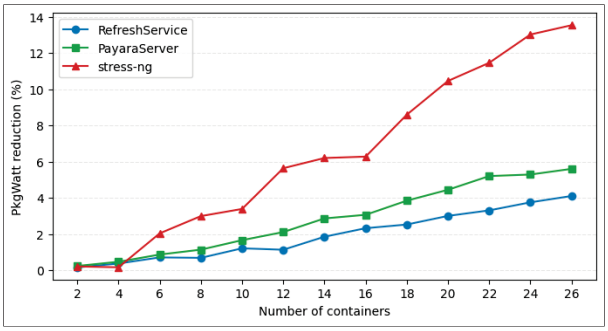


Figure-A II-5 EC6 power savings (%) for different idle burner services

hardware-agnostic abstraction for power management that is widely adopted by other major processor vendors, including AMD and ARM.

- **AMD CPUs** implement an analogous set of core (CC-states) and package C-states that are exposed to the operating system via ACPI, functioning similarly to their Intel counterparts.
- While the underlying implementation on **ARM architectures** relies on a Power State Coordination Interface (PSCI) (ARM, 2014), this firmware layer maps platform-specific power states (e.g., retention, power-down) to the standardized ACPI C-state model presented to the OS.

Consequently, our approach is designed to apply to any server platform that supports ACPI-compliant C-states, allowing the OS to control core idleness. The portability of the core technique depends on this standardized interface, not on a specific vendor's microarchitecture.

B. Orchestration Framework Compatibility

The proposed methodology depends on a key orchestration primitive: the ability to dynamically reallocate CPU cores assigned to a running container without requiring a restart. Our implementation leverages this capability as provided by Alibaba's Koordinator. For environments where Koordinator is not deployed, such as lightweight Kubernetes distributions (e.g., K3s (K3s, 2025), K0s (K0s, 2025)) common in edge computing, alternative mechanisms can achieve the same outcome.

1. **Native Kubernetes CPUManager:** The upstream Kubernetes `CPUManager`, configured with a `static` policy, provides the foundation for exclusive core allocation to pods with a *Guaranteed* QoS class. The `InPlacePodVerticalScaling` feature gate (alpha) extends this functionality by allowing the `kubelet` to modify the `cpuset.cpus` cgroup parameter of a running pod (K8s, 2025b). This combination enables the in-place, dynamic reallocation of CPU resources that our approach requires. While this native solution offers seamless integration with no additional daemon overhead, it is less flexible than Koordinator, requiring integer CPU requests and offering limited policy customization.
2. **Custom Privileged Agent:** For maximum flexibility, a custom-developed daemon, deployed as a privileged `DaemonSet`, can directly manipulate the cgroup filesystem to modify the `cpuset.cpus` assignments for target containers. This approach decouples the reallocation logic from the `kubelet` and its policies, allowing for sophisticated, fine-grained control. However, this method introduces the overhead of developing, deploying, and maintaining a custom component and requires elevated security permissions within the cluster.

7. Conclusion and Future Work

This work presents EC6, an idle service-aware scheduling enhancement for K8s that reduces energy consumption by consolidating idle workloads onto a single core (idle area). This approach allows the remaining cores to sustain deep C-state residency, thereby reducing unnecessary power consumption. Experimental results demonstrated that EC6 improves C6 residency and achieves socket-level power savings of up to 13.6% for aggressive idle patterns and 4–6% for microservices exhibiting typical background activity. These findings highlight the inefficiencies of static pod-to-core binding in K8s and confirm the effectiveness of EC6's idle workload consolidation approach.

Despite its effectiveness, EC6 currently reserves one core per socket for the idle area to host migrated idle services, thereby reducing the CPU pool available for scheduling active services. To address this, we plan to make the idle area more flexible: the reserved core may be temporarily borrowed by the active area when needed—e.g., when the number of available cores is insufficient to schedule a pod—and returned to the idle pool once load decreases. This dynamic sharing mechanism will help preserve EC6's energy-saving benefits without compromising resource availability for active workloads. Other future directions include integrating tickless kernel support and RCU callback forwarding to minimize operating system-induced interruptions on idle cores. Moreover, to support latency-sensitive services, we plan to investigate proactive pod restoration strategies, enabling pods to be returned to their allocated cores ahead of service activity resumption, thereby eliminating wake-up penalties.

LIST OF REFERENCES

Memory and computing power - AWS Lambda. Accessed: 2024-03-12.

Apache OpenWhisk. Accessed: 2024-03-12.

Linux kernel scaling governors. https://t.ly/smCgA, last accessed on 01/08/2023.

Node.js Performance Profiling. Accessed: 2025-04-15.

Energy efficiency predictions for data centres in 2023. https://datacentremagazine.com/articles/efficiency-to-loom-large-for-data-centre-industry-in-2023, last accessed on 01/08/2023.

CPU Frequency Trace Low. https://rb.gy/fsx4qv, last accessed on 01/11/2023.

CPU Frequency Trace high. https://rb.gy/ya8myt, last accessed on 01/11/2023.

Google Cloud Functions Pricing - Compute Time. Accessed: 2024-03-12.

Gunicorn's worker types and behavior. Accessed: 2025-03-15.

JVM Garbage Collection. Accessed: 2025-03-10.

NVIDIA Jetson. https://developer.nvidia.com/EMBEDDED/Jetson-modules, last accessed on 01/08/2023.

ODROID XU3. https://www.hardkernel.com/shop/odroid-xu3/, last accessed on 01/08/2023.

UInside the Python GIL. Accessed: 2025-03-15.

Running Average Power Limit Energy Reporting. Accessed: 2025-03-04.

Python's asyncio Event Loop Mechanism. Accessed: 2025-03-25.

Intel Idle Driver. Accessed: 2024-05-12.

K3s. https://k3s.io/, last accessed on 01/08/2023.

kubeedge. https://kubeedge.io/, last accessed on 01/08/2023.

microk8s. https://microk8s.io/, last accessed on 01/08/2023.

Building a large-scale feature flag system. Accessed: 2025-05-01.

(2014). *Power State Coordination Interface (PSCI) System Architecture*. Consulted at https://documentation-service.arm.com/static/5f905b78f86e16515cdc1fca?token=.

(2021a). OpenFaaS Documentation. Accessed: 2024-03-12.

(2021b). OpenFaaS Architecture: Stack. Accessed: 2024-03-12.

(2021). Nuclio. Consulted at https://nuclio.io./.

[Accessed 14 Jul 2025]. (2022). Advanced Configuration and Power Interface (ACPI) Specification 6.4. Consulted at https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/08_Processor_Configuration_and_Control/processor-power-states.html.

(2023). Runtime options with Memory, CPUs.

(2025). *AMD EPYC™ 9005 BIOS & Workload Tuning Guide*. Consulted at https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58467_amd-epyc-9005-tg-bios-and-workload.pdf.

Intel Corporation. (2025). Intel® 64 and IA-32 Architectures Software Developer's Manual. Version 088, accessed 14 Jul 2025, Consulted at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[Accessed 14 Jul 2025]. (2025). k0s — The Zero Friction Kubernetes. Consulted at https://k0sproject.io/.

[Accessed 14 Jul 2025]. (2025). K3s — Lightweight Kubernetes. Consulted at https://docs.k3s.io/.

[Accessed 14 Jul 2025]. (2025a). Control CPU Management Policies on the Node. Consulted at https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/.

[Accessed 14 Jul 2025]. (2025b). Feature Gates — `InPlacePodVerticalScaling`. Consulted at https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/.

(2025). Koordinator: QoS-Based Scheduling System for Kubernetes. Commit history accessed 14 Jul 2025.

[Accessed 14 Jul 2025]. (2025). CPUSETS — The Linux Kernel Documentation. Consulted at https://docs.kernel.org/admin-guide/cgroup-v1/cpusets.html.

Abbas, N., Zhang, Y., Taherkordi, A. & Skeie, T. (2017). Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1), 450–465.

Abdullah, M., Iqbal, W., Mahmood, A., Bukhari, F. & Erradi, A. (2020). Predictive autoscaling of microservices hosted in fog microdata center. *IEEE Systems Journal*, 15(1), 1275–1286.

Accounting, A. C. (2004). The Greenhouse Gas Protocol. *World Resources Institute and World Business Council for Sustainable Development: Washington, DC, USA*.

Ahuja, K. (2024). Legal Framework for Carbon Credits and Emission Trading. Consulted at https://tinyurl.com/2jef8hku.

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P. & Hilt, V. (2018). {SAND}: Towards {High-Performance} Serverless Computing. *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pp. 923–935.

Alam, M., Rufino, J., Ferreira, J., Ahmed, S. H., Shah, N. & Chen, Y. (2018). Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9), 118–123.

Alibaba Cloud Documentation. (2024). ACK Koordinator (FKA ACK SLO Manager). Accessed: 2025-05-20.

Aljulayfi, A. F. & Djemame, K. (2021). A Machine Learning based Context-aware Prediction Framework for Edge Computing Environments. *CLOSER*, 2021, 143–150.

Alzahrani, E. J., Tari, Z., Zeephongsekul, P., Lee, Y. C., Alsadie, D. & Zomaya, A. Y. (2016). SLA-Aware Resource Scaling for Energy Efficiency. *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 852-859. doi: 10.1109/HPCC-SmartCity-DSS.2016.0123.

Anderson, T., Belay, A., Chowdhury, M., Cidon, A. & Zhang, I. (2023). Treehouse: A case for carbon-aware datacenter software. *ACM SIGENERGY Energy Informatics Review*, 3(3), 64–70.

Andrae, A. (2017). Total consumer power consumption forecast. *Nordic Digital Business Summit*, 10, 69.

Andrae, A. S. & Edler, T. (2015). On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1), 117–157.

Andringa, L. (2024). *Estimating Energy Consumption of Cloud-Native Applications*. (Master's Thesis, University of Groningen). Consulted at https://fse.studenttheses.ub.rug.nl/33583/1/Estimating-energy-consumption-of-Cloud-Native-applications.pdf.

Anthony, L. F. W., Kanding, B. & Selvan, R. (2020). Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv preprint arXiv:2007.03051*.

Antoniou, G., Bartolini, D., Volos, H., Kleanthous, M., Wang, Z., Kalaitzidis, K., Rollet, T., Li, Z., Mutlu, O., Sazeides, Y. et al. (2024). Agile C-states: a core C-state architecture for latency critical applications optimizing both transition and cold-start latency. *ACM Transactions on Architecture and Code Optimization*, 21(4), 1–26.

ArchWiki contributors. [ArchWiki]. (2026). CPU frequency scaling. Consulted at https://wiki.archlinux.org/title/CPU_frequency_scaling.

Ariel, C. AI Is Pushing The World Toward An Energy Crisis. Accessed: 23-05-2024.

Arroba, P., Moya, J. M., Ayala, J. L. & Buyya, R. (2015). DVFS-aware consolidation for energy-efficient clouds. *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 494–495.

Arroba, P., Moya, J. M., Ayala, J. L. & Buyya, R. (2017). Dynamic Voltage and Frequency Scaling-aware dynamic consolidation of virtual machines for energy efficient cloud data centers. *Concurrency and Computation: Practice and Experience*, 29(10), e4067.

Asaad, M., Ahmad, F., Alam, M. S. & Rafat, Y. (2018). IoT enabled monitoring of an optimized electric vehicle's battery system. *Mobile Networks and Applications*, 23(4), 994–1005.

Aslanpour, M. S., Toosi, A. N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., Assuncao, M., Gill, S. S., Gaire, R. & Dustdar, S. (2021). Serverless edge computing: vision and challenges. *Proceedings of the 2021 Australasian computer science week multiconference*, pp. 1–10.

Aslanpour, M. S., Toosi, A. N., Cheema, M. A. & Gaire, R. (2023). DVFaaS: Leveraging DVFS for FaaS Workflows. *Proceedings of the 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*.

Aslanpour, M. S., Toosi, A. N., Cheema, M. A. & Gaire, R. (2022). Energy-aware resource scheduling for serverless edge computing. *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 190–199.

Asyabi, E., Bestavros, A., Sharafzadeh, E. & Zhu, T. (2020). Peafowl: In-application cpu scheduling to reduce power consumption of in-memory key-value stores. *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 150–164.

Avalkar, R. (2022). AWS Lambda: Reserved Concurrency vs. Provisioned Concurrency Scaling. Accessed: 2024-03-12.

Avasalcai, C., Tsigkanos, C. & Dustdar, S. (2021). Resource management for latency-sensitive IoT applications with satisfiability. *IEEE Transactions on Services Computing*.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. et al. (2017). Serverless computing: Current trends and open problems. In *Research advances in cloud computing* (pp. 1–20). Springer.

Barroso, L. A. & Hölzle, U. (2007). The case for energy-proportional computing. *Computer*, 40(12), 33–37.

Barroso, L. A., Clidaras, J. & Hölzle, U. (2013). The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3), 1–154.

Bedard, D., Lim, M. Y., Fowler, R. & Porterfield, A. (2010). Powermon: Fine-grained and integrated power monitoring for commodity computer systems. *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pp. 479–484.

Belgaid, M. C. (2022). *Green coding: an empirical approach to harness the energy consumption ofsoftware services*. (Ph.D. thesis, Université de Lille).

Bellal, Z., Lahlou, L., Kara, N. & Khayat, I. E. (2024a). GAS: DVFS-Driven Energy Efficiency Approach For Latency-Guaranteed Edge Computing Microservices. *IEEE Transactions on Green Communications and Networking*, 1-1. doi: 10.1109/TGCN.2024.3420957.

Bellal, Z., Lahlou, L., Kara, N. & Khayat, I. E. (2024b). GAS: DVFS-driven energy efficiency approach for latency-guaranteed edge computing microservices. *IEEE Transactions on Green Communications and Networking*, 1–1. doi: 10.1109/TGCN.2024.3420957.

Bellal, Z., Lahlou, L., Kara, N., Murphy, T. & Nguyen, T. P. (2025a). EC6: Enhancing Energy Efficiency in Kubernetes Through Dynamic Extension of CPU Deep Idle States (C6). *2025 16th International Conference on Network of the Future (NoF)*, pp. 200-208. doi: 10.1109/NoF66640.2025.11223306.

Bellal, Z., Lahlou, L., Kara, N., Murphy, T. & Nguyen, T. P. (2025b). 3GC: A Deadline-Aware and Energy-Efficient Resource Allocation Scheme for Serverless Edge Computing. *2025 IEEE 22nd Consumer Communications & Networking Conference (CCNC)*, pp. 1–9.

Bellal, Z., Lahlou, L., Kara, N., Murphy, T. & Nguyen, T. P. (2025c). 3GC: A Deadline-Aware and Energy-Efficient Resource Allocation Scheme for Serverless Edge Computing. *Proceedings of the 2025 IEEE CCNC Conference*.

Berg, F. v. d., Postema, B. F. & Haverkort, B. R. (2016). Evaluating load balancing policies for performance and energy-efficiency. *arXiv preprint arXiv:1610.08172*.

Beyer, D. & Wendler, P. CPU Energy Meter: A Tool for Energy-Aware Algorithms Engineering. 12079, 126.

Beyer, D., Löwe, S. & Wendler, P. (2019). Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21, 1–29.

Bhasi, V. M., Gunasekaran, J. R., Thinakaran, P., Mishra, C. S., Kandemir, M. T. & Das, C. (2021). Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. *Proceedings of the ACM Symposium on Cloud Computing*, pp. 153–167.

Biswas, D., Ivanov, D., Ayala, J. L., Benini, L., Macii, E., Jantsch, A. & Ros, E. (2017). Machine learning for run-time energy optimisation in many-core systems. *2017 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1588–1592. doi: 10.23919/DATE.2017.7927243.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E. & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 50–57. doi: 10.1145/2890784.

Buyya, R., Vecchiola, C. & Selvi, S. T. (2013). *Mastering cloud computing: foundations and applications programming*. Newnes.

Cadorel, E. & Saingre, D. (2024). A protocol to assess the accuracy of process-level power models. *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 74–84.

Calheiros, R. N. & Buyya, R. (2014). Energy-efficient scheduling of urgent bag-of-tasks applications in clouds through DVFS. *2014 IEEE 6th international conference on cloud computing technology and science*, pp. 342–349.

Cañete, A., Djemame, K., Amor, M., Fuentes, L. & Aljulayfi, A. (2022). A proactive energy-aware auto-scaling solution for edge-based infrastructures. *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pp. 240–247.

Cao, K., Liu, Y., Meng, G. & Sun, Q. (2020). An overview on edge computing research. *IEEE access*, 8, 85714–85728.

Casalicchio, E. (2019). Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, 221–235.

Castillo, E., Moreto, M., Casas, M., Alvarez, L., Vallejo, E., Chronaki, K., Badia, R., Bosque, J. L., Beivide, R., Ayguade, E. et al. (2016). CATA: criticality aware task acceleration for multicore processors. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 413–422.

Castro, P., Isahagian, V., Muthusamy, V. & Slominski, A. (2022). Hybrid Serverless Computing: Opportunities and Challenges. *arXiv preprint arXiv:2208.04213*.

Chang, M.-F. & Liang, W.-Y. (2011). Learning-directed dynamic voltage and frequency scaling for computation time prediction. *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1023–1029.

Chen, J., Manivannan, M., Abduljabbar, M. & Pericas, M. (2022). ERASE: Energy efficient task mapping and resource management for work stealing runtimes. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(2), 1–29.

Chen, Y.-L., Chang, M.-F., Yu, C.-W., Chen, X.-Z. & Liang, W.-Y. (2018). Learning-directed dynamic voltage and frequency scaling scheme with adjustable performance for single-core and multi-core embedded and mobile systems. *Sensors*, 18(9), 3068.

Chen, Y., Agarwal, R. & Stoica, I. (2019). PARTIES: QoS-aware resource partitioning for multiple interactive services. *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 107–120. doi: 10.1145/3297858.3304023.

Cherkasova, L. (1998). *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA.

Choochotkaew, S., Amaral, M. & Chen, H. (2024a). Idle Power Matters: Kepler Metrics for Public Cloud Energy Efficiency. Accessed: Feb. 16, 2025, Consulted at https://tag-env-sustainability.cncf.io/blog/2024-06-idle-power-matters-kepler-metrics-for-public-cloud-energy-efficiency/.

Choochotkaew, S., Wang, C., Chen, H., Chiba, T., Amaral, M., Lee, E. K. & Eilam, T. (2024b). A Robust Power Model Training Framework for Cloud Native Runtime Energy Metric Exporter. *arXiv preprint arXiv:2407.00878*.

Chou, C.-H., Wong, D. & Bhuyan, L. N. (2016). Dynsleep: Fine-grained power management for a latency-critical data center application. *Proceedings of the 2016 international symposium on low power electronics and design*, pp. 212–217.

Chou, C.-H., Bhuyan, L. N. & Wong, D. (2019). μdpm: Dynamic power management for the microsecond era. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 120–132.

Cicconetti, C., Conti, M. & Passarella, A. (2021). FaaS Execution Models for Edge Applications. *arXiv preprint arXiv:2111.06595*.

Cisco, U. (2020). Cisco annual internet report (2018–2023) white paper. *Cisco: San Jose, CA, USA*.

(CNCF), C. N. C. F. (2023). Exploring Kepler's Potentials – Unveiling Cloud Application Power Consumption.

Colmant, M., Kurpicz, M., Felber, P., Huertas, L., Rouvoy, R. & Sobe, A. (2015). Process-level power estimation in vm-based systems. *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–14.

Colmant, M., Felber, P., Rouvoy, R. & Seinturier, L. (2017). Wattskit: Software-defined power monitoring of distributed systems. *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 514–523.

Community, I. (2024a). Counting Cache Miss. Accessed: 2025-03-04, Consulted at https://community.intel.com/t5/Software-Tuning-Performance/Counting-Cache-miss/td-p/1096236.

Community, I. (2024b). Reading Performance Counters with rdpmc. Accessed: 2025-03-04, Consulted at https://tinyurl.com/yjw5vth2.

computing, S. CLEVER: Container Level Energy-efficient VPA Recommender. Accessed: 30-05-2024.

computing, S. PEAKS: Power Efficiency Aware Kubernetes Scheduler. Accessed: 30-05-2024.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Corporation, I. (2022). Kubernetes Power Manager.

Dabbagh, M., Hamdaoui, B., Guizani, M. & Rayes, A. (2014). Release-time aware VM placement. *2014 IEEE Globecom Workshops (GC Wkshps)*, pp. 122–126.

Dabbagh, M., Hamdaoui, B., Guizani, M. & Rayes, A. (2015). Energy-efficient resource allocation and provisioning framework for cloud data centers. *IEEE Transactions on Network and Service Management*, 12(3), 377–391.

Das, A., Leaf, A., Varela, C. A. & Patterson, S. (2020). Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. *2020 IEEE 13th international conference on cloud computing (CLOUD)*, pp. 609–618.

De, P., Dunne, E. J., Ghosh, J. B. & Wells, C. E. (1997). Complexity of the discrete time-cost tradeoff problem for project networks. *Operations research*, 45(2), 302–306.

de Kruijff, P. (2025). DeepSeek efficiencies make AI cheaper. *ABC News*. Consulted at https://tinyurl.com/4v7c8d58.

Delimitrou, C. & Kozyrakis, C. (2013). Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS '13), 77–88. doi: 10.1145/2499368.2451125.

Deng, Q., Meisner, D., Bhattacharjee, A., Wenisch, T. F. & Bianchini, R. (2012). Coscale: Coordinating cpu and memory system dvfs in server systems. *2012 45th annual IEEE/ACM international symposium on microarchitecture*, pp. 143–154.

Deng, S., Zhao, H., Xiang, Z., Zhang, C., Jiang, R., Li, Y., Yin, J., Dustdar, S. & Zomaya, A. (2021). Dependent Function Embedding for Distributed Serverless Edge Computing. *IEEE Transactions on Parallel and Distributed Systems*.

Dhiman, G., Pusukuri, K. K. & Rosing, T. (2008). Analysis of dynamic voltage scaling for system level energy management. *USENIX HotPower*, 8.

Doweck, J., Kao, W.-F., Lu, A. K.-y., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A. & Yoaz, A. (2017). Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2), 52–62.

ebpf. What is eBPF. Accessed: 30-05-2024.

Eimer, T., Lindauer, M. & Raileanu, R. (2023). Hyperparameters in Reinforcement Learning and How To Tune Them. *Proceedings of the 40th International Conference on Machine Learning*, 202(Proceedings of Machine Learning Research), 9104–9149. Consulted at https://proceedings.mlr.press/v202/eimer23a.html.

Eismann, S., Grohmann, J., Van Eyk, E., Herbst, N. & Kounev, S. (2020). Predicting the costs of serverless workflows. *Proceedings of the ACM/SPEC international conference on performance engineering*, pp. 265–276.

El Khazen, M. W., Amor, S. B., Kougblenou, K., Gogonel, A. & Cucu-Grosjean, L. (2023). Work in progress: Towards a statistical worst-case energy consumption model. *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 333–336.

Elgamal, T., Sandur, A., Nahrstedt, K. & Agha, G. (2018). Costless: Optimizing cost of serverless computing through function fusion and placement. *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 300–312.

Ermolenko, D., Kilicheva, C., Muthanna, A. & Khakimov, A. (2021). Internet of Things services orchestration framework based on Kubernetes and edge computing. *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, pp. 12–17.

Etinski, M., Corbalán, J., Labarta, J. & Valero, M. (2012). Understanding the future of energy-performance trade-off via DVFS in HPC environments. *Journal of Parallel and Distributed Computing*, 72(4), 579–590.

Eusebio, L. M. (2022). Green Daemon: Green Technology Data Acquiescence Tool. *Proceedings of the 5th International Conference on Information Science and Systems*, pp. 156–161.

Evans, B. J. & Gough, J. (2024). *Optimizing Cloud Native Java: Practical Techniques for Improving JVM Application Performance*. " O'Reilly Media, Inc.".

Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sørensen, R. B., Wägemann, P. & Wegener, S. (2016). TACLeBench: A benchmark collection to support worst-case execution time research. *16th International Workshop on Worst-Case Execution Time Analysis*.

Falkevych, V. & Lisniak, A. (2025). Cache invalidation based on a declarative approach for separating business logic of microservices from cache update rules. *Eastern-European Journal of Enterprise Technologies*, 2(2 (134)), 68–74. doi: 10.15587/1729-4061.2025.325932.

FastestGrowing. (2020). The Fastest-Growing Cloud. Consulted at https://www.cbinsights.com/research/serverless-cloud-computing/.

fastly. (2020). fastly edge cloud platform. Consulted at https://www.fastly.com/.

Fernández-Cerero, D., Fernández-Montes, A., Ortega, F. J., Jakóbik, A. & Widlak, A. (2020). Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption. *Simulation Modelling Practice and Theory*, 101, 101966.

Fieni, G., Rouvoy, R. & Seinturier, L. (2020). SmartWatts: Self-calibrating software-defined power meter for containers. *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 479–488.

Fieni, G., Rouvoy, R. & Seiturier, L. (2021). Selfwatts: On-the-fly selection of performance events to optimize software-defined power meters. *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 324–333.

Florence, A. P., Shanthi, V. & Simon, C. (2016). Energy conservation using dynamic voltage frequency scaling for computational cloud. *The Scientific World Journal*, 2016.

Freina, D., Jansen, M. & Trivedi, A. A Survey of Energy Measurement Methodologies for Computer Systems.

Fuerst, A. & Sharma, P. (2021). FaasCache: keeping serverless computing alive with greedy-dual caching. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 386–400.

Fujimoto, K., Harasawa, H., Natori, K., Otani, I., Saito, S. & Shiraga, A. (2022). PWU: Pre-Wakeup for CPU Idle to Reduce Latency and Power Consumption. *2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–6.

Gackstatter, P., Frangoudis, P. A. & Dustdar, S. Pushing Serverless to the Edge with WebAssembly Runtimes.

Gadepalli, P. K., Peach, G., Cherkasova, L., Aitken, R. & Parmer, G. (2019a). Challenges and opportunities for efficient serverless computing at the edge. *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pp. 261–2615.

Gadepalli, P. K., Peach, G., Parmer, G., Espy, J. & Day, Z. (2019b). Chaos: a system for criticality-aware, multi-core coordination. *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 77–89.

Gadepalli, P. K., McBride, S., Peach, G., Cherkasova, L. & Parmer, G. (2020). Sledge: a serverless-first, light-weight wasm runtime for the edge. *Proceedings of the 21st International Middleware Conference*, pp. 265–279.

Garcia, C. (2020). The Real Amount of Energy A Data Center Uses. Consulted at https://www.akcp.com/articles/the-real-amount-of-energy-a-data-center-use.

Ge, Y., Zhang, Y., Malani, P., Wu, Q. & Qiu, Q. (2012). Low power task scheduling and mapping for applications with conditional branches on heterogeneous multi-processor system. *Journal of Low Power Electronics*, 8(5), 535–551.

Gebrehiwot, M. E., Aalto, S. & Lassila, P. (2017). Energy efficient load balancing in web server clusters. *2017 29th International Teletraffic Congress (ITC 29)*, 3, 13–18.

Gholkar, N., Mueller, F. & Rountree, B. (2019a). Uncore power scavenger: A runtime for uncore power conservation on hpc systems. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–23.

Gholkar, N., Mueller, F. & Rountree, B. (2019b). Uncore power scavenger. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2019)*, pp. 1–23. doi: 10.1145/3295500.3356150.

Giridhar, B., Cieslak, M., Duggal, D., Dreslinski, R., Chen, H. M., Patti, R., Hold, B., Chakrabarti, C., Mudge, T. & Blaauw, D. (2013). Exploring DRAM organizations for energy-efficient and resilient exascale memories. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12.

Gocht, A., Schöne, R. & Bielert, M. (2019). Q-learning inspired self-tuning for energy efficiency in HPC. *2019 IEEE 17th International Conference on High Performance Computing, Data and Analytics (HiPC) or HPCS*, pp. 344–347. doi: 10.1109/HPCS48598.2019.9188112.

Goel, B. & McKee, S. A. (2016). A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 273–282. doi: 10.1109/IPDPS.2016.118.

Gomez-Selles, J. (2024). Power Monitoring for Red Hat OpenShift: Technology Preview. *Red Hat Blog*. Consulted at https://www.redhat.com/en/blog/power-monitoring-red-hat-openshift-technology-preview. Technology Preview announcement.

Gouicem, R., Carver, D., Lozi, J.-P., Sopena, J., Lepers, B., Zwaenepoel, W., Palix, N., Lawall, J. & Muller, G. (2020). Fewer Cores, More Hertz: Leveraging {High-Frequency} Cores in the {OS} Scheduler for Improved Application Performance. *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 435–448.

Govindaraj, V., George, S., Kandemir, M., Sampson, J. & Naryanan, V. (2021). PowerPrep: A power management proposal for user-facing datacenter workloads. *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–7.

Greenberg, S., Mills, E., Tschudi, B., Rumsey, P. & Myatt, B. (2006). Best practices for data centers: Lessons learned from benchmarking 22 data centers. *Proceedings of the ACEEE Summer Study on Energy Efficiency in Buildings in Asilomar, CA. ACEEE, August*, 3, 76–87.

Guermouche, A., Étienne André, Trahay, F. & Dulong, R. (2022). Combining power capping and uncore frequency scaling for energy efficiency. No DOI available; workshop paper.

Gunasekaran, J. R., Mishra, C. S., Thinakaran, P., Kandemir, M. T. & Das, C. R. (2020). Implications of public cloud resource heterogeneity for inference serving. *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pp. 7–12.

Gupta, H., Vahid Dastjerdi, A., Ghosh, S. K. & Buyya, R. (2017). iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience*, 47(9), 1275–1296.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200.

Hagras, T. & El-Sayed, G. A. (2024). Maintaining the completion-time mechanism for Greening tasks scheduling on DVFS-enabled computing platforms. *Cluster Computing*, 1–16.

Haj-Yahya, J., Volos, H., Bartolini, D. B., Antoniou, G., Kim, J. S., Wang, Z., Kalaitzidis, K., Rollet, T., Chen, Z., Geng, Y. et al. (2022). AgileWatts: An Energy-Efficient CPU Core Idle-State Architecture for Latency-Sensitive Server Applications. *arXiv preprint arXiv:2203.02550*. Consulted at https://arxiv.org/abs/2203.02550.

Haja, D., Turanyi, Z. R. & Toka, L. (2020). Location, proximity, affinity–the key factors in FaaS. *Infocommunications Journal*, 12(4), 14–21.

Hall, A. & Ramachandran, U. (2019). An execution model for serverless functions at the edge. *Proceedings of the International Conference on Internet of Things Design and Implementation*, pp. 225–236.

Haque, M. E., He, Y., Elnikety, S., Nguyen, T. D., Bianchini, R. & McKinley, K. S. (2017). Exploiting Heterogeneity for Tail Latency and Energy Efficiency. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. doi: 10.1109/MICRO.2017.49.

Hat, R. (2022). Save power with per-pod power management for latency-sensitive applications.

Hat, R. (2024). Power Monitoring on OpenShift – Technology Preview.

Hè, H., Friedman, M. & Rekatsinas, T. (2024). Energat: Fine-grained energy attribution for multi-tenancy. *ACM SIGENERGY Energy Informatics Review*, 4(3), 18–25.

Hellerstein, J. L., Diao, Y., Parekh, S. & Tilbury, D. M. (2004). *Feedback Control of Computing Systems*. John Wiley & Sons.

Herbert, S. & Marculescu, D. (2007). Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. *Proceedings of the 2007 international symposium on Low power electronics and design*, pp. 38–43.

Herzog, B., Hönig, T., Schröder-Preikschat, W., Plauth, M., Köhler, S. & Polze, A. (2019). Bridging the gap: Energy-efficient execution of software workloads on heterogeneous hardware components. *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, pp. 428–430.

Herzog, B., Hügel, F., Reif, S., Hönig, T. & Schröder-Preikschat, W. (2021). Automated selection of energy-efficient operating system configurations. *Proceedings Of The Twelfth ACM International Conference On Future Energy Systems*, pp. 309–315.

Herzog, B., Reif, S., Hügel, F., Schröder-Preikschat, W. & Hönig, T. (2022). Bears: Building Energy-Aware Reconfigurable Systems. *2022 XII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 1–8.

Hintemann, R. & Hinterholzer, S. (2019). Energy consumption of data centers worldwide. *Business, Computer Science (ICT4S)*.

Hoffmann, H., Sidiroglou, S., Carbin, M. & Amarasinghe, S. (2008). Analysis of Dynamic Voltage Scaling for System Level Energy Management. *Proceedings of the 1st Workshop on Power-Aware Computing and Systems (HotPower)*.

Hsu, C.-H. & Kremer, U. (2003). The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 38–48.

Hubblo. Scaphandre. Accessed: 01-02-2025.

Huber, F., Körber, N. & Mock, M. (2019). Selena: A serverless energy management system. *Proceedings of the 5th International Workshop on Serverless Computing*, pp. 7–12.

i scoop. The Impact of Growing IT Sector Electricity Demand. Accessed: 03-07-2024.

IBM. What is containerization? Accessed: 2025-05-19, Consulted at https://www.ibm.com/think/topics/containerization.

IDC. (2021). Internet of Things Statistics, Facts Trends for 2022. Consulted at https://www.crn.com/slide-shows/networking/gartner-s-magic-quadrant-for-wan-edge-infrastructure-2021-hpe-joins-leader-circle-thanks-to-silver-peak-buy.

(IEA), I. E. A. Data Centres and Data Transmission Networks. Accessed: 03-07-2024.

Intel. Package C-States - 005 - ID:655258 | Core™ Processors. Accessed: 2025-05-19, Consulted at https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/005/package-c-states/.

Intel Corporation. (2017). Intel® 64 and IA-32 Architectures. Available at: https://shorturl.at/xSYvx.

Intel Corporation. (2019a). Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3 (System Programming Guide). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

Intel Corporation. (2019b). *CPU Management - CPU Pinning and Isolation in K8S*. Consulted at https://tinyurl.com/yzyjnptx.

Intel Corporation. (2020a). C-State. Accessed: 30-05-2024.

Intel Corporation. (2020b). RDT (Intel® Resource Director Technology) — CRI Resource Manager. Version 0.9.2, Consulted at https://intel.github.io/cri-resource-manager/stable/docs/policy/rdt.html.

Intel Corporation. (2022). Intel® VTune™ Profiler User Guide: CPU Metrics Reference. Version 2023.0, accessed March 25, 2026.

Intel Corporation. (2024). Intel PCM.

IO, S. C. (2024a). Kepler GitHub Project Board – Multi-Architecture Backlog ("all HW x86, arm, s390x").

IO, S. C. (2024b). GitHub Issue #1066 – CPU Architecture Reported as Unknown for AMD CPUs.

IO, S. C. (2024c). GitHub Issue #1556 – ARM (AWS Graviton) Crash on Startup.

Ismail, L. & Materwala, H. (2020). Computing server power modeling in a data center: Survey, taxonomy, and performance evaluation. *ACM Computing Surveys (CSUR)*, 53(3), 1–34.

Javadpour, A., Sangaiah, A. K., Pinto, P., Ja'fari, F., Zhang, W., Abadi, A. M. H. & Ahmadi, H. (2023). An energy-optimized embedded load balancing using DVFS computing in cloud data centers. *Computer Communications*, 197, 255–266.

Javed, H., Toosi, A. N. & Aslanpour, M. S. (2021). Serverless Platforms on the Edge: A Performance Analysis. *arXiv preprint arXiv:2111.06563*.

Jay, M., Ostapenco, V., Lefèvre, L., Trystram, D., Orgerie, A.-C. & Fichel, B. (2023). An experimental comparison of software-based power meters: focus on CPU and GPU. *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 106–118.

Jeong, J., Kim, J., Bae, D. & Lee, D. (2020). Gemini: Learning-based per-query DVFS for latency-critical workloads. No DOI available; conference paper.

Jia, X. & Zhao, L. (2021). RAEF: Energy-efficient resource allocation through energy fungibility in serverless. *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 434–441.

Jiang, P. H. (2023). Towards Optimized Microservices Performance & Sustainability via Istio, Kepler and Smart Scheduling. *CNCF TAG Environmental Sustainability Blog*. Consulted at https://tag-env-sustainability.cncf.io/blog/2023-sustainability-istio-kepler-smart-scheduling. Guest post as part of Cloud Native Sustainability Week.

Jin, R., Yang, Q. & Zhao, M. (2020). Is faas suitable for edge computing. *USENIX Association, June*.

Jonggyu Park,Theano Stavrinos, S. P. & Anderson., T. (2024). EMPower: The Case for a Cloud Power Control Plane. *Proceedings of 3rd Workshop on Sustainable Computer Systems (HotCarbon'24)*, (HotNets '24). doi: 10.1145/3696348.3696896.

Juan, J. L. & Marculescu, R. (2012). Power-Aware Performance Increase via Core/Uncore Reinforcement Control for Chip Multiprocessors. *Proceedings of the 17th ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 99–104. doi: 10.1145/2333660.2333717.

Kaffes, K., Sbirlea, D., Lin, Y., Lo, D. & Kozyrakis, C. (2020). Leveraging application classes to save power in highly-utilized data centers. *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 134–149.

Kalia, A., Lazarev, N., Xue, L., Foukas, X., Radunovic, B. & Yan, F. Y. (2025). Towards Energy-Efficient 5G vRAN Servers. *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Kanev, S., Hazelwood, K., Wei, G.-Y. & Brooks, D. (2014). Tradeoffs between power management and tail latency in warehouse-scale applications. *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 31–40.

Kannan, R. S., Subramanian, L., Raju, A., Ahn, J., Mars, J. & Tang, L. (2019). Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16.

Kannan, S. & Kremer, U. (2023). Towards Application Centric Carbon Emission Management. *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pp. 1–7.

Kanso, A. & Youssef, A. (2017). Serverless: beyond the cloud. *Proceedings of the 2nd International Workshop on Serverless Computing*, pp. 6–10.

Karpowicz, M. P. (2016). Energy-efficient CPU frequency control for the Linux system. *Concurrency and Computation: Practice and Experience*, 28(2), 420–437.

Kaur, M. & Aron, R. (2022). An energy-efficient load balancing approach for scientific workflows in fog computing. *Wireless Personal Communications*, 125(4), 3549–3573.

Kavanagh, R., Armstrong, D. & Djemame, K. (2016). Accuracy of energy model calibration with IPMI. *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pp. 648–655.

kepler CNCF. Kepler. Accessed: 30-01-2026.

Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K. & Ou, Z. (2018). RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2). doi: 10.1145/3177754.

Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I. & Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97, 219–235.

Khullar, R. & Hossain, G. (2022). A New Algorithm for Energy Efficient Task Scheduling Towards Optimal Green Cloud Computing. *2022 IEEE/ACS 19th International Conference on Computer Systems and Applications (AICCSA)*, pp. 1–6.

Kim, W., Gupta, M. S., Wei, G.-Y. & Brooks, D. (2008). System level analysis of fast, per-core DVFS using on-chip switching regulators. *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 123–134.

Kolpe, T., Zhai, A. & Sapatnekar, S. S. (2011). Enabling improved power management in multicore processors through clustered DVFS. *2011 Design, Automation & Test in Europe*, pp. 1–6.

Kothari, N. & Bhattacharya, A. (2009). Joulemeter: Virtual machine power measurement and management. *MSR Tech Report*.

Kraemer, F. A., Braten, A. E., Tamkittikhun, N. & Palma, D. (2017). Fog computing in healthcare–a review and discussion. *IEEE Access*, 5, 9206–9222.

Kubernetes. Overview. Accessed: 2025-05-19, Consulted at https://kubernetes.io/docs/concepts/overview/.

Kubernetes. (2025). Pod QoS Classes. Accessed: 2025-05-19.

Kubernetes Authors. (2024). Set a Container's Memory and CPU Requests and Limits. Accessed: 2025-05-20.

Kumbhare, A. G., Azimi, R., Manousakis, I., Bonde, A., Frujeri, F., Mahalingam, N., Misra, P. A., Javadi, S. A., Schroeder, B., Fontoura, M. et al. (2021). {Prediction-Based} power oversubscription in cloud platforms. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 473–487.

Labassi, L., Bounour, A. & Boumerdassi, S. (2019). Co-optimizing latency and energy for IoT services using heterogeneous multicore processors in fog clusters. *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 121–128. doi: 10.1109/FMEC.2019.8795353.

Le Sueur, E. & Heiser, G. (2010). Dynamic voltage and frequency scaling: The laws of diminishing returns. *Proceedings of the 2010 international conference on Power aware computing and systems*, pp. 1–8.

LeBeane, M., Ryoo, J. H., Panda, R. & John, L. K. (2015). Watt watcher: fine-grained power estimation for emerging workloads. *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 106–113.

Lefurgy, C., Wang, X. & Ware, M. (2007). Power capping: a prelude to power shifting. *Cluster Computing*, 11(2), 183–195. doi: 10.1007/s10586-007-0045-3.

Lenovo Press. Using Processor Idle C-States with Linux on ThinkSystem Servers. Accessed: 2025-05-19, Consulted at https://lenovopress.lenovo.com/lp1945-using-processor-idle-c-states-with-linux-on-thinksystem-servers.

Li, C., Li, Y., Skarlatos, D. & Kozyrakis, C. (2020). Twig: Energy-Efficient QoS-Aware Co-Location via Deep Reinforcement Learning. *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 259–272. doi: 10.1109/HPCA47549.2020.00023.

Li, J., Kulkarni, S. G., Ramakrishnan, K. & Li, D. (2019). Understanding open source serverless platforms: Design considerations and performance. *Proceedings of the 5th international workshop on serverless computing*, pp. 37–42.

Li, W. & co authors. (2024). Characterizing LLM Inference Energy-Performance Tradeoffs across Workloads and GPU Scaling.

Li, X., Kang, P., Molone, J., Wang, W. & Lama, P. (2022a). KneeScale: Efficient Resource Scaling for Serverless Computing at the Edge. *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 180–189.

Li, Y., Zhou, D. & Liao, X. (2023). CPU frequency scheduling using reinforcement learning for periodic real-time applications. *Journal of Systems Architecture*, 142, 102923. doi: 10.1016/j.sysarc.2023.102923.

Li, Z., Chen, Q. & Guo, M. (2021a). Pagurus: Eliminating Cold Startup in Serverless Computing with Inter-Action Container Sharing. *arXiv preprint arXiv:2108.11240*.

Li, Z., Guo, L., Cheng, J., Chen, Q., He, B. & Guo, M. (2021b). The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Computing Surveys (CSUR)*.

Li, Z., Guo, L., Cheng, J., Chen, Q., He, B. & Guo, M. (2022b). The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 54(10s), 1–34.

Lin, C. & Khazaei, H. (2020). Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3), 615–632.

Lin, C., Mahmoudi, N., Fan, C. & Khazaei, H. (2022). Fine-grained performance and cost modeling and optimization for faas applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(1), 180–194.

Lin, C.-C., Liu, P. & Wu, J.-J. (2011). Energy-efficient virtual machine provision algorithms for cloud systems. *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pp. 81–88.

Lin, C.-C., Chen, J.-J., Liu, P. & Wu, J.-J. (2018). Energy-efficient core allocation and deployment for container-based virtualization. *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 93–101.

Lin, W., Shi, F., Wu, W., Li, K., Wu, G. & Mohammed, A.-A. (2020). A taxonomy and survey of power models and power modeling for cloud servers. *ACM Computing Surveys (CSUR)*, 53(5), 1–41.

Liu, J., Yang, P. & Chen, C. (2023). Intelligent energy-efficient scheduling with ant colony techniques for heterogeneous edge computing. *Journal of Parallel and Distributed Computing*, 172, 84–96.

Liu, L., Tan, H., Jiang, S. H.-C., Han, Z., Li, X.-Y. & Huang, H. (2019). Dependent task placement and scheduling with function configuration in edge computing. *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pp. 1–10.

Liu, Y., Draper, S. C. & Kim, N. S. (2014). SleepScale: Runtime joint speed scaling and sleep states management for power efficient data centers. *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, pp. 313–324. doi: 10.1109/ISCA.2014.6853235.

Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A. & Kozyrakis, C. (2014). Towards energy proportionality for large-scale latency-critical workloads. *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, pp. 301–312. doi: 10.1109/ISCA.2014.6853237.

Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A. & Kozyrakis, C. (2015). Heracles: Improving resource efficiency at scale. *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 450–462. doi: 10.1145/2749469.2750378.

Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A. & Kozyrakis, C. (2017). Hipster: Hybrid power management for latency-critical cloud workloads. *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 409–421. doi: 10.1109/HPCA.2017.13.

Long, S., Li, Y., Huang, J., Li, Z. & Li, Y. (2022). A review of energy efficiency evaluation technologies in cloud data centers. *Energy and Buildings*, 111848.

Lu, T., Zeng, F., Chen, G., Shu, W., Shen, J. & Zhang, W. (2021a). A Novel Hybrid Model for Task Dependent Scheduling in Container-based Edge Computing. *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–7.

Lu, T., Zeng, F., Shen, J., Chen, G., Shu, W. & Zhang, W. (2021b). A Scheduling Scheme in a Container-Based Edge Computing Environment Using Deep Reinforcement Learning Approach. *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 56–65.

Lyu, X., Cherkasova, L., Aitken, R., Parmer, G. & Wood, T. (2022). Towards efficient processing of latency-sensitive serverless DAGs at the edge. *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, pp. 49–54.

Ma, H., Huang, P., Zhou, Z., Zhang, X. & Chen, X. (2022). GreenEdge: Joint green energy scheduling and dynamic task offloading in multi-tier edge computing systems. *IEEE Transactions on Vehicular Technology*, 71(4), 4322–4335.

Macken, P., Degrauwe, M., Van Paemel, M. & Oguey, H. (1990). A voltage reduction technique for digital systems. *1990 37th IEEE International Conference on Solid-State Circuits*, pp. 238–239.

Mampage, A., Karunasekera, S. & Buyya, R. (2021). A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Computing Surveys (CSUR)*.

Mampage, A., Karunasekera, S. & Buyya, R. (2022). A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 54(11s), 1–36.

Mao, Y., You, C., Zhang, J., Huang, K. & Letaief, K. B. (2017). A survey on mobile edge computing: The communication perspective. *IEEE communications surveys & tutorials*, 19(4), 2322–2358.

Marantos, C., Salapas, K., Papadopoulos, L. & Soudris, D. (2021). A flexible tool for estimating applications performance and energy consumption through static analysis. *SN Computer Science*, 2(1), 1–11.

Marinescu, D. C. (2023). Chapter 5 - Cloud resource virtualization. In Marinescu, D. C. (Ed.), *Cloud Computing (Third Edition)* (ed. Third Edition, pp. 135-173). Morgan Kaufmann. doi: https://doi.org/10.1016/B978-0-32-385277-7.00012-9.

Marquez, E. (2024). Decode serverless pricing in AWS to avoid high costs. Accessed: 2024-03-12.

Masip-Bruin, X., Marín-Tordera, E., Tashakor, G., Jukan, A. & Ren, G.-J. (2016). Foggy clouds and cloudy fogs: a real need for coordinated management of fog-to-cloud computing systems. *IEEE Wireless Communications*, 23(5), 120–128.

Masouros, D., Xydis, S. & Soudris, D. (2021). Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(1), 184–198. doi: 10.1109/TPDS.2020.3013948.

Mazouz, A., Laurent, A., Pradelle, B. & Jalby, W. (2014). Evaluation of CPU frequency transition latency. *Computer Science-Research and Development*, 29(3-4), 187–195.

McBride, S. P. (2021). *SledgeEDF: Deadline-Driven Serverless for the Edge*. (Ph.D. thesis, The George Washington University).

Mehta, H. K., Harvey, P., Rana, O., Buyya, R. & Varghese, B. (2020). WattsApp: Power-aware container scheduling. *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 79–90.

Mei, X., Wang, Q., Chu, X., Liu, H., Leung, Y.-W. & Li, Z. (2021). Energy-aware task scheduling with deadline constraint in dvfs-enabled heterogeneous clusters. *arXiv preprint arXiv:2104.00486*.

Meisner, D. & Wenisch, T. F. (2010). Peak power modeling for data center servers with switched-mode power supplies. *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pp. 319–324.

Mishra, S. K., Parida, P. P., Sahoo, S., Sahoo, B. & Jena, S. K. (2018). Improving energy usage in cloud computing using DVFS. In *Progress in Advanced Computing and Intelligent Engineering* (pp. 623–632). Springer.

Mokaripoor, P. & Hosseini Shirvani, M. (2016). A state of the art survey on DVFS techniques in Cloud Computing Environment. *J. Multidiscip. Eng. Sci. Technol*, 3(5), 4740–4743.

Mucci, P., Browne, S., Deane, C. & Ho, G. (2015). Performance Application Programming Interface (PAPI).

Nadgowda, S., Suneja, S., Bila, N. & Isci, C. (2017). Voyager: Complete container state migration. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2137–2142.

Nadjahi, C., Louahlia, H. & Lemasson, S. (2018). A review of thermal management and innovative cooling strategies for data center. *Sustainable Computing: Informatics and Systems*, 19, 14–28.

Naha, R., Garg, S., Battula, S. K., Amin, M. B. & Georgakopoulos, D. (2022). Multiple linear regression-based energy-aware resource allocation in the fog computing environment. *Computer Networks*, 216, 109240.

Napieralla, J. (2020). Considering webassembly containers for edge computing on hardware-constrained iot devices.

Nathuji, R., Kansal, A. & Ghaffarkhah, A. (2010). Q-clouds: managing performance interference effects for qos-aware clouds. *Proceedings of the 5th European conference on Computer systems*, pp. 237–250.

Newman, S. (2021). *Building microservices*. " O'Reilly Media, Inc.".

Nine, M. S. Q. Z., Kosar, T., Bulut, M. F. & Hwang, J. (2023). GreenNFV: Energy-Efficient Network Function Virtualization with Service Level Agreement Constraints. *Proceedings of the 2023 ACM/IEEE Supercomputing Conference (SC)*. doi: 10.1145/3581784.3607090.

Nishtala, R., Petrucci, V., Carpenter, P. & Sjalander, M. (2021). Cuttlefish: Dynamic optimization of core and uncore frequency scaling. Preprint or unpublished work; no DOI available at the time of writing.

Novaković, D., Vasić, N., Novaković, S., Kostić, D. & Bianchini, R. (2013). {DeepDive}: Transparently identifying and managing performance interference in virtualized environments. *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 219–230.

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. & Arpaci-Dusseau, R. (2018). {SOCK}: Rapid Task Provisioning with {Serverless-Optimized} Containers. *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 57–70.

Onnebrink, G., Walbroel, F., Klimt, J., Leupers, R., Ascheid, G., Murillo, L. G., Schürmans, S., Chen, X. & Harn, Y. (2017). DVFS-enabled power-performance trade-off in MPSoC SW application mapping. *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 196–202.

Ournani, Z., Belgaid, M. C., Rouvoy, R., Rust, P., Penhoat, J. & Seinturier, L. (2020). Taming energy consumption variations in systems benchmarking. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 36–47.

Pahl, C., Brogi, A., Soldani, J. & Jamshidi, P. (2019). Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3), 677–692. doi: 10.1109/TCC.2017.2702586.

Palmur, M., Li, Z. & Zadok, E. (2013). Cpuidle from user space. *Stony Brook University, Stony Brook, NY, Tech. Rep. FSL-13–05*.

Pan, L., Wang, L., Chen, S. & Liu, F. (2022). Retention-aware container caching for serverless edge computing. *Proc. of IEEE INFOCOM, IEEE*.

Panda, S. K., Lin, M. & Zhou, T. (2022). Energy-efficient computation offloading with DVFS using deep reinforcement learning for time-critical IoT applications in edge computing. *IEEE Internet of Things Journal*, 10(8), 6611–6621.

Patros, P., Spillner, J., Papadopoulos, A. V., Varghese, B., Rana, O. & Dustdar, S. (2021). Toward sustainable serverless computing. *IEEE Internet Computing*, 25(6), 42–50.

Patrou, M., Kent, K. B., Siu, J. & Dawson, M. (2021). Energy and runtime performance optimization of node. js web requests. *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 71–82.

Patrou, M., Kent, K. B., Siu, J. & Dawson, M. (2022). Optimizing Energy Efficiency of Node. js Applications with CPU DVFS Awareness. *2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8.

Payara Team. (2024). Payara Server. Accessed: 2025-05-20, Consulted at https://www.payara.fish/products/payara-server/.

Phung, J., Lee, Y. C. & Zomaya, A. Y. (2019). Lightweight power monitoring framework for virtualized computing environments. *IEEE Transactions on Computers*, 69(1), 14–25.

Pietri, I. & Sakellariou, R. (2014). Energy-aware workflow scheduling using frequency scaling. *2014 43rd International Conference on Parallel Processing Workshops*, pp. 104–113.

Pijnacker, B. (2024). *Estimating Container-Level Power Usage in Kubernetes*. (Master's Thesis, University of Groningen). Consulted at https://fse.studenttheses.ub.rug.nl/34420/1/mCS2024PijnackerB.pdf.

Pijnacker, B., Setz, B. & Andrikopoulos, V. (2025a). Container-level Energy Observability in Kubernetes Clusters. *arXiv preprint arXiv:2504.10702*.

Pijnacker, B., Setz, B. & Andrikopoulos, V. (2025b). Container-Level Energy Observability in Kubernetes Clusters. *2025 11th International Conference on ICT for Sustainability*, (ICT4S 2025). doi: 10.1109/ICT4S68164.2025.00016.

Pillai, P. & Shin, K. G. (2001). Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pp. 89–102. doi: 10.1145/502034.502044.

Pilz, K. F., Mahmood, Y. & Heim, L. (2025). *AI's Power Requirements Under Exponential Growth* (Report n°RR-A3572-1). Consulted at https://www.rand.org/pubs/research_reports/RRA3572-1.html.

Podzimek, A., Bulej, L., Chen, L. Y., Binder, W. & Tuma, P. (2015). Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 1–10. doi: 10.1109/CCGrid.2015.164.

Pool, J., Wong, I. S. K. & Lie, D. (2007). Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical. *HotOS*.

Poth, A., Schubert, N. & Riel, A. (2020). Sustainability efficiency challenges of modern it architectures–a quality model for serverless energy footprint. *European Conference on Software Process Improvement*, pp. 289–301.

Qin, S., Wu, H., Wu, Y., Yan, B., Xu, Y. & Zhang, W. (2020). Nuka: A generic engine with millisecond initialization for serverless computing. *2020 IEEE International Conference on Joint Cloud Computing*, pp. 78–85.

Qiu, M., Kung, S.-Y. & Yang, Q. (2019). IEEE transactions on sustainable computing, special issue on smart data and deep learning in sustainable computing. *IEEE Transactions on Sustainable Computing*, 4(1), 1–3.

Raffin, G. & Trystram, D. (2025). Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 36(1), 96-107. doi: 10.1109/TPDS.2024.3492336.

Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z. & Zhu, X. (2008). No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. *ACM SIGARCH Computer Architecture News*, 36(1), 48–59. doi: 10.1145/1353535.1346283.

Raïs, I., Orgerie, A.-C., Quinson, M. & Lefèvre, L. (2018). Quantifying the impact of shutdown techniques for energy-efficient data centers. *Concurrency and Computation: Practice and Experience*, 30(17), e4471.

Rangan, K. K., Wei, G.-Y. & Brooks, D. (2009). Thread motion: fine-grained power management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 37(3), 302–313.

Rao, W. & Li, H. (2025a). Energy-aware Scheduling Algorithm for Microservices in Kubernetes Clouds. *Journal of Grid Computing*, 23(1), 1–22.

Rao, W. & Li, H. (2025b). Energy-aware Scheduling Algorithm for Microservices in Kubernetes Clouds. *Journal of Grid Computing*, 23(1), 1–22. doi: 10.1007/s10723-024-09788-w.

Rastegar, S. H., Shafiei, H. & Khonsari, A. (2023). EneX: An Energy-Aware Execution Scheduler for Serverless Computing. *IEEE Transactions on Industrial Informatics*.

Rausch, T., Hummer, W., Muthusamy, V., Rashed, A. & Dustdar, S. (2019). Towards a serverless platform for edge {AI}. *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

Raza, A., Matta, I., Akhtar, N., Kalavari, V. & Isahagian, V. (2021). Function-as-a-service: From an application developer's perspective. *JSys*, 1(1), 1–20.

Rejiba, Z. & Chamanara, J. (2022). Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches. *ACM Comput. Surv.*, 55(7). doi: 10.1145/3544788.

Ren, J. & Lu, K. (2023). Is Kepler Accurate on Specific Platforms? *KubeCon + CloudNativeCon + Open Source Summit, China 2023*. Consulted at https://static.sched.com/hosted_files/kccncosschn2023/45/KubeCon%20China%202023%20-%20Is%20Kepler%20Accurate%20on%20Specific%20Platforms%20-%20English.pdf.

Roberts, M. (2018). Serverless Architectures. Consulted at https://martinfowler.com/articles/serverless.html.

Roberts, M. (2020). The Fastest-Growing Cloud. Consulted at https://www.cbinsights.com/research/serverless-cloud-computing/.

Sabban, A. (2021). Introductory Chapter: Green Computing Technologies and Industry in 2021. In *Green Computing Technologies and Computing Industry in 2021*. IntechOpen.

Saenko, K. ChatGPT's Energy Consumption Compared to Google. Accessed: 12-05-2024.

Saenko, K. Generative AI's Environmental Impact and Carbon Footprint. Accessed: 03-07-2024.

Saileshwar, G., Kariyappa, S. & Qureshi, M. (2021). Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 37–49.

Samadi, N., Yadollahi, F. & Shah-Mansouri, H. (2025). iDEAS: Intelligent DVFS for Energy-Efficient Task Scheduling in Mobile Devices with big.LITTLE Computing Architecture. *IEEE Access*, 13, 200711–200724. doi: 10.1109/ACCESS.2025.3636995.

Sampaio, A. M. & Barbosa, J. G. (2016). Chapter Three - Energy-Efficient and SLA-Based Resource Management in Cloud Data Centers. In Hurson, A. R. & Sarbazi-Azad, H. (Eds.), *Energy Efficiency in Data Centers and Clouds* (vol. 100, pp. 103-159). Elsevier. doi: https://doi.org/10.1016/bs.adcom.2015.11.002.

Savasci, M., Souza, A., Wu, L., Irwin, D., Ali-Eldin, A. & Shenoy, P. (2024a). Slo-power: Slo and power-aware elastic scaling for web services. *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 136–147.

Savasci, M., Souza, A., Wu, L., Irwin, D., Ali-Eldin, A. & Shenoy, P. (2024b). SLO-Power: SLO and Power-aware Elastic Scaling for Web Services. *Proceedings of the 24th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 136–147. doi: 10.1109/CCGrid59990.2024.00025.

Saxena, D. & Singh, A. K. (2021). A proactive autoscaling and energy-efficient VM allocation framework using online multi-resource neural network for cloud data center. *Neurocomputing*, 426, 248–264.

Schmitt, N., Iffländer, L., Bauer, A. & Kounev, S. (2019). Online power consumption estimation for functions in cloud applications. *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 63–72.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sedefoğlu, Ö. & Sözer, H. (2021). Cost minimization for deploying serverless functions. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 83–85.

Sen, R. & Halverson, A. (2017). Frequency governors for cloud database OLTP workloads. *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6.

Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M. & Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218.

Shahrad, M., Elnikety, S. & Bianchini, R. (2021). Provisioning differentiated last-level cache allocations to vms in public clouds. *Proceedings of the ACM Symposium on Cloud Computing*, pp. 319–334.

Sharafzadeh, E., Kohroudi, S. A. S., Asyabi, E. & Sharifi, M. (2019). Yawn: A CPU Idle-State Governor for Datacenter Applications. *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 1–7. doi: 10.1145/3343737.3343740.

Sharma, P. (2023). Challenges and opportunities in sustainable serverless computing. *ACM SIGENERGY Energy Informatics Review*, 3(3), 53–58.

Sharma, P., Chaufournier, L., Shenoy, P. & Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. *Proceedings of the 17th international middleware conference*, pp. 1–13.

Shen, J., Yang, T., Su, Y., Zhou, Y. & Lyu, M. R. (2021). Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 194–204.

Shilkov, M. (2021a). Cold Starts in Azur. Consulted at https://mikhail.io/serverless/coldstarts/azure/.

Shilkov, M. (2021b). Internet of Things Statistics, Facts Trends for 2022. Consulted at https://findstack.com/internet-of-things-statistics/.

Shillaker, S. & Pietzuch, P. (2020). Faasm: Lightweight isolation for efficient stateful serverless computing. *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433.

Shojafar, M., Cordeschi, N., Amendola, D. & Baccarelli, E. (2015). Energy-saving adaptive computing and traffic engineering for real-time-service data centers. *2015 IEEE international conference on communication workshop (ICCW)*, pp. 1800–1806.

Silva-de Souza, W., Iranfar, A., Bráulio, A., Zapater, M., Xavier-de Souza, S., Olcoz, K. & Atienza, D. (2020). Containergy—a container-based energy and performance profiling tool for next generation workloads. *Energies*, 13(9), 2162.

Singh, B. P., Kumar, S. A., Gao, X.-Z., Kohli, M. & Katiyar, S. (2020). A study on energy consumption of DVFS and Simple VM consolidation policies in cloud computing data centers using CloudSim Toolkit. *Wireless Personal Communications*, 112(2), 729–741.

Singh, S. P., Kumar, R., Sharma, A., Abawajy, J. H. & Kaur, R. (2022a). Energy efficient load balancing hybrid priority assigned laxity algorithm in fog computing. *Cluster Computing*, 25(5), 3325–3342.

Singh, S. P., Kumar, R., Sharma, A. & Nayyar, A. (2022b). Leveraging energy-efficient load balancing algorithms in fog computing. *Concurrency and Computation: Practice and Experience*, 34(13), e5913.

Singh, T., Rangarajan, S., John, D., Henrion, C., Southard, S., McIntyre, H., Novak, A., Kosonocky, S., Jotwani, R., Schaefer, A. et al. (2017). 3.2 Zen: A next-generation high-performance× 86 core. *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 52–53.

Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S. & Akella, A. (2021). Atoll: A Scalable Low-Latency Serverless Platform. *Proceedings of the ACM Symposium on Cloud Computing*, pp. 138–152.

Skutella, M. (1998). Approximation algorithms for the discrete time-cost tradeoff problem. *Mathematics of Operations Research*, 23(4), 909–929.

Sohal, P., Bechtel, M., Mancuso, R., Yun, H. & Krieger, O. (2022). A closer look at intel resource director technology (rdt). *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pp. 127–139.

Song, S., Xu, M., Zhang, Z., Gao, C., Zeng, F., Ding, Y., Ye, K. & Xu, C. (2026). C-Koordinator: Interference-Aware Management for Large-Scale and Co-Located Microservice Clusters. *Software: Practice and Experience*. doi: 10.1002/spe.70059.

Sonmez, C., Ozgovde, A. & Ersoy, C. (2018). Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11), e3493.

Standard Performance Evaluation Corporation (SPEC). (2008). SPECpower®: Power and Performance Benchmarking. Accessed: Feb. 16, 2025, Consulted at https://www.spec.org/power_ssj2008/.

Stojkovic, J., Iliakopoulou, N., Xu, T., Franke, H. & Torrellas, J. (2024). EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency. *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 471–486. doi: 10.1109/ISCA59077.2024.00042.

Sun, B. J. J. & Ranjan, R. Serverless Architecture for Edge Computing.

Suo, K., Shi, Y., Xu, X., Cheng, D. & Chen, W. (2020). Tackling cold start in serverless computing with container runtime reusing. *Proceedings of the Workshop on Network Application Integration/CoDesign*, pp. 54–55.

Sustainable Computing IO. (2025). Kepler Models. Accessed: Feb. 16, 2025, Consulted at https://github.com/sustainable-computing-io/kepler-model-db/tree/main.

Tang, C., Zhu, C., Wu, H., Li, Q. & Rodrigues, J. J. (2021). Toward response time minimization considering energy consumption in caching-assisted vehicular edge computing. *IEEE Internet of Things Journal*, 9(7), 5051–5064.

Tang, Q., Xie, R., Yu, F. R., Huang, T. & Liu, Y. (2020). Decentralized computation offloading in IoT fog computing system with energy harvesting: A dec-POMDP approach. *IEEE Internet of Things Journal*, 7(6), 4898–4911.

Tang, W., Ke, Y., Fu, S., Jiang, H., Wu, J., Peng, Q. & Gao, F. (2022). Demeter: QoS-aware CPU scheduling to reduce power consumption of multiple black-box workloads. *Proceedings of the 13th Symposium on Cloud Computing*, pp. 31–46.

Thömmes, M. (2020). How to make serverless platforms blazing fast. [Accessed: Jul. 9, 2024], Consulted at https://bit.ly/3XY0XxE.

Toor, A., ul Islam, S., Sohail, N., Akhunzada, A., Boudjadar, J., Khattak, H. A., Din, I. U. & Rodrigues, J. J. (2019). Energy and performance aware fog computing: A case of DVFS and green renewable energy. *Future Generation Computer Systems*, 101, 1112–1121.

Tuli, S., Casale, G. & Jennings, N. R. (2022). SplitPlace: AI Augmented Splitting and Placement of Large-Scale Neural Networks in Mobile Edge Environments. *IEEE Transactions on Mobile Computing*.

Tzenetopoulos, A., Marantos, C., Gavrielides, G., Xydis, S. & Soudris, D. (2021). FADE: FaaS-inspired application decomposition and Energy-aware function placement on the Edge. *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, pp. 7–10.

Tzenetopoulos, A., Masouros, D., Soudris, D. & Xydis, S. (2023). DVFaaS: Leveraging DVFS for FaaS workflows. *IEEE Computer Architecture Letters*.

Tzenetopoulos, A., Masouros, D., Xydis, S. & Soudris, D. (2024). Leveraging Core and Uncore Frequency Scaling for Power-Efficient Serverless Workflows. *arXiv preprint arXiv:2407.18386*. doi: 10.48550/arXiv.2407.18386.

Ubuntu Manpage Repository. (2024). stress-ng — Linux manual page. Accessed: 2025-05-20.

Ukarande, A., Basaklar, T., Cao, M. & Ogras, U. (2024). PACT: Accurate Power Analysis and Carbon Emission Tracking for Sustainability. *Proceedings of the 29th ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 1–6.

Vamanan, S. U., Ansari, T. E. & Vijaykumar, T. N. (2014). TimeTrader: Exploiting latency slack to save energy in online data-intensive applications. Extended technical report; DOI not assigned.

Wang, A., Chang, S., Tian, H., Wang, H., Yang, H., Li, H., Du, R. & Cheng, Y. (2021). {FaaSNet}: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 443–457.

Wang, Y., Yang, X., Zhou, X. & Wu, J. (2022a). Online power management for multi-cores: a reinforcement learning based approach. *IEEE Transactions on Parallel and Distributed Systems*, 33(4), 751–764. doi: 10.1109/TPDS.2021.3092270.

Wang, Y., Shi, L., Lee, M. H., Sydir, J., Zhou, Z., Chi, Y. & Li, B. (2024). Scalable Dynamic Resource Allocation via Domain Randomized Reinforcement Learning. The RL-ADR approach for 5G UPF frequency scaling and power reduction; no DOI assigned at the time of writing.

Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K., Zheng, Y., Yan, M., Zhang, X. & Liu, A. X. (2022b). DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. *Proceedings of the 13th Symposium on Cloud Computing*, pp. 16–30.

webassembly. (2022). Compile a WebAssembly. Consulted at https://webassembly.org/getting-started/developers-guide/.

Weiser, M., Welch, B., Demers, A. & Shenker, S. (1996). Scheduling for reduced CPU energy. *Mobile Computing*, 449–471.

WikiChip. [Accessed 14 Jul 2025]. (2025). Zen Microarchitectures — Power Management. Consulted at https://en.wikichip.org/wiki/amd/microarchitectures/zen.

Wu, C., Buyya, R. & Ramamohanarao, K. (2019). Cloud pricing models: Taxonomy, survey, and interdisciplinary challenges. *ACM Computing Surveys (CSUR)*, 52(6), 1–36.

Wu, M., Mi, Z. & Xia, Y. (2020). A survey on serverless computing and its implications for jointcloud computing. *2020 IEEE International Conference on Joint Cloud Computing*, pp. 94–101.

Xie, R., Tang, Q., Qiao, S., Zhu, H., Yu, F. R. & Huang, T. (2021). When serverless computing meets edge computing: architecture, challenges, and open issues. *IEEE Wireless Communications*, 28(5), 126–133.

Yang, W., Zhao, M., Li, J. & Zhang, X. (2024). Energy-efficient DAG scheduling with DVFS for cloud data centers. *The Journal of Supercomputing*, 1–25.

Yao, F. F., Demers, A. J. & Shenker, S. (1995). A Scheduling Model for Reduced CPU Energy. *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 374–382. doi: 10.1109/SFCS.1995.492493.

Yasin, A. (2014). A Top-Down Method for Performance Analysis and Counters Architecture. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44. doi: 10.1109/ISPASS.2014.6844459.

Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C. & Chen, H. (2020). Characterizing serverless platforms with serverlessbench. *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 30–44.

Zhang, W., Wen, Y. & Wu, D. O. (2013). Energy-efficient scheduling policy for collaborative execution in mobile cloud computing. *2013 Proceedings Ieee Infocom*, pp. 190–194.

Zhang, Y.-W. (2021). Energy-Aware Nonpreemptive Scheduling of Mixed-Criticality Real-Time Task Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(9), 2891–2900.

Zhang, Y., Wang, Y. & Hu, C. (2015). CloudFreq: Elastic energy-efficient bag-of-tasks scheduling in DVFS-enabled clouds. *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 585–592.

Zhang, Z., Zhao, Y., Chang, M., Lin, C. & Liu, J. (2025). E4: Energy-Efficient DNN Inference for Edge Video Analytics via Early Exiting and DVFS. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Zhao, G., Xu, H., Zhao, Y., Qiao, C. & Huang, L. (2021). Offloading tasks with dependency and service caching in mobile edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(11), 2777–2792.

Zhao, N., Tarasov, V., Albahar, H., Anwar, A., Rupprecht, L., Skourtis, D., Warke, A. S., Mohamed, M. & Butt, A. R. (2019). Large-scale analysis of the docker hub dataset. *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10.

Zhao, P., Li, X. & co authors. (2023). Condense: Adapting Neural Models for Robust Performance Across Device Frequencies. *Proceedings of the Design Automation Conference (DAC)*.

Zhao, Y. & co authors. (2023). DVFO: Energy-Efficient Deep Reinforcement Learning for DVFS and Offloading in Edge AI Systems.

Zhou, Z., Li, K., Abawajy, J., Shojafar, M., Chowdhury, M., Li, F. & Li, K. (2022). An Adaptive Energy-Aware Stochastic Task Execution Algorithm in Virtualized Networked Datacenters. *IEEE Transactions on Sustainable Computing*, 7(2), 371-385. doi: 10.1109/TSUSC.2021.3115388.

Zhu, Y. et al. (2024). Hierarchical reinforcement learning for NFV server power management. Patent application describing a hierarchical RL controller for core and uncore frequency; no peer-reviewed publication.

Étienne André, Dulong, R., Guermouche, A. & Trahay, F. (2022). duf: Dynamic uncore frequency scaling to reduce power consumption. *Concurrency and Computation: Practice and Experience*, 34(3). doi: 10.1002/cpe.6580.