2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-6654-7315-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPADS56603.2022.00082

Service Level Objective Adaptive Energy Efficiency Management

Chuan Song
Intel Asia-Pacific Research & Development Ltd.
Shanghai, China
edmund.song@intel.com

Feng Jiang
Intel Asia-Pacific Research & Development Ltd.
Shanghai, China
feng.jiang@intel.com

Xiaoguo Liang
Intel Asia-Pacific Research & Development Ltd.
Shanghai, China
xiaoguo.liang@intel.com

Nishi Ahuja
Intel Corp.
Hillsboro, Oregon, USA
nishi.ahuja@intel.com

Mohan J. Kumar
Intel Corp.
Hillsboro, Oregon, USA
mohan.j.kumar@intel.com

***Abstract*: Cloud applications are transforming to microservice based deployment. Microservice comes with many advantages - scalability, agility, availability – and it suits the complicated cloud applications. For microservice deployment, service quality metrics play important role to ensure reliable user experiences for those applications built upon many small services, and the service level objective (SLO) is adopted to govern resource provision for optimal operation cost. Amid growing awareness on data center carbon emission, more organizations are paying attention to energy-efficient practices in cloud deployment to reduce energy consumption and operational carbon footprint. Modern processors provide fine-grained power scaling capabilities, e.g., Intel Xeon PCPS (Per-Core P-States), this paper presented a dynamic energy efficiency management framework to adaptively scale processor core frequency as well as implicated energy consumption in accordance with service quality requirements. In this paper, we discussed hurdles and solutions for applying platform energy intelligence to microservices that are often with intrinsic platform-agnostic. The experiments proved the proposed method can effectively reduce energy consumption with competitive cost benefits without compromising service quality goals.**

Keywords—Microservices, Service Mesh, Per-Core P-States, Service Level Objective (SLO), Service Level Indicator (SLI), Energy Efficiency, Carbon Reduction, Sustainability, Adaptive Control, Core Frequency Scaling

I. INTRODUCTION

The microservice architecture is a recent popular computing paradigm and has been increasingly adopted for various cloud applications in last few years, where a collection of loose-coupling and small services cooperates to serve application requests. Comparing to monolithic applications, microservices offer agile resource management, elastic scaling capability, failure resilience and high availability. Besides, along its advantages on scalability and flexibility, it also greatly increases the complexity for service management. To ease microservice management, service mesh is introduced into microservices, which allows segregating service infrastructure capabilities apart from application logics, including telemetry, network, and security. Service mesh improves application observability, enhance security and reduce deployment complexity.

The cloud applications need to meet a bunch of performance metrics, such as response time, throughput, to ensure that the offered service level is consistent with business needs. The service level objective (SLO) serves as a set of quality indicators defined with specific service level target. The goal of defining SLOs is to deliver reliable end-to-end user experiences for those cloud applications in supporting certain business objectives, under which the overall cost is equally important as user experiences, so usually SLO is not defined in terms of best performance offerings, but by customer expectations under certain premise. Based on this assumption, an SLO is given to govern resource provision and delivery to achieve the optimal business result, e.g., TCO (total cost of ownership), resource utilization, or energy consumption.

Data centers are among the highest consumers of electric power and recent studies showed that data center energy demands continue increasing speedily. To avert impacts of climate change and preserve a sustainable development, the global is pledging net-zero race by 2050. In response the global effort to reach net-zero carbon emission, more organizations are putting effort to adopt energy-efficient design methodology into cloud deployment to reduce data center energy consumption as well as operational carbon footprint. This paper focuses on energy efficiency design for microservices. The principle of energy efficiency design is to minimize energy consumption to a proper level at which the desired performance (depicted with SLO) can be met without service quality degradation. Although microservices bring advantages in many aspects, due to its inherent capability on platform agnostic, unlike traditional services architecture, the complexity of energy efficiency management does increase – i.e., fine-grained energy control is closely related to platform specific capabilities, or in other words platform-awareness, exactly the opposite of platform-agnostic.

In this paper, we explored key hurdles while applying energy efficiency design to microservices as well as corresponding solutions; and a proposal is put forward to dynamically scale processor or platform energy consumption to match quality of service (QoS) requirements along with actual service loading, so as to achieve the goal of the optimal energy cost. The remainder of this article is organized as follows: Section II discusses problems and challenges for enforcing energy efficiency design into microservices; Section III details design methodology, implementation architecture and algorithms to scale processor core frequency in line with service quality objective; Section VI talks about experiments setup, evaluation results, and benefits analysis.

II. BACKGROUND

For cloud applications, the microservice architecture comes with great benefits on scalability, availability, resiliency over monolithic service. As the foundation of cloud

978-1-6654-7315-6/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPADS56603.2022.00082

applications is shifting to microservices, service level objectives (SLOs) are widely adopted as the basic metrics to define service quality level as well as to guide resource provision. In large-scale cloud system, the microservice-based applications specify SLO metric for each small service included to ensure end-to-end service quality can satisfy user experience [1]. For an application chained with many microservices, building a comprehensive service quality map is both critical and challenging. The emergence of service mesh ease end-to-end service quality management through dedicating service metrics monitor and collecting into a single and uniform framework, named service control plane. The state-of-art service mesh includes Istio, Linkerd etc.

Reducing the power consumption in cloud computing environment has drawn a lot of attention along with increasing energy demand from cloud data center. Dynamically adapting processor voltage and frequency to system loading is a common technique to manage power consumption in design of traditional operation systems. As cloud computing becomes mainstream, how to take account of energy efficiency in complex cloud deployment turns to be a primary challenge.

A. *Problem Statement*

Most of modern CPU processors support dynamic voltage and frequency scaling (DVFS). Furthermore, the state-of-art multicore processor allows more advanced fine-grained power control technologies, e.g. refining voltage and frequency control to processor core granularity, separating core power control from other chip components (e.g. un-core).

All modern operation systems design has dedicated kernel module for dynamic power and performance management, for example, Linux uses governor mechanism to steer CPU processor frequency according to resource utilization and system idleness. However, for cloud service deployment, using system level resource loading and idleness metrics are not able to exactly reflect service quality metrics and SLOs belonging to specific workload or application services, thus the application of kernel governor policy often interferes with actual quality demands from upper-layer application services. Therefore, most of cloud operators choose disabling native governor policy to prevent its impacts on fluctuation-sensitive service.

To guarantee a service stability for consistent performance output, cloud operators usually define a deterministic processor operation frequency F in line with the performance requirement at peak loading, thus specify processor to this frequency F. Because the loading of a cloud service fluctuates rapidly with changes in users' behaviors, the state of locking processor operation frequency to a certain value under peak loading results in overprovision for computing and energy resource.

In cloud computing, autoscaling is a popular feature to adapt computational capabilities delivered to a cloud application with dynamic and unpredictable loading. The autoscaling allows cloud orchestration software to automatically scale resource based on defined criteria, such as throughput or resource utilization. Autoscaling can be performed from two dimensions: horizontal scaling (a.k.a. scale out/in) by increasing or decreasing replica number, and vertical scaling (a.k.a. scale up/down) by resizing resources for same replica [2]. However, the major decision factor of autoscaling concentrate on resource domain, like vCPU number, memory size, IO bandwidth etc., and it doesn't take account of core frequency change and energy efficiency goals.

In summary, here are some key challenges while enforcing energy efficiency design to microservices:

First, locking the operating frequency of processor with certain value for peak loading fails to accommodate the variation of power demand due to service loading fluctuation. The operating state of power overprovision wastes energy and computing resource, meanwhile it increases operational cost and drives up total cost of ownership.

Second, for today's microservices, the system power management and service quality monitoring fall into two isolated layers, resource layer and service layer. Ignoring service quality metrics in system power control leads that system power steering is out of sync with service demands.

Third, the autoscaling purely relying on resource allocation path generally is accompanied with slower response and higher overhead.

B. *Novelty and Contributions*

To address above challenges discussed in section A, this paper makes following contributions:

- ✓ Design one framework called SLO-AEEM (SLO Adaptive Energy Efficiency Management) for microservices to incorporate service quality metrics into system power control and establish one mechanism for adapting processor core frequency to service loading and quality goals.
- ✓ Explore closed-loop control policy and power scaling algorithms and energy efficiency benefits. Discuss the method to expand microservice autoscaling capability to power domain.
- ✓ Define evaluation method and conduct lab experiments for applying energy efficiency practices into microservice deployment.

III. ARCHITECTURE AND IMPLEMENTATION

Cloud service infrastructure evolution to microservice dramatically reduces operational burden for complicated cloud deployment. Service mesh introduces one dedicated service infrastructure layer to handle service-to-service communication and implements one array of lightweight proxies deployed alongside application logic without application needing to be aware [3, 4].

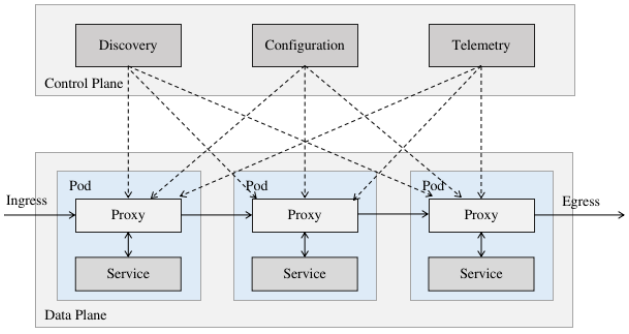

Figure 1 Service Mesh Diagram

A service mesh infrastructure, e.g., Istio, logically consists of a data plane and a control plane [5]. The data plane handles ingress traffic and egress traffic, and it is implemented to be one transparent proxy for attached application. The control plane acts as the brain of a service mesh to manage and configure service proxies deployed data plane [3]. Figure 1 shows typical diagram for service mesh. Usually, the control plane of microservice is designed as one standalone layer in independent of resource orchestration stack, e.g., Kubernetes, which is responsible for underlying resource scheduling and provision. One advantage from this design is the platform agnostic capability providing great scalability for deployment, but it also implicates disadvantages or complexities for applying platform specific capabilities.

An orchestration stack consists of a set of worker node to host service applications, and some master nodes to host controller. The controller manages worker nodes and the Pods in the cluster. The worker node hosts Pod(s) as the underlying resource container for application service, as well as maintain runtime environment.

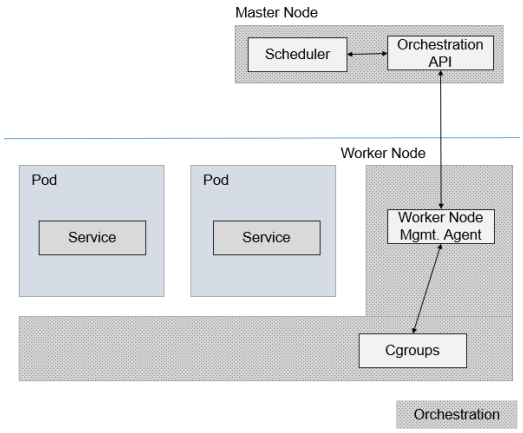

Figure 2 Orchestration Stack

A. Architecture and Framework

As we have discussed in section II, while integrating power management into microservices, one key hurdle is that the control of system power and the collecting of service quality metrics belong to separated layers, resource layer and service layer. The design purpose is stick system power to service loading and quality targets (SLO), the power management system needs to timely sense the change of service loading. The SLO-AEEM framework implements three components to break down boundary between resource layer and service layers: MM-HUB (Metrics Management HUB), RM-HUB (Resource Mapping Hub), PM-Agent (Power Management Agent), as shown in Figure 3.

Generally, an application service is only bound to resource bearing layer with resource quote, e.g., vCPU ratio, memory ratio etc., but is not aware of the platform specific information where to host allocated resources. On orchestration side, the Pod running in worker node is used to manage resource requested from application services, and it can host certain application process or multi-application processes sharing a set of resources.

Linux uses Cgroups to enforce resource for a given group of processes. Linux kernel scheduler support time sharing policy to specify a processing unit (processor core) to each process. Alternatively, the user can choose pinning mode to enforce CPU affinity binding [6], in which the kernel creates a set of processing unit (processor core) to contain process list if the process has resource affinity requirements, so that the process can only run with binding processor core(s). The SLO-AEEM presented in this paper adopts core affinity capability and pins application service to certain processor core(s). The PM-Agent determines performance states policy for core(s) running specified service according to its service quality objectives.

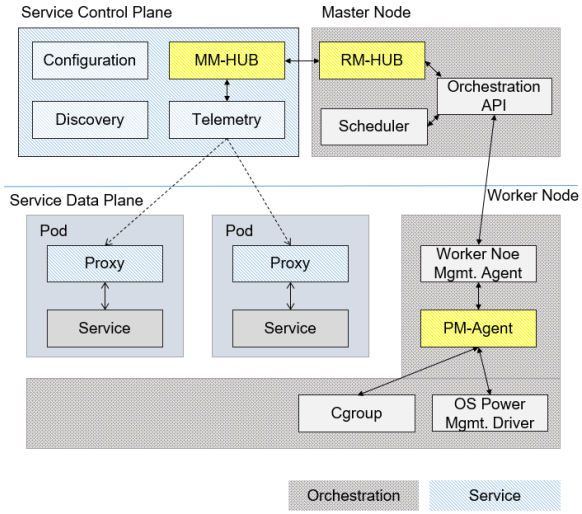

Figure 3 SLO-AEEM Architecture Diagram

Metric Management Hub (MM-HUB): MM-HUB is a software component located at the control plane of service mesh, and it is responsible for collecting service quality metrics in runtime as well as for managing metric subscription request from external orchestration stack. To interoperate with different orchestration software, MM-HUB defines RESTful APIs as interface for external application on registration, subscription etc.

Resource Mapping Hub (RM-HUB): RM-HUB is one component in orchestration controller, e.g., Kubernetes(K8S) master node. This component is used to establish service instance mapping with resource instance – they are managed by service control plane and orchestration controller respectively. In SLO-AEEM, RM-HUB implements one quintuple list:

$$(Id_{service}, Id_{node}, SLO_{target}, SLI_t)$$

Through this quintuple, RM-HUB can filter out worker node (Id_{node}) scheduled to host specified service instance ($Id_{service}$). RM-HUB subscribes service metrics events through RM-HUB Restful API and caches service quality information (SLO_{target}, SLI_t) into local database. RM-HUB uses Id_{node} to locate the targeted worker node.

Power Management Agent (PM-Agent): PM-Agent is one component running in worker node as part of node level resource management system, which creates the mapping from service instance ($Id_{service}$) to processor core set (Id_{core}) for targeted service instance. PM-Agent queries service

quality information (SLO_{target}, SLI_t) and caches into local database.

$(Id_{service}, Id_{core}, SLO_{target}, SLI_t)$

PM-Agent runs power management algorithm to adaptive core(s) power consumption according to real time service quality metrics.

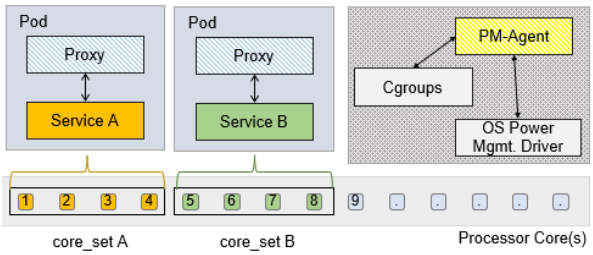

Figure 4 Core Power Management Agent

B. Methodology

Per Core Frequency Scaling

Modern processor's power consumption can be divided into two parts: the transistor leakage power, and the dynamic power, where the dynamic power dominates processor power consumption. DVFS is well-adopted technique for changing dynamic power through adjusting the clock speed of the processor. The frequency (f) at which the processor runs is proportional to the voltage (V) and the dynamic power consumed varies linearly with V^2f [6][7]. By using DFVS, users can scale the clock speed dynamically as burden changes to achieve better balance of efficiency and performance.

$$P_t = P_{leakage} + P_{dynamic}$$
$$P_{leakage} = m * V, m: constant$$
$$P_{dynamic} = CV^2f, C: capacitance$$

Intel Xeon processor supports two ways to decrease power consumption: powering down subsystems by C-states, and dynamic voltage frequency reduction by P-states. C-states are idle states, which are only available when system runs into idle states. P-states are the executing states, and while the running system does not require full performance so the voltage and/or frequency it operates can be decreased [8]. The recent state-of-the-art Intel Xeon processors (e.g., Haswell-EP) with fully integrated voltage regulators (FIVR) can provide an individual voltage for each core to support per core power scaling [9], as known as Intel Per-Core P-States (PCPS). The Per-Core P-States enables runtime services and operation system to lower the power consumption of single core without inference other cores performance status.

Service Quality Metrics

Assuring service quality is a common and critical goal for all online cloud services. Microservice offers flexibility, resiliency, and scalability as new cloud architecture. High service quality assurance requires to establish one comprehensive quality metrics across all individual services chained together to serve user application service. The commonly used service quality metrics includes latency, transaction rate, and throughput.

Latency vs. Tail Latency: Service latency is the duration taken to serve a service request, defined as the delta time between when the request was received and when a response was returned. Because a delayed response causes poor service quality and worsens user experiences, it is important to measure the duration of the service. Tail latency, known as high-percentile latency, refers to the small percentage of response times from a system out of all its responses to served requests. Compared to the average latency, which is the sum of latencies divided by the count and usually masks the long tail of response time, the tail latency is a more suitable indicator to measure service quality on duration.

Throughput: The throughput is the rate of data or requests which is served by a service can serve over a sustained period, given as a rate: the number of requests per second or the mega-bytes per second.

Service Level Indicator (SLI) and Service Level Objective (SLO): To track the health of a service, the system needs to understand which behaviors matter for that service and how to measure and evaluate those behaviors. The service level indicator (SLI) is a quantitative metric of some aspects of the service, e.g., latency, error rate and throughput. A service level objective (SLO) defines indicators or metrics with specific service level, in which you expect a service to satisfy user experiences or application requirements. The SLO may be an optimal range or a specific value to measure each service or process constituting a cloud application.

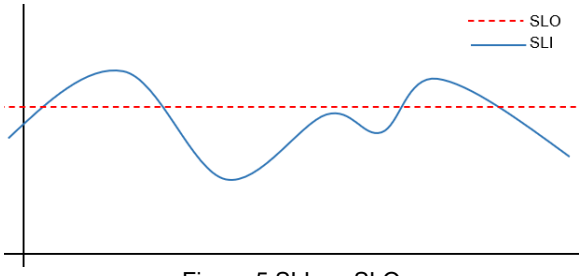

Figure 5 SLI vs. SLO

Control-loop System

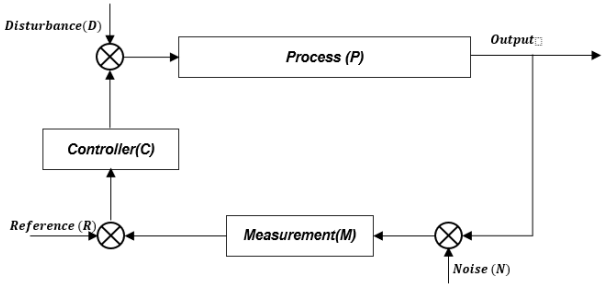

Figure 6 Closed-loop Feedback System

The deterministic core frequency setting for peak loading usually results in under-utilized core in which the available performance capability is higher than actual demand of bearing service, hence the energy is wasted. The control-loop feedback system has been widely adopted by industrial control systems to automatically adjust system behavior to be optimal for certain metrics, and it consists of input signals, output

signals, process control system, measurement units and a controller. The controller measures the output signals and manipulate input signals to drive process variables toward the desired setpoint. There are two kinds of control-loop system, the open loop system, and the closed loop system. In an open loop control system, control actions are independent of desired output, and the output is not measured or fed back to input. In contrast, the closed loop systems measure, monitor and control the process variable through feedback to compare the output against the desired setpoint. Figure 6 shows the architecture diagram from one typical closed loop feedback system consisting of a controller(C), actuator, process system and feedback measurement unit (M). The inputs to control system are reference set-point (R), disturbance(D) and sensor noise(N).

SLO-AEEM adopted closed-loop feedback system to adapt core(s) P-States according to service quality goals. MM-HUB and telemetry collector works as the measurement unit to periodically collect service quality indicator ($\boldsymbol{M_{SLI}}$) for managed service. MM-HUB and RM-HUB feed $\boldsymbol{M_{SLI}}$ to system controller (PM-Agent) and the controller drive the actuator (Per-Core P-States) to achieve optimal energy efficiency under defined service level objective.

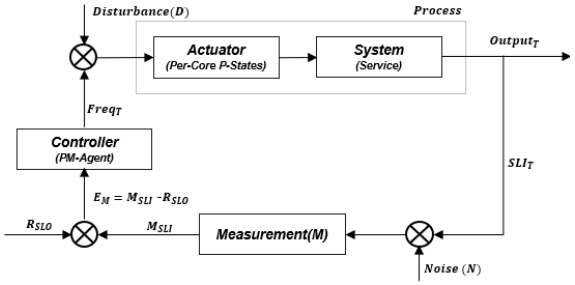

Figure 7 SLO Adaptive Power Scaling Control-loop

C. SLO Adaptive Control Algorithm

The service quality indicator ($\boldsymbol{M_{SLI}}$) is fed into the controller and compared with desired service level objectives ($\boldsymbol{R_{SLO}}$) to calculate the control error $\boldsymbol{E_M}$ ($\boldsymbol{E_M = M_{SLI} - R_{SLO}}$). The controller (PM-Agent) uses $\boldsymbol{E_M}$ as the input to generates the drive signal – Per-Core P-States – to reduces $\boldsymbol{E_M}$ until the actual output $\boldsymbol{M_{SLI}}$ equals the reference set point ($\boldsymbol{R_{SLO}}$) or remains within the pre-defined margin. In cloud services, the commonly used SLI is the tail latency calculated in percentiles, e.g., 90th, 95th, and 99th percentiles, commonly referred to as p90, p95 and p99.

The major objective of SLO adaptive control is to adaptively tune core frequency setting to ensure the output quality indicator ($\boldsymbol{M_{SLI}}$) meet service level objectives($\boldsymbol{R_{SLO}}$) - $\boldsymbol{R_{SLO}}$ is the target line of control system. In control loop feedback system, given noises and disturbances are evitable in phases of measurement and process, usually, we need to define one target region $\boldsymbol{R_{Margin}}$ instead of the single value ($\boldsymbol{R_{SLO}}$) to hedge potential overshoot caused by disturbance, noisy, and control lag. Figure 6 depicts on example to define the target region as 10 percent of target value: $\boldsymbol{R_{Margin} = 10\% \, of \, R_{SLO}}$.

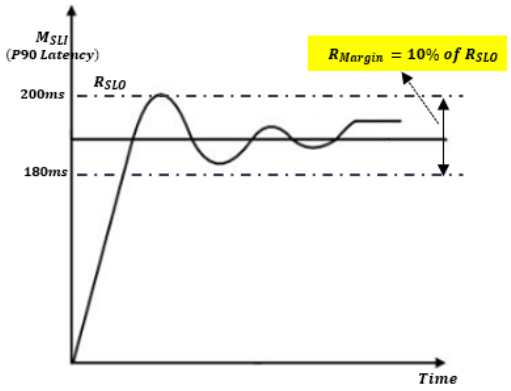

Figure 8 Control Target and Margin

Per-Core P-States allows more fine-grained control of voltages and frequencies, which enables performance and energy efficiency decisions at core(s) levels for energy-aware runtime services and operation system. The algorithm running in controller uses $\boldsymbol{E_M}$ ($\boldsymbol{E_M = M_{SLI} - R_{SLO}}$) to calculate the targeted frequency for specified core(s), and drive core operation frequency adjustment to target value through Per-Core P-States interface. Currently, the core frequency point and operation stepping that can be adjusted are limited to the specification definition for processor SKUs, and most of Intel Xeon processor can support frequency stepping at 100MHz.

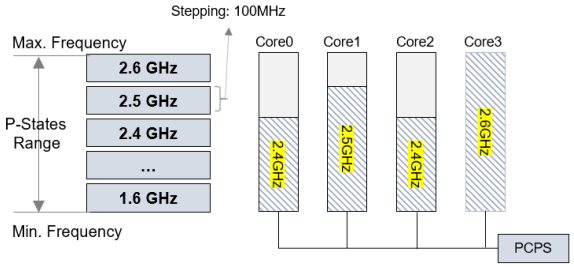

Figure 9 Per-Core P-States Control

The latency, from a general point view, is a time delay between the cause and the effect of one change in system. The control latency in SLO-AEEM consists of three portions: SLI measurement latency, system process latency and actuator response time. The SLI metrics adopted in this system is tail-latency calculated by service mesh across a period, and usually it is hundred-million second level. The system process latency covers the time for communicating SLI metrics to controller and the time for controller to calculate target frequency, and this latency is also at hundred-million second level. The actuator here is the Per-Core P-States control logic in CPU processor, its response time depends on the processor's P-State transition latency. The P-State transition latency - the time that PCU (power control unit) takes to change the frequency of a core – is at microsecond(us) level, e.g., 10us in 3rd Generation Intel Xeon scalable processor [10], and its impact to the control loop latency with million seconds-level can be neglected.

The frequent switch of core frequency causes system unstable. In control loop system, the hysteresis controller is the control logic to insert a response lag to filter signals so that the output of state change reacts less rapidly. The core

frequency is operated in either of two states: up or down. For a specific tail latency at time T (TL_T), the hysteresis band is defined as $Band_T$ ($Band_T = 10\% \ of \ TL_T$), and if the distance delta to TL_T is above the band, the core is operated in up-state, if it is below the band, the core is operated in down-state, and if the delta is within the band, the core operating is left unchanged. The hysteresis logic can help reduce unnecessary operation states switching.

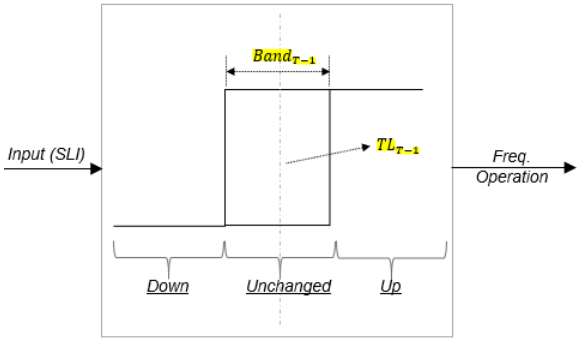

Figure 10 Hysteresis Logic for Core Freq. Operation

IV. EVALUATION

In this section, we defined the methodology to evaluate SLO-AEEM benefits and built a lab experimental cluster to do real workload evaluation. The evaluation proved SLO adaptive core power scaling can achieve obvious energy efficiency without compromising service quality goals.

A. Experimental Environments

The software environment used for evaluation is based on microservices framework deployed in a lab cluster with three computing nodes. Each computing node is 2 sockets system with 24-cores Intel Xeon processor, and P1 frequency is 2.1GHz, detail configuration shown in Table 1.

Hardware Configuration	
CPU	Intel(R) Xeon(R) Gold 5318Y CPU@ 2.10GHz
Socket(s)	2
Core per socket	24
TDP	165 W
L1d cache	48K
L1i cache	32K
L2 cache	1280K
L3 cache	36864K
OS	CentOS Linux release 8.4.2105
Kernel	4.18.0-305.3.1.el8.x86_64
Memory	16X16 GB 2666 MHZ DDR4
Hard Drive	1.8T SSD
Network Bandwidth	10Gb/s

Table 1 Hardware Configuration

The workload tool used for this evaluation is one open-source benchmark tool-suite - microservices-framework-benchmark. Microservices-framework-benchmark is a benchmark tool to run various benchmarks on throughput, latency for microservices framework [11], which contains several popular workload patterns, e.g., http-services, spring-boot etc. More specially for this this evaluation, we select go-lang http-server as server-side service. Go-lang http-server is a web-service based workload. The client-side tool is WRK, a modern HTTP benchmarking tool capable of generating various loading [12] for hundreds or thousands in-parallel http requests to web service. The WRK tool can support flexible configuration, like duration, thread number, and connection number, to simulate usage scenarios. This tool can also output statistics report, covering throughput, tail latency etc. To avoid mutual interference, the evaluation configures WRK tool and microservices-framework-benchmark to run in different computing node.

B. Energy Efficiency Evaluation

To characterize Go-lang http-server energy-performance relevance, we run Go-lang http-server with two different core frequencies - 1.6GHz and 2.1GHz. Table 2 is the evaluation result, obviously, lower core frequency brings longer service latency while less energy consumption. There exists energy saving opportunities as long as the tail latency fall into acceptable margin for specific SLO requirement. Using Table 2 as the example, if SLO is defined as >100ms, we'll get ~15% power saving while enforcing core frequency at 1.6GHz and won't compromise service quality target.

CPU Power (Watts)	Freq. (MHz)	P90 (MS)	Throughput (Requests/Second)	Traffic (MB/ Second)
109.45	1600	87	677590.2	83.36
109.75	1600	87	665131.71	81.83
109.32	1600	85	673745.4	82.89
125.36	2100	63	819806.38	100.86
125.15	2100	62	811290.25	99.81
125.57	2100	65	845382.52	104.00

Table 2 Go-lang Http-Server Energy Characteristics

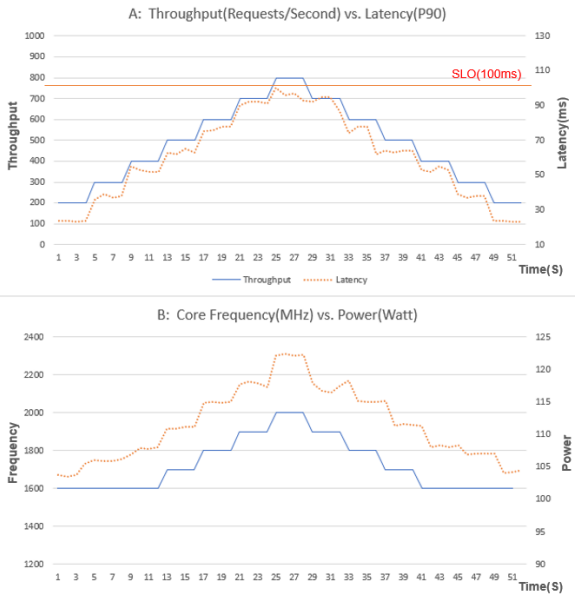

Figure 11 Evaluation Results

SLO-AEEM adopted closed-loop feedback system to dynamically scale core frequency in accordance with SLO requirements. It periodically polls service latency variation and feeds the delta value into controller to calculate the proper frequency for corresponding processing core(s). Figure 11.A depicts the correlation between service

throughput and core frequency. The throughput is measured via requests per second, and while the throughput is going down, the delta between service latency and targeted SLO ($\boldsymbol{E_M = M_{SLI} - R_{SLO}}$) drive the actuator (Per-Core P-States) to lower down core working frequency to certain level for better energy efficiency. In Figure 11.B, the dot line indicates dynamic core frequency setting in accordance with runtime throughput without comprising SLO requirement, the solid line is the real-time energy consumption. Figure 12 shows energy saving result, and the upper diagram is energy saving ratio while applying SLO adaptive power scaling algorithm.

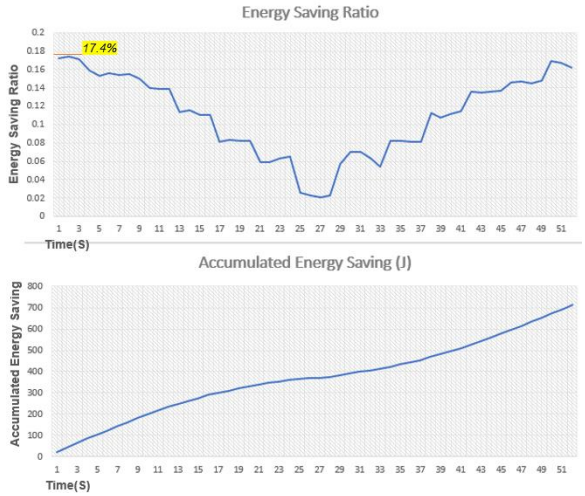

Figure 12 Energy Saving Result

V. CONCLUSION

This paper describes the technique for using service level objective, e.g., tail latency, to govern core frequency adjustment for microservices. A prototype implementation SLO-AEEM has been presented. The method introduced in this paper can scale processor core energy in line with service quality requests. The Per-core P-states (PCPS) is adopted to enable power scaling at core granularity instead of traditional per-socket frequency dynamic control to align with power management requirements from small microservice. As demonstrated via experiments and evaluation of the protype, the proposed method can effectively reduce energy cost in microservice deployment without compromising service quality goals.

The future work would consider more diversified service level indicators and fine-grained power management capabilities, including expanding service level indicators to memory bandwidth requirements as well as adding memory frequency and uncore frequency into power management knobs.

ACKNOWLEDGMENT

The authors would like to thank Meiqi Qiu for help with running the evaluation. The authors are thankful to the reviewers for their careful comments and suggestions.

REFERENCES

[1] G. Yu, P. Chen, Z. Zheng, "Microscaler: Cost-effective Scaling for Microservice Applications in the Cloud with an Online Learning Approach"

[2] J. P. K.S. Nunes, T. Bianchi, A. Y.Iwazaki, E. Yumi Nakagawa, "State of the Art on Microservices Autoscaling: An Overview"

[3] W. Li, Y. Lemieux, Z. Zhao, Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities"

[4] Buoyant Inc., "A Service Mesh for Kubernetes"

[5] A. Koschel, M. Bertram, R. Bischof, K. Schulze, M. Schaaf, I. Astrova,"A Look at Service Meshes"

[6] D. GhatrehSamani, C. Denninnart, J. Bacik, M. A. Salehi, "The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms"

[7] J. Lee, E. Lee, "Concerto: Dynamic Processor Scaling for Distributed Data Systems with Replication"

[8] Intel Corporation Enhanced SpeedStep® Technology for the Intel ® Pentium ® M Processor White Paper, March 2004.

[9] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, D. Hackenberg, "Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance"

[10] K. Devey, D. Hunt, C. MacNamara, "Power Management - Technology Overview"

[11] Microservices-framework-benchmark https://github.com/networknt/microservices-framework-benchmark

[12] WRK - https://github.com/wg/wrk