SLO-Power: SLO and Power-aware Elastic Scaling for Web Services

Mehmet Savasci
University of Massachusetts Amherst
Amherst, USA
msavasci@cs.umass.edu

Abel Souza
University of Massachusetts Amherst
Amherst, USA
asouza@cs.umass.edu

Li Wu
University of Massachusetts Amherst
Amherst, USA
liwu@cs.umass.edu

David Irwin
University of Massachusetts Amherst
Amherst, USA
irwin@ecs.umass.edu

Ahmed Ali-Eldin
Chalmers University of Technology
Göteborg, Sweden
ahmed.hassan@chalmers.se

Prashant Shenoy
University of Massachusetts Amherst
Amherst, USA
shenoy@cs.umass.edu

***Abstract*—Managing the performance of online web services in cloud data centers while optimizing resource allocation and power consumption is a multifaceted challenge. Often, resource and power management techniques, such as elastic scaling and power capping, are handled independently, leading to conflicts and sub-optimal power-performance trade-offs. To tackle this issue, we introduce SLO-Power, a system that coordinates the resource and power scaling techniques to achieve power savings while adhering to service level objectives (SLOs), such as tail latency constraints. Our approach employs a combination of analytic queuing models and feedback-driven techniques to jointly allocate resources and power to cloud applications in an SLO and power-aware manner. We implement a prototype of our system and evaluate it using realistic workloads to demonstrate its ability to harmonize elastic and power scaling, enabling enhanced resource utilization and reduced power consumption while ensuring the application performance. Our findings indicate that SLO-Power achieves exceptional power and resource efficiency, approaching near-optimal power-efficiency levels at 90%, all while preventing SLO violations. Furthermore, compared to state-of-the-art solutions, SLO-Power demonstrates lower P95 latency, accompanied by a 12% reduction in resource usage.**

***Index Terms*—Elastic Scaling, Power Capping, Web Services, Application SLO.**

I. Introduction

Today's cloud platforms are a popular choice for hosting online web services in domains such as banking, news and entertainment, e-commerce, and social media web applications. Modern web services tend to see dynamic workloads that exhibit variations at multiple time scales [1], making them well suited for cloud platforms that support on-demand allocation of resources to match observed workload dynamics. One popular approach for handling dynamic web workloads is to employ elastic scaling, where the resources allocated to the web service are varied dynamically to match the workload variations [2]. Many cloud platforms have built-in support for elastic scaling, making it straightforward for web applications to employ such functionality [3]–[5]. By its very nature, elastic scaling methods are designed to *overprovision* resources for two reasons. First, since end-users are highly sensitive to the tail latency of web requests [6], web services have strict service level objectives (SLOs). Inadequate resource provisioning can introduce bottlenecks and hurt tail latencies seen by users.

Second, elastic scaling techniques target handling the peak workload seen within each provisioning interval to reduce SLO violations and avoid frequent provisioning decisions. The need to handle peak workloads and reduce tail latencies require overprovisioning the resources allocated to an application. However, resource overprovisioning also increases the energy footprint of the application. This is because the application will typically see a lower ("average") workload intensity than the provisioned peak, which underutilizes the allocated resources and wastes power most of the time since the peak is not often reached. Since computing resources are translated into power consumption, this peak-provisioning also results in overprovisioning of power resources. Thus, elastic scaling methods are known to be SLO-aware but *power-unaware*.

The growth of web services has led to unabated growth of cloud platforms over the past decade, which, together with the networks complementing these infrastructures, now consume 3% of the world's electricity [7]. As such, since the demand continues to increase, today's massive cloud data centers are raising environmental concerns about their energy and carbon footprint. Consequently, reducing the power and energy footprint of cloud workloads has emerged as an important topic for today's cloud operations. There has been a wealth of research on optimizing the power usage of servers and data centers. This includes exploring various hardware mechanisms such as running average power limit (RAPL) [8] and dynamic voltage and frequency scaling (DVFS) [9], as well as software techniques, such as power-aware scheduling [10] and cluster-level power management [11]. When used judiciously, such techniques can reduce the power usage of the underlying servers [12]. However, recent research has also shown that aggressive power optimization can increase the risk of higher tail latency and hurt application performance since power optimization techniques are typically *SLO-unaware* and workloads tend to be very sensitive in dynamic changes in power [8].

Thus, in today's systems, resource and power management techniques, such as elastic scaling and power capping, operate independently of one another, leading to sub-optimal power-performance tradeoffs. To address this issue, we propose coordinating the elastic scaling and power capping techniques for web workloads so that the resource and power allocation decisions are both SLO- and power-aware. We hypothesize that it is feasible to design appropriate power management in careful coordination with elastic scaling techniques to achieve more aggressive power savings without hurting tail latencies or other SLOs. We design `SLO-Power`, a resource and power management system for latency-critical web applications that can meet tail latency SLOs while also reducing the application's power consumption. A key idea of `SLO-Power` is that it is not necessary to run overprovisioned resources at power levels that target peak performance to meet SLO objectives and that "slower" power settings can still achieve the desired performance while reducing the power footprint of the application. To sum up, we make the following contributions.

- We showcase the empirical effects of overprovisioning in traditional auto-scaling designed for web services, followed by an exploration of the latency impacts associated with aggressive power optimization (Section II).
- We introduce `SLO-Power`'s design that combines elastic scaling with power capping to optimize performance and power usage for web applications jointly. Our approach leverages analytic queuing models with feedback-driven techniques to jointly allocate resources and power to web applications in an SLO and power-aware manner (Section III).
- We implement a prototype of `SLO-Power` (Section IV) and evaluate it using realistic applications and workloads. Our results show that power and resource efficiency can reach near-optimal levels of 90% while avoiding SLO violations (Section V).
- In addition, compared to state-of-the-art solutions, `SLO-Power` achieves lower P95 latency while utilizing 12% less resources (Section V).
- We open-source `SLO-Power` for reproducibility[1].

II. Background and Motivation

This section provides background on latency-critical web services, elastic scaling techniques, power capping techniques, and the motivation of our system.

A. Latency-Critical Web Services

We aim to enhance the Quality-of-Service (QoS) of distributed web services, particularly those that are sensitive to latency, within cloud data centers. Research has shown that achieving low average or median latency alone is insufficient to ensure user satisfaction. To better improve the user experience, it is imperative to attain low tail (e.g., 95th or 99th percentile) latencies [13], [14].

One of the primary contributors to undesirable tail latency is the substantial variability in workloads over time and across various services. In particular, different web services typically exhibit varying response times, ranging from a few milliseconds to several seconds. This divergence is primarily attributed to variations in the design and development of these services, resulting in vastly different achievable response times. Additionally, the multi-tenancy aspects of the cloud have several impacts on the underlying infrastructure where workloads run, causing high-performance variability across applications. This variability across different web services introduces complexities into the management of distributed systems, as it becomes challenging to determine the attainable tail latency for each application precisely and to devise strategies for guaranteeing such latency. As a result, data center servers typically sacrifice server efficiency (having utilization of 5 – 30%) to maintain tail-latency targets resulting in wasting billions of dollars in equipment and terawatt-hours of energy annually [15].

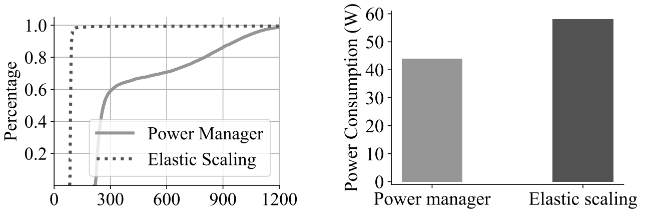
Additional sources of prolonged tail latency stem from CPU power-saving mechanisms in cloud data centers. Specifically, at low levels of resource utilization, power-saving techniques such as idle power states [16] and frequency scaling can exacerbate the tail latency, leading to elevated response time values. Thus, when the system operates at low utilization levels, a trade-off arises between power saving and tail latency [17].

B. Elastic Scaling

Unlike average latency, tail latency is more sensitive to changes in usage load and traffic patterns, system configurations, and resource availability [6]. Because of the sensitivity of tail latency to application and system factors, it is critical to design and manage systems for these latency-critical services efficiently.

Elastic scaling is one of the approaches that dynamically adjusts software configurations and hardware resource provisioning at runtime to adapt to time-varying conditions such as fluctuations in service workload [4]. The main goal of elastic scaling is to prevent both over- and under-provisioning of computing resources while meeting the QoS requirements (e.g., tail latency). To achieve that goal, several factors related to elastic scaling need to be considered. One critical factor is the elastic scaling policies for adding or removing resources, which can be conditions based on observed metrics, such as average CPU utilization, average memory utilization, workload, or any other custom metrics. Another essential consideration is the scaling step size, representing the number of resources configured during each provisioning process to accommodate workload fluctuations. In general, there are two types of scaling: horizontal scaling and vertical scaling. For horizontal scaling, which involves adding more instances or nodes to a system, the elastic scaler decides the number of instances of nodes to adjust; for vertical scaling, which involves adding more resources (i.e., CPU, RAM, storage, etc.) to an existing service instance or node, the elastic scaler decides the number of the corresponding resource.

[1] The code available at: https://github.com/umassos/SLO-Power



(a) Response time (ms) (b) Power consumption (W)

Fig. 1. The response time and power consumption of power manager (power-saving mode) and elastic scaling.

C. Power Capping

Power efficiency is a major concern in operating cloud data centers, which affects the operation costs and profoundly impacts on the environment [18]. Recent studies show that the power consumption of data centers will double across five European countries [19] and effective power management techniques could yield savings of over 25 billion kWh in the US [20].

In order to decrease the adverse effects of high power consumption, server power capping emerges as a solution to limit the power consumption of a server not to exceed a specific power budget. This solution allows data center operators to reduce the peak power consumption at the cost of performance degradation of hosted applications. Traditionally, DVFS was used to reduce CPU power consumption by decreasing the voltage or frequency. More recently, RAPL [21] has been proposed as an alternative that enables direct control over the power consumption of CPU and memory of a single server.

RAPL is a management interface provided by Intel processors that combines automatic DVFS and clock throttling. It improves the plain DVFS in two major aspects. First, it integrates power monitoring and control inside the chip, making it more accurate and faster to identify and adapt to workload fluctuations. Second, it combines DVFS and clock throttling, providing power levels than DVFS alone and, therefore, finer management granularity [22]. Hence, we employ RAPL to implement the dynamic power allocation in our `SLO-Power`.

D. Motivation

The main insight that motivates our work is that the elastic resource provisioning mechanism prioritizes SLOs, leading to resource over-provisioning and excessive power consumption. Conversely, the sophisticated power management in cloud data centers focuses on power efficiency but lacks SLO awareness. This dichotomy presents an opportunity to create a system that is both SLO-aware and power-efficient, ultimately enabling the cloud service provider to meet SLOs while reducing power consumption.

Figure 1 plots the response time achieved by two different approaches: power manager and elastic scaling. In our experiments, we employed the Linux kernel's *CPUFreq* (CPU Frequency scaling) subsystem, enabling dynamic CPU frequency and voltage adjustments for power management. Specifically, we set the scaling governor to *powersave*, forcing the CPU to operate at its minimum frequency to optimize power consumption. For the elastic scaling approach, we dynamically allocate the number of CPU cores to the web service based on incoming requests. As illustrated in Figure 1(a), the elastic scaling approach demonstrates excellent performance, achieving an average response time of 91ms. In contrast, the power manager exhibits approximately four times the response time compared to the elastic scaling approach. This is because the power manager lowers the CPU speed, and it takes longer to process the requests. Furthermore, we compare the power consumption of these two approaches. Figure 1(b) depicts that the elastic scaling has an average power consumption of 58.1 Watts, which is 32.2% higher than the power manager's consumption (43.96 Watts on average). This is because the elastic scaling tends to provision more CPU cores to meet the SLOs, resulting in increased power consumption. These results show that the elastic scaling approach is SLO-aware but power-unaware, while the power manager is power-aware but SLO-unaware.

Takeaway. When running web services in the cloud data center, using either power management or elastic scaling alone can only achieve one goal of meeting SLOs and saving power. This motivates us to exploit a hybrid approach that combines power management with elastic scaling to not only meet the service performance requirements but also reduce power consumption.

Challenges. When combining the power manager with elastic scaling in practice, major challenges are from the interference between them. *1) conflict objectives:* Since the objective of power manager and elastic scaling have conflict, focusing on saving power and guaranteeing SLOs, respectively, can lead to suboptimal resource utilization and potentially affect the overall system performance when both mechanisms are active. For example, the power manager might reduce the frequency and voltage of CPU cores to save power, but elastic scaling may want to allocate more CPU cores to meet high demand; *2) uncertain CPU performance:* The dynamic adjustments of CPU frequency and voltage can affect the performance of CPU cores, making it challenging for elastic scaling to accurately estimate how many cores are needed to maintain the SLOs. This uncertainty can result in over-provisioning or under-provisioning of resources; *3) Fluctuating response time:* When elastic scaling adjusts the number of CPU cores, it may take some time for the system to stabilize. During this transition period, the CPU frequency and voltage changes may not align with the newly allocated cores, causing fluctuations in response time. These response time fluctuations can impact the QoS, especially for web applications with strict tail latency constraints.

III. SLO-Power Design

In this section, we outline the design of `SLO-Power` and present the details of key components.

A. System Overview

`SLO-Power` is an SLO and power-aware elastic scaling system tailored for latency-sensitive web services in the cloud.

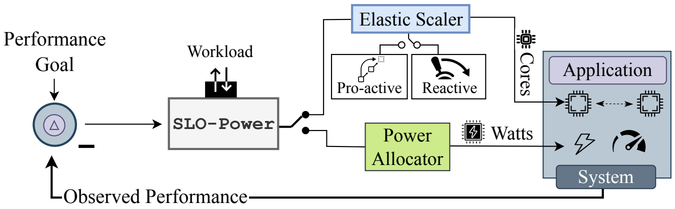


Fig. 2. `SLO-Power` system design: dynamic mechanism controls system's resources and power through the Elastic Scaler (blue) and the Power Allocator (green) units.

It incorporates elastic scaling and power capping techniques to achieve two goals: the first is to ensure performance guarantees (e.g., tail latencies) to applications; the second is to effectively reduce server-level power consumption, including CPU cores and CPU power.

Figure 2 shows the `SLO-Power` system design. It comprises three components: two actuators – the Elastic Scaler (blue) and the Power Allocator (green) –, along with a Control Unit (light gray), the entity used to dynamically decide when to trigger the actuators and which one to use. To address the challenge of conflict objectives due to interference between elastic scaling and power management, the Control Unit always ensures the SLOs are guaranteed first, then adjusts resource or power allocation to save power. Additionally, the Control Unit orchestrates the interplay between the Elastic Scaler and Power Allocator, leveraging local agents to establish power budgets and allocate CPU cores on individual servers through RAPL and cgroup [23] quotas. Regarding the two actuators, the Elastic Scaler's responsibility lies in determining the number of cores for the application based on its workload. It leverages vertical scaling to dynamically fine-tune resource allocation, subsequently impacting performance, as measured by response times and resource utilization. Meanwhile, the Power Allocator allocates the necessary power by considering the current power consumption and target SLOs. Notably, to avoid the uncertainty in CPU performance and fluctuations in response times caused by degraded power levels, the Power Allocator adjusts the power budget accordingly when the number of cores is changed. In addition, in `SLO-Power`, the Power Allocator is designed to operate continuously, while the Elastic Scaler is invoked when further enhancements in response times cannot be achieved with power allocation and the existing number of cores.

B. Control Unit

In our `SLO-Power`, the Control Unit is designed to decide when to trigger the resource and power allocation based on observed metrics. Once allocation decisions are made by Elastic Scaler or Power Allocator, it is also responsible for enforcing the resource and power changes to the system via cgroup quota and RAPL.

Figure 3 shows how the Control Unit selects the actuators. By observing the real-time response times (i.e., 95th percentile latency) of the web service, it compares it with a predefined threshold and makes the scale-up/down decisions.

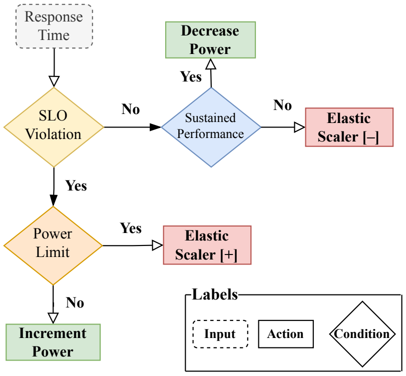


Fig. 3. `SLO-Power` resource and power allocation flow.

If the response time is above the threshold, the scale-up action on power and/or resources is needed. Next, the Control Unit checks if the current allocated power is at maximum levels of the given number of cores. If yes, it indicates only a change in resource allocation is possible to mitigate the SLO violation. In that case, the Elastic Scaler is triggered, where a scale-up decision is taken; otherwise, the Power Allocator is triggered to allocate more power immediately. In our `SLO-Power`, the threshold for SLO is defined on a guarded response time using parameter α, which is proportional to the target response time T. If the observed response time is above the guarded response time, denoted as $latency > \alpha \cdot T$, it triggers the scale-up actions. Additionally, to avoid SLO violations, the Control Unit also triggers Elastic Scaler to proactively allocate more resources when the resource utilization is above a user-defined threshold $\beta \in (0, 1)$.

Similarly, when sustainable performance levels are maintained for a user-defined duration D, during which both the observed response time and resource utilization are under the defined thresholds, the Control Unit triggers the Elastic Scaler to scale down the number of cores for resource savings and power conservation. To save more power, during the duration of D, the Control Unit triggers the Power allocator to decrease the power budget consecutively.

To summarize, the Control Unit makes the decision based on the observed response times, resource utilization, and power consumption, following the principles of *fast reaction to SLO violations and slow reaction to save power, and power adjustment prioritizes resource changes.*

C. Elastic Scaler

When the Elastic Scaler is selected by the Control Unit, it is required to adjust the resources allocated to the server to accommodate the SLOs. To achieve this, an appropriate amount of resources needs to be decided based on the amount of time resources take to process requests. A well-established technique for doing that involves leveraging queueing theory to model the processing of web applications [24]–[27]. This approach treats an application as a logical server encapsulated within a container or virtual machine (VM) and assigns a specific resource capacity to it. Depending on the resource management configurations, this provisioning may involve portions of CPU cores or even the dynamic scaling of multiple cores.

We design our Elastic Scaler as an $M/G/k/PS$ system, where the use of Processor Sharing (PS) queueing policy stems from our assumption that operating systems (OS) employ traditional CPU time-sharing methods to process requests within the server. Within this model, we utilize the following closed-form equation to estimate the expected response time, based on [28]. Let $E[R]$ denote the expected response time. Then,

$$E[R] = \frac{c}{\lambda} \cdot \frac{\rho}{1-\rho} = \frac{c}{c\mu - \lambda}, \tag{1}$$

where c is the number of allocated cores, λ and μ are the request arrival rate and service rate at the server, respectively. Moreover, $\rho = \frac{\lambda}{c\mu}$ is the system utilization and represents the fraction of time the server is busy. Equation 1 can be used to compute the resource allocation c based on a given target response time T, i.e., the SLO. That is, replacing $E[R] = T$ in Equation 1,

$$T \times (c\mu - \lambda) = c \Leftrightarrow c = \frac{T \times \lambda}{T \times \mu - 1} \tag{2}$$

Equation 2 depends primarily on the workload rate and on the relative capacity that one server core to handle requests. Finally, a queueing system is considered stable when the value of $\rho < 1$. When $\rho > 1$, it indicates insufficient server capacity, leading to unbounded or excessively large response times, violating all pre-defined SLOs, and causing performance degradation. This condition, referred to as *saturation*, is highly undesirable in distributed systems and should be prevented [29].

Internally, the Elastic Scaler has proactive and reactive auto-scalers to allocate resources.

Proactive Auto-scaler. The proactive auto-scaler of `SLO-Power` is designed to allocate resources on relatively large time scales, according to the predicted increase in workload. This approach leverages a workload estimator to calculate the expected demand for the upcoming period. By employing a workload estimator, our predictive component strives to allocate resources proactively, well before the demand arises. Our workload estimator is inspired by a technique that relies on historical workload data to forecast the demand using the best-fit model.

Since Equation 2 is designed to address the average request load, its allocations c are optimized for the typical web application usage and do not consider unexpected workload spikes, which can inadvertently cause several SLO violations. Moreover, the proactive auto-scaler may unnecessarily overallocate resources as it is based on a worst-case situation. In response to these limitations in the proactive mechanism, we propose a reactive auto-scaler integrated with it to quickly respond and allocate resources as needed.

Reactive Auto-scaler. The reactive auto-scaler functions within short time scales, typically ranging from seconds to minutes. It is designed to ensure that potential SLO violations are effectively mitigated and overprovisioned resources are reduced appropriately for power-saving purposes. Therefore, it quickly responds to any performance degradation stemming from underestimations by the proactive auto-scaler, promptly allocating additional cores to handle the increased workload. When the performance is sustained, it consecutively reduces the number of cores, aiming to save power without introducing SLO violations.

Algorithm 1: Reactive Auto-scaler

```
Input: curr_cores, curr_latency, curr_resource, $D$, $\delta$
Initialization: counter ← 0
 /* Scale-up                                      */
if SLO_violation then
    if power_hit_limit then
        curr_cores ← $c_{t+1}$ from Equation 3
        power_allocation(curr_cores)
    else
        power_allocation(curr_latency)
 /* Scale-down                                    */
else
    counter ← counter + 1
    if counter $\geq D$ & ($c_{t+1}$ - curr_cores) $\geq \delta$ then
        curr_cores ← curr_cores - $\delta$
        counter ← 0
power_allocation(curr_resource)
```

$$c_{t+1} = \frac{(\alpha \cdot T) \times \lambda_{t+1}}{(\alpha \cdot T) \times \mu - 1} \tag{3}$$

To decide the number of allocated CPU cores, reactive autoscaler uses the same queueing theory model as the proactive mechanism, but it estimates the workload of the next step λ_{t+1} online [30] and uses the guarded response time $\alpha \cdot T$ as average response time, as shown in Equation 3. When SLOs get violated, the reactive auto-scaler employs the number of cores c_{t+1} to adjust the resource allocation, thus mitigating the SLO violations; whereas for sustained performance, c_{t+1} is used to limit the resource-saving to avoid severe system performance degradation. Instead, a constant step size δ is defined for scaling down the resources.

Furthermore, we summarize the resource allocation of reactive auto-scaler in Algorithm 1. In the algorithm, we can see that the Power Allocator is constantly involved in adjusting the power allocation in response to the core changes and the scale-up/down decisions. In the next section, we will introduce how the Power Allocator decides the requisite power budgets.

D. Power Allocator

In `SLO-Power`, we prioritize the Power Allocator (Figure 2, green) actions over resource allocation ones. This is because: first, changes in power settings result in less performance degradation than changes in resource allocation,

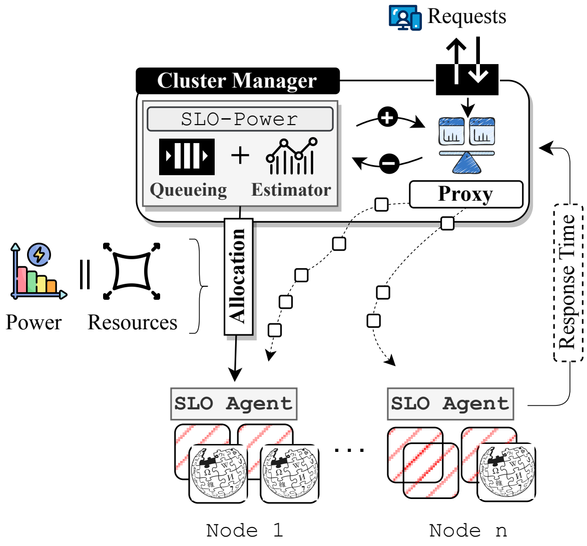


Fig. 4. SLO-Power implementation. Striped squares mean non-allocated resources.

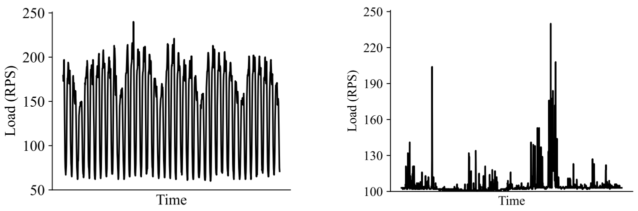
which may trigger OS resource reconfiguration due to changes in running threads. Second, power settings take effect over a millisecond, faster than the resource allocation that requires several milliseconds to take effect, often due to overheads in the OS kernel. To react quickly to potential SLO violations, the Power Allocator operates continuously and in a *lower* granularity than the Elastic Scaler. Changes in resource allocations should be preferred only when changes in power are deemed insufficient or not possible to satisfy web service SLOs. As such, the Power Allocator is designed as the complimentary mechanism for elastic scaling.

$$p_{t+1} = \begin{cases} p_t + \eta \times (c_{t+1} - c_t), & \text{if core changes,} \\ p_t + |\frac{T - T_t}{T_t}| \times p_t, & \text{if scale-up,} \\ p_t - \frac{\eta}{D}, & \text{if scale-down.} \end{cases} \tag{4}$$

As shown in Algorithm 1, power allocation is triggered during core changes (Line 5), scale-up (Line 7), and scale-down (Line 13) decisions. For core changes, the Power Allocator adjusts the power budget based on the difference of core number $c_{t+1} - c_t$. For each newly added core, it allocates the maximum power limit per core η to it, resulting in a power change of $\eta \times (c_{t+1} - c_t)$. For the scale-up decision, the newly allocated power is proportional to the current power p_t, and the proportionality is determined by the response time T_t deviation from the target T, denoted as $|\frac{T-T_t}{T_t}|$. For the scale-down decision, the power budget is determined by the duration D which is used to trigger a core change after consecutive power reduction. The step size is $\frac{\eta}{D}$. Furthermore, we summarize the power allocation in Equation 4.

IV. Implementation

We implement SLO-Power in a cluster setting, as shown in Figure 4, where it works as a component of a resource and cluster manager along with a proxy server – e.g., the load-balancer or DNS server – directing process requests and which can be used for monitoring, e.g., incoming request rates, throughput, and response time results. While our SLO and power-aware elastic scaling is broadly applicable to any cluster manager, including Kubernetes, our prototype is an extension of LXD [31]. LXD is a cluster and container management system built on top of LXC, the Linux container runtime system. We selected LXD for its versatility and compatibility with stateful applications, as well as its support for vertical resource scaling. LXD facilitates the provisioning of complete operating systems within containers, similar to lightweight virtual machines. It allows vertical scaling of each container's resources through cgroups and incorporates a virtual filesystem (LXCFS) mounted over $/proc$, ensuring precise resource accounting, view, and control for each application instance.



(a) Wikipedia trace (b) Azure trace

Fig. 5. Workload traces for evaluation.

In the implementation, our SLO-Power possesses privileged access to the underlying hardware, allowing it control over the system components regulating the CPU power modes and the cluster for monitoring and resource allocation. In particular, the changes in power and resources are realized through daemon agents running at each host that commits SLO-Power's actions, either through cgroup quota or through the RAPL interface. In total, SLO-Power is implemented in around 1.1K LoC in Python.

V. Experimental Setup and Evaluation

In this section, we first describe the setup for our experiments, including the real-world application and workload traces. Next, we evaluate our SLO-Power, together with its components against performance, power saving, and resource efficiency, and analyze its sensitivity towards various adjustable parameters. Lastly, we compare SLO-Power against three baseline methods.

A. Experimental Setup

Hardware and Cluster Setup. The experiments are conducted on a cluster of Dell PowerEdge R440 servers running Ubuntu 20.04 LTS. Each server has a two-socket Intel Xeon Silver CPU, 8 cores each, 2.10GHz, and 64GB of memory. We disable hyperthreading and turbo boost since they affect the comparisons, regardless of the approach used. To emulate a standard cloud environment and facilitate vertical scalability across servers, we have opted for the LXD cluster manager. An application instance is deployed on each physical machine. Each instance of the tested application is set with all the necessary components, including web servers, database servers, or microservices.

Benchmark Application. We use a latency-sensitive cloud application: MediaWiki [32], which hosts a replica of Wikipedia. MediaWiki is a custom-made, free, and open-source wiki software platform written in PHP using a traditional LAMP stack software, running on Linux. Along with the Apache web server, MediaWiki deploys a MySQL database to host wiki articles, and a Memcached database that provides low-latency reads of recently accessed objects in memory. Specifically, we use the pre-built version of the German Wikipedia, which comprises a total of 10 GB of content. We use the HA-Proxy as a load balancer for the application and also use it to collect application arrival rate and response time data. Requests are sent through a load-balancer, which exposes an external HTTP port interconnecting all nodes in the cluster. We set the load balancer to send requests to the nodes in a round robin way.

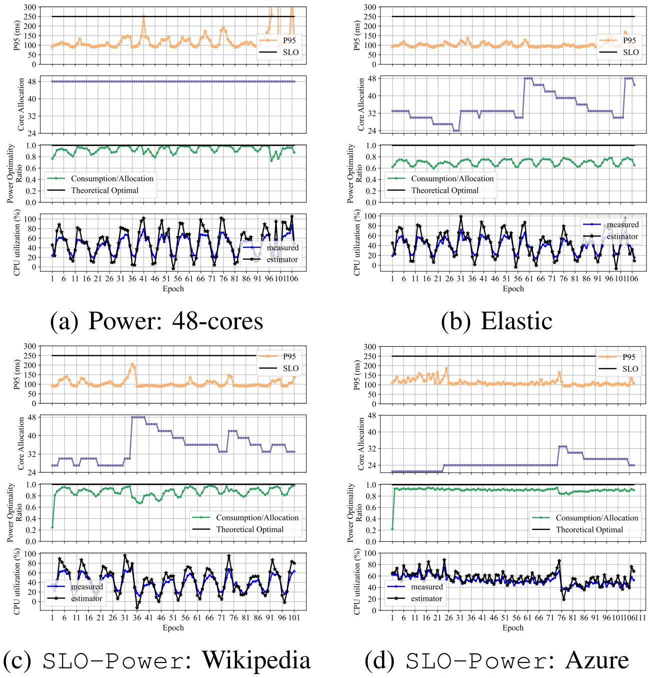
Workloads. In our experimental setup, we use two realistic workloads: Wikipedia [33] and Azure traces[2] shown in Figure 5. To avoid saturating the server, the traces are scaled to our cluster size. When submitting the cluster to these workloads, requests are issued using an open-loop system model [34], which creates requests both pre-determinedly and randomly by not waiting for the system's response, i.e., a new request is generated regardless if a response to previous requests is received. To emulate users accessing applications, we use the httpmon tool[3], which supports the open-loop system model behavior. The number of users varies following each workload trace.

Telemetry. The response time of a request is characterized as the time length from the moment the request is dispatched to the moment the request output response is received. Here, we focus on the processing time of the requests by CPU. We collect the average and 95th percentile (P95) response times at every 1 second.

B. Workload Validation and Component Analysis

In our initial sets of experiments, we seek to validate and demonstrate both the elastic scaler and the power allocator, our `SLO-Power`' components. They are tested both in isolation, as well as integrated with one another in a unit. For these experiments, we utilize the Wikipedia and Azure workloads, as depicted in Figure 5, under a cluster capacity of 48 cores, with the maximum power allocation for the cluster set 276 Watts, and with a SLO target configuration set at 250ms. We report the P95, the resource allocation, and utilization. In addition, we also report the power ratio, which is defined as the ratio between power consumption over power allocation. Theoretically, if a power allocation is optimally configured, the power ratio is near the value of 1.0.

Power Component. The power component strives to slowly reduce the power cap allocated to the processor, effectively reducing its processing power and affecting the application's overall response time. However, besides mitigating the side-effects of resource reconfiguration due to core changes – e.g., thread recreations –, the power component allows the `SLO-Power`'s control unit to react to SLO violations. Figure 6(a) presents results for the power component analysis. In this run, the cluster size is set to a maximum of 48 cores. It can be seen that the power ratio averages 0.91, with 189 Watts of power consumption and 203 Watts of power allocation. However, although nearing the theoretical optimal, the power allocator mechanism incurs 5% SLO violations, negatively affecting the P95 latency, even though the setting allots the full cluster size capacity. The reason for these violations is mainly due to the under-power allocation that the power-saving technique enforces, resulting in an overall processing power that is limited and lower than what the application *physically* requires (i.e., in terms of compute-power).



(a) Power: 48-cores

(b) Elastic

(c) `SLO-Power`: Wikipedia

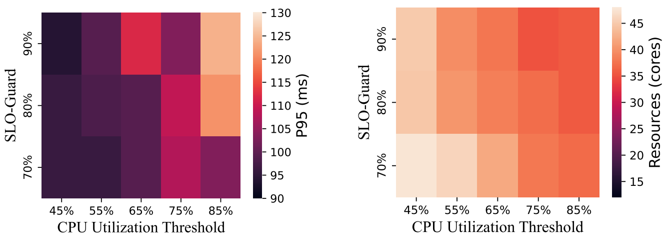
(d) `SLO-Power`: Azure

Fig. 6. Power × Elastic × `SLO-Power` Mechanisms (Wikipedia/Azure): Power and resource allocations are regulated while controlling latency. Resource allocation (2nd subplot) is utilized to control latency (1st subplot), up until the estimator (lower subplot) detects workload peaks to avoid SLO violations.

Elastic Component. The elastic auto-scaler operates over a hybrid time scale, using both the proactive and reactive mechanisms in conjunction. It requires accurate workload rate estimations of the immediate upcoming time windows to effectively adjust the resource allocation and avert SLO violations (Section III). To assess its effectiveness, we subject `SLO-Power` to the Wikipedia workload, configuring it not to use the power allocator component. The elastic component analysis results are depicted in Figure 6(b). Notably, the proactive controller consistently optimizes resource availability while effectively managing the P95 latency. Resource allocation exhibits fluctuations of up to 12 cores around the 35-core average, resulting in an average power usage of 193 Watts. However, the power ratio is only 70% as no capping is applied. Interestingly, despite substantial resource

[2] https://github.com/Azure/AzurePublicDataset

[3] https://github.com/cloud-control/httpmon



(a) Latency (b) Resource allocation

Fig. 7. SLO-Power Sensitivity Analysis: P95 latency decreases with stricter power-resource parameters, and more resources are allocated with smaller CPU utilization thresholds.

reduction, no P95 SLO violations are observed, ensuring the latency remains within the 250ms SLO target and achieves 96.64ms. This success is primarily attributed to the workload estimator (lower plot), which accurately identifies peaks in the request rate, facilitating the timely scaling of resources by the proactive mechanism. Despite its efficiency in resource management, the elastic mechanism attains only 39% of overall CPU utilization.

SLO-Power. Finally, in this experiment, we integrate both the Elastic and the Power mechanisms in one component. It is used as a heuristic to jointly combine both resource and power optimizations (Figure 3 and Algorithm 1). We also test it under two workloads: Wikipedia and Azure. Figures 6(c) and (d) depict results for SLO-Power under both workloads. Overall, in both cases, SLO-Power steers the P95 latency towards the SLO target due to both accurate resource estimations and power deallocation: our workload estimator achieves 0.16 and 0.05 normalized root-mean-square error (NRMSE) respectively for each workload. This results in a power-efficiency ratio of 86% and 90%, respectively, only 1 to 4% lower than the power mechanism. In addition, it has zero SLO violations, achieving a 117ms P95 latency, well below the SLO target. SLO-Power provides an effective mechanism to apply power-saving mechanisms jointly with resource scaling while respecting performance targets.

Key Takeaway. While achieving near-optimal power efficiency, power-capping mechanisms inadvertently degrade application performance. On the other hand, resource scaling achieves good performance but falls short on power efficiency. Jointly utilizing both mechanisms combines 86% power efficiency while meeting SLO targets.

C. Sensitivity Analysis

Next, we analyze how the SLO-guard and CPU resource utilization threshold defined in the Control Unit affects the performance of SLO-Power. We analyze the same influences when utilizing the SLO-safeguard mechanism by varying the SLO-Guard from 70 to 90%. In addition, we scrutinize the impacts on latency and resource efficiency while varying the CPU threshold from 45% to 85%.

Results. Figures 7 represent the overall results of our analysis. Figure 7(a) shows that lower CPU utilization thresholds yield lower P95 latencies. This is due to the higher sensitivity to peaks in resource usage, triggering the autoscaler to allocate more cores, which can be seen in Figure 7(b). It can also be seen that the effects of SLO-Guard are less noticeable with lower CPU thresholds, indicating a higher sensitivity of the workload on the CPU utilization. However, when the CPU threshold is high, like 85%, a tighter SLO-Guard results in a higher P95, which tends to have less resource allocation and less power consumption.

Key Takeaway. Indiscriminately allocating more resources may not necessarily enhance workload performance. Workloads tend to respond to changes in resource utilization, highlighting the need for meticulous configuration of elastic systems.

D. Different SLO Targets Comparisons

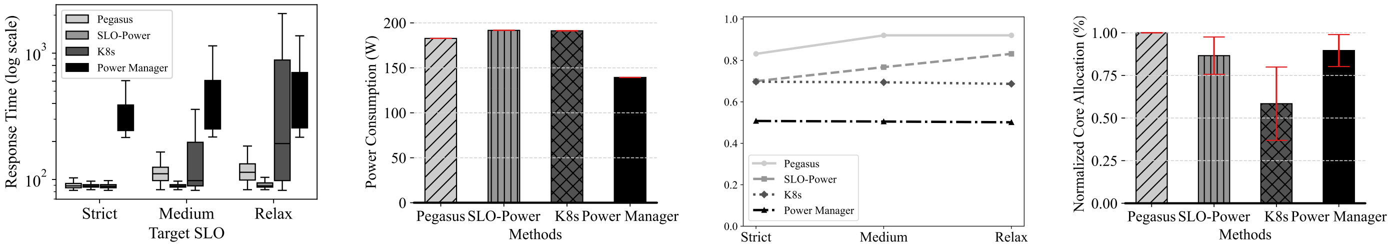
Here, we aim to evaluate how SLO-Power compares against modern auto-scalers under different performance targets. Based on service level agreements commonly used by cloud providers [35], we define three levels of SLOs: (i) strict, (ii) medium, and (iii) relax. The strict level has 150ms as the target and generally utilizes more power and compute resources to satisfy targets. In comparison, both medium (200ms target) and relax (250ms target) levels consume less power and compute. Unlike SLO-Power and Pegasus, the K8S and Power Manager strategies do not use SLOs. Thus, we use resource and power allocation parameters that reflect the target levels when submitting them to the Wikipedia workload. We report the P95 latency, power consumption and ratio, and resource allocation and efficiency for all experiments. Below, we detail the state-of-the-art baselines we compare against SLO-Power.

K8S Autoscaler. The K8S auto-scaler is based on the average CPU utilization across all cluster instances and allocates resources based on a given threshold that is application-dependant [36]. When autoscaling, it does not consider the response time being experienced by the application. As such, it is a SLO-unaware policy.

Power manager. The Power manager policy uses the pre-set best DVFS configuration to save power, affecting the application performance. It is a SLO-unaware approach that reduces power consumption based on the performance counters of the underlying hardware. We set different power caps according to following the target SLOs.

Pegasus. Pegasus [37] is a system that considers the SLO target to provision power resources to the application. It uses a controller alternating among several pre-defined heuristics, e.g., adjusting power to the maximum if the response time exceeds a certain threshold. Due to its dynamic and model-free nature, Pegasus can quickly react to changes in workload and handle SLO violations. We use the same SLO targets when comparing it against SLO-Power.

Results. Figures 8 presents the latency, power, and resource utilization results for all baselines. As shown in Figure 8(a), the higher availability of resources across all policies enables



(a) P95 response time (b) Power consumption (c) Power efficiency (d) Resource usage

Fig. 8. Evaluating SLO-Power with different target 95th percentile response times.

very low levels of P95 latency. The exception is for the power manager: due to the DVFS configuration, the power manager policy reduces too much of the CPU frequency, severely impacting the latency levels. As the SLO targets reduce, the impacts of using the workload trend to make decisions are more noticeable, negatively affecting all baselines but SLO-Power. Since Pegasus does not use a predictive mechanism, its reaction features do not timely control the latency.

Figure 8(b) shows the dynamic power consumption remains constant across all baselines, which is primarily influenced by resource utilization, which in turn, depends on the application runtime cycles. Since the application is the same across runs, power remains unchanged, with the power manager achieving lower consumption due to its CPU frequency management. Figure 8(c) shows that despite achieving 10% higher power efficiency ratios, Pegasus uses 12% more resources than SLO-Power (Figure 8(d)). Overall, Pegasus achieves up to 5% SLO violations, while SLO-Power achieves only 2% violations, even with stricter targets. This is because of SLO-Power's resource efficiency, which better controls both the P95 and the latency variability, resulting in more predictable performance.

Key Takeaway. While SLO and power-aware mechanisms achieve both high efficiency and performance, they come at increased resource consumption of at least 10%. Power-elastic mechanisms that incorporate resource-awareness demonstrate a 2× reduction in SLO violations while reducing resource usage by 12%.

VI. Related Work

In the literature, there is a wealth of research on elastic scaling and power management. In this section, we discuss the most related pieces of work and present the key differences involved in the design of our system.

Substantial prior research has concentrated on elastic resource provisioning [3], [5], [38]–[42], with different techniques, such as queueing theory, machine learning, and heuristics. Similar to our analytic queueing model, Sharma et al. employ a tandem M/G/1-PS model to characterize multi-tier applications, estimating response time distribution. These distributions are then utilized by a horizontal scaler to determine the number of servers required for provisioning [39]. However, our SLO-Power applies the queueing model to reactive auto-scaler as well to adapt to the spikes in workloads. Furthermore, hybrid auto-scalers combining proactive and reactive scalers have also been proposed. Ahmed Ali-Eldin et al. [24] propose using reactive scaling for scale-up and proactive scaling for scale-down. However, these approaches fail to integrate power management to save power while meeting tail latencies as our SLO-Power does.

Regarding power-performance management, various work have been proposed [8]–[12], [43]–[46]. Eric Rutten et al. benchmark the application offline and subsequently derive a system model. They leverage this model to develop a Proportional-Integral (PI) controller for making power allocation decisions [43]. Similarly, DDPC [8] employs a Proportional-Integral (PI) power controller to capture the application's power-latency trade-offs. However, this approach requires intensive profiling to get a power-performance model, which might fail to adapt to different workloads well and is designed to guarantee the average response time. However, our SLO-Power is model-free and can meet tail latency constraints.

VII. Conclusion

In this paper, we propose an SLO and power-aware elastic scaling system for online web services, SLO-Power. Our system is designed to reduce server-level power consumption while adhering to the application SLO targets – e.g., 95th percentile latency (P95). To achieve this goal, SLO-Power incorporates the elastic resource scaling and power capping techniques that dynamically allocate CPU cores and power caps in response to workload fluctuations and observed application response times. In particular, we combine analytical queueing models, workload estimators, and feedback-driven techniques for joint allocation decisions, with careful attention to pro-active allocations to address the challenges coming from the interference between them. At last, we prototype our system and evaluate it under real-world workload and applications, against state-of-the-art auto-scalers. Our findings indicate that SLO-Power can achieve power and resource efficiency close to optimal levels, reaching up to 90%, besides avoiding SLO violations. Moreover, compared to state-of-the-art solutions, SLO-Power attains lower, controllable P95 latency, all while utilizing 12% fewer resources. We open source SLO-Power[4]. One limitation of SLO-Power is that

[4] https://github.com/umassos/SLO-Power

it uses an Intel-specific hardware actuator, RAPL, to manage power allocation. In the case of servers with heterogeneity, relying on RAPL might limit the applicability of our power controller component. For this reason, in the future, we plan to develop hardware-independent software-based techniques for power management.

Acknowledgment

This research has been supported by the Republic of Türkiye Ministry of National Education YLSY program and NSF grants 2211888, 2213636, 2105494, and 2021693. We appreciate anonymous reviewers for their feedback.

References

[1] A. Kumar, I. Narayanan, T. Zhu, and A. Sivasubramaniam, "The fast and the frugal: Tail latency aware provisioning for coping with load variations," in *Proceedings of The Web Conference 2020*, 2020, pp. 314–326.

[2] M. Eriksen, K. Veeraraghavan, Y. Abdulghani, A. Birchall, P.-Y. Chou, R. Cornew, A. Kabiljo, M. Lieuw, J. Meza, S. Michelson *et al.*, "Global capacity management with flux," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 589–606.

[3] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[4] H. Qian, Q. Wen, L. Sun, J. Gu, Q. Niu, and Z. Tang, "Robustscaler: Qos-aware autoscaling for complex workloads," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 2762–2775.

[5] Z. Wang, S. Zhu, J. Li, W. Jiang, K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, and A. X. Liu, "Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 16–30.

[6] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[7] V. Rozite, E. Bertoli, and B. Reidenbach, "Data centres and data transmission networks," *IEA*, 2023. [Online]. Available: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks

[8] M. Savasci, A. Ali-Eldin, J. Eker, A. Robertsson, and P. Shenoy, "Ddpc: Automated data-driven power-performance controller design on-the-fly for latency-sensitive web services," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 3067–3076.

[9] L. Zhou, L. N. Bhuyan, and K. Ramakrishnan, "Gemini: Learning to manage cpu power for latency-critical search engines," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 637–349.

[10] A. G. Kumbhare, R. Azimi, I. Manousakis, A. Bonde, F. Frujeri, N. Mahalingam, P. A. Misra, S. A. Javadi, B. Schroeder, M. Fontoura *et al.*, "{Prediction-Based} power oversubscription in cloud platforms," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 473–487.

[11] P. Patel, E. Choukse, C. Zhang, Í. Goiri, B. Warrier, N. Mahalingam, and R. Bianchini, "Polca: Power oversubscription in llm cloud providers," *arXiv preprint arXiv:2308.12908*, 2023.

[12] S. Li, X. Wang, F. Kalim, X. Zhang, S. A. Jyothi, K. Grover, V. Kontorinis, N. Narodytska, O. Legunsen, S. Kodakara *et al.*, "Thunderbolt:{Throughput-Optimized},{Quality-of-Service-Aware} power capping at scale," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1241–1255.

[13] M. Brooker, "Tail latency might matter more than you think," https://brooker.co.za/blog/2021/04/19/latency.html, accessed: 2023-10-04.

[14] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, 2018.

[15] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[16] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 205–216, 2009.

[17] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[18] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis, "Power management of datacenter workloads using per-core power gating," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 48–51, 2009.

[19] BloombergNEF, "Data centers set to double their power demand in europe, could play critical role in enabling more renewable energy," https://about.bnef.com/blog/data-centers-set-to-double-their-power-demand-in-europe-could-play-critical-role-in-enabling-more-renewable-energy, accessed: 2023-10-04.

[20] A. Shehabi, S. J. Smith, E. Masanet, and J. Koomey, "Data center growth in the united states: decoupling the demand for services from electricity use," *Environmental Research Letters*, 2018.

[21] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, 2010, pp. 189–194.

[22] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 545–559, 2016.

[23] T. L. K. documentation, "Control groups," https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html, accessed: 2024-2-27.

[24] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 204–212.

[25] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, "Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 800–813, 2018.

[26] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2008.

[27] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE, 2007, pp. 27–27.

[28] Q. Liang, W. A. Hanafy, A. Ali-Eldin, and P. Shenoy, "Model-driven cluster resource management for ai workloads in edge clouds," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 18, no. 1, pp. 1–26, 2023.

[29] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu, "Metastable Failures in Distributed Systems," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 221–227.

[30] R. T. Birge and J. Weinberg, "Least-squares' fitting of data by means of polynomials," *Reviews of Modern Physics*, vol. 19, no. 4, p. 298, 1947.

[31] T. L. K. documentation, "Lxd," https://canonical.com/lxd, accessed: 2024-2-27.

[32] "Mediawiki," https://www.mediawiki.org/wiki/MediaWiki, accessed: 2023-8-19.

[33] G. Urdaneta, G. Pierre, and M. Van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[34] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*. USENIX Association, 2006.

[35] AWS, "Aws service level agreements (slas)," https://aws.amazon.com/legal/service-level-agreements/, accessed: 2023-12-14.

[36] Kubernetes, "Kubernetes horizontal pod autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, accessed: 2023-12-14.

[37] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 301–312, 2014.

[38] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, 2005.

[39] U. Sharma, P. Shenoy, and D. F. Towsley, "Provisioning multi-tier cloud applications using statistical bounds on sojourn time," in *Proceedings of the 9th international conference on Autonomic computing*, 2012.

[40] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth, "Towards faster response time models for vertical elasticity," in *2014 ieee/acm 7th international conference on utility and cloud computing*. IEEE, 2014, pp. 560–565.

[41] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut, "Using machine learning for black-box autoscaling," in *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2016, pp. 1–8.

[42] P. Singh, A. Kaur, P. Gupta, S. S. Gill, and K. Jyoti, "Rhas: robust hybrid auto-scaling for web applications in cloud computing," *Cluster Computing*, vol. 24, no. 2, pp. 717–737, 2021.

[43] E. Rutten, S. Cerf, R. Bleuse, V. Reis, and S. Perarnau, "Sustaining performance while reducing energy consumption: A control theory approach," *arXiv preprint arXiv:2107.02426*, 2021.

[44] G. Chen and X. Wang, "Performance optimization of machine learning inference under latency and server power constraints," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022.

[45] W. Tang, Y. Ke, S. Fu, H. Jiang, J. Wu, Q. Peng, and F. Gao, "Demeter: Qos-aware cpu scheduling to reduce power consumption of multiple black-box workloads," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022.

[46] K. Kaffes, D. Sbirlea, Y. Lin, D. Lo, and C. Kozyrakis, "Leveraging application classes to save power in highly-utilized data centers," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[47] M. Savasci, "Slo-power," Mar. 2024. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.10672465

APPENDIX A
ARTIFACT DESCRIPTION

***Abstract*—This artifact section presents the evaluation of the `SLO-Power`. The `SLO-Power`, implemented in Python over 1.1K LoC, coordinates the resource and power scaling techniques to achieve power savings while adhering to service level objectives (SLOs), such as tail latency constraints. Our findings indicate that `SLO-Power` achieves exceptional power and resource efficiency.**

A. Description

This artifact points to source code files written in Python and accompanying documentation. It provides brief information about the designed system dependencies. More details are provided on the GitHub page given in the following section.

1) How to access: The code is available at https://github.com/umassos/SLO-Power. We assigned it a persistent identifier by linking it to Zenodo [47].

2) Hardware: The experiments are conducted on a cluster of Dell PowerEdge R440 servers running Ubuntu 20.04 LTS. Each server has a two-socket Intel Xeon Silver 4110 CPU, 8 cores each, 2.10GHz, and 64GB of memory. We disable hyperthreading and turbo boost. Moreover, we set OS DBPM as a CPU power management system and balanced performance as an energy-efficient policy from the BIOS settings.

3) Application: We use a latency-sensitive cloud application: MediaWiki[5]. For easy deployment, we created the LXC image of the application. The deployment details are given on the GitHub page.

B. Installation

`SLO-Power` requires the following software packages to be able to deploy.

- Ubuntu 20.04 LTS OS
- Python 3.8.10
- LXD 5.19 & LXC 5.19
- cgroups v1
- HAProxy load balancer
- httpmon workload generator

`SLO-Power` requires the following Python modules:

- grpcio
- numpy
- rapl

We generated requirement.txt for required Python modules except rapl module because it is an external module obtained from here. The details on installing the rapl module are provided on our GitHub page.

Please note that RAPL, introduced in the Sandy Bridge architecture, is Intel technology. Therefore, you need an Intel CPU that has a RAPL interface to run `SLO-Power` without any modification. If you have an AMD CPU, it should be noted that AMD has its own technology, called TDP Power Cap. It was introduced in the Bulldozer architecture for power capping. Therefore, you should check if your AMD CPU has that feature. In case it has, as soon as the power agent component of `SLO-Power` is updated to make it compatible with the AMD CPU, `SLO-Power` can be used. This operation should be straightforward.

We suggest readers create a Python virtual environment and install modules inside of this virtual environment.

C. `SLO-Power` Code Structure

`SLO-Power` is composed of two parts: cluster manager and SLO agent. The cluster manager makes the power allocation and elastic core scaling decisions, while the SLO agent enforces incoming power allocation and core scaling decisions on its running machines. The cluster manager and the SLO agent communicate via RPC calls. For this purpose, we use the gRPC framework. We also developed a service that provides an interface for CPU power measurement over the Intel RAPL interface and CPU utilization. All these components are provided in our GitHub repo.

D. Experiment workflow

To emulate a standard cloud environment and facilitate vertical scalability across servers, we have opted for the LXD[6] cluster manager. LXD is a system container manager commonly used in various use cases such as application development, testing, deployment, and cloud computing. It is a lightweight, open-source solution for running and managing Linux containers, providing a high isolation level similar to virtual machines but with lower overhead and greater efficiency. An application instance is deployed on each physical machine. Each instance of the tested application has all the necessary components,

[5] https://www.mediawiki.org/wiki/MediaWiki

[6] https://canonical.com/lxd

including web servers, database servers, or microservices. We use a latency-sensitive cloud application: MediaWiki. which hosts a replica of German Wikipedia. User requests to the application are sent through a HAProxy load-balancer, which exposes an external HTTP port interconnecting all nodes in the cluster. To emulate users accessing applications, we use the httmon tool[7], which supports the open-loop system model behavior.

E. Evaluation and expected results

In our evaluation, we use two realistic workloads: Wikipedia [33] and Azure traces[8]. These traces are provided on the GitHub page. The traces are scaled to our cluster size to avoid saturating the server. When submitting the cluster to these workloads, requests are issued using an open-loop system model [34], which creates requests both pre-determinedly and randomly by not waiting for the system's response, i.e., a new request is generated regardless if a response to previous requests is received.

F. Experiment customization

We experimented `SLO-Power` with two different real workload traces. We believe that `SLO-Power` can be experimented with more workload traces. In addition to this, `SLO-Power` can easily adapt to different cluster sizes.

[7] https://github.com/cloud-control/httpmon

[8] https://github.com/Azure/AzurePublicDataset