

PRESTO: a latency-aware power-capping orchestrator for cloud-native microservices

Rolando Brondolin

DEIB, Politecnico di Milano, Milano, IT
rolando.brondolin@polimi.it

Marco D. Santambrogio

DEIB, Politecnico di Milano, Milano, IT
marco.santambrogio@polimi.it

Abstract—Power consumption is a major concern for cloud data-centers. In this context, cloud-native applications emerged in the last few years and fostered the adoption of the cloud computing model across many organizations. Cloud-native workloads are highly heterogeneous, co-located and latency-sensitive and are able to scale to a high number of machines. To properly manage their power consumption, within this paper we propose Power REGulator for Service Time Optimization (PRESTO), a latency-aware power-capping orchestrator. PRESTO defines an Observe Decide Act (ODA) loop to manage power consumption and average latency of microservice-based workloads by considering all the network interactions between microservices in the cluster. PRESTO reduces the power consumption by 37.13% on average with a control error that is below 12.5% and below 1.5ms on average w.r.t. an unconstrained execution.

Index Terms—Autonomic power management, Latency awareness, Container orchestration

I. INTRODUCTION

Cloud computing is currently the de-facto standard solution to develop and deploy complex systems and services at scale. Applications baked by cloud computing infrastructures offer scalable services to a variable amount of users with fast response times, supporting both latency-critical workloads as well as batch workloads. In the last few years, cloud computing applications shifted from monolithic architectures to microservice-based ones, allowing to improve the development, testing, and deployment cycle of such applications. In particular, Docker [1] and Kubernetes [2] are the main tools adopted to support the microservice design pattern at scale and are the main building blocks of the so-called *cloud-native* applications. Cloud-native applications developed with the microservice pattern typically have a high degree of heterogeneity, as microservices can leverage different languages, different run-time environments, and different storage databases. They also show a high degree of co-location, as many microservices can be hosted within a single server. Finally, given that they usually provide a user-facing service, their fundamental performance metric is the request's latency.

The development of cloud-native tools and techniques combined with the ability to exploit virtually infinite resources on-demand fostered the adoption of the cloud computing model. To sustain this growth, cloud providers are continuously increasing their offering by consolidating workloads, expanding the available data-centers, and building new ones. Of course, these efforts are increasing the number of available machines

that, in turn, will increase the energy required to run them. As a consequence, according to [3], the energy usage of data-centers will reach $\simeq 8\%$ of the total energy consumption of the world by 2030. Moreover, energy usage currently represents 20% of the Total Cost of Ownership (TCO) of a data-center [4] and a relevant portion of the energy usage is devoted to servers. To make this growth sustainable, we need to carefully improve the energy efficiency of the data-center components.

As of today, the CPU represents one of the most power-hungry components of a server [5], and, as such, a thorough optimization process of the application performances' from a power consumption perspective is required. Unfortunately, modern servers are not energy proportional, meaning that their performance does not grow linearly w.r.t. their energy usage [6]. Within this context, this paper focuses on the design of a power capping orchestrator able to manage power consumption while keeping the performance of the workloads near a predefined latency Service Level Agreement (SLA). We decided to pursue this goal with a black-box approach to avoid any kind of instrumentation of the target workloads, as well as of the orchestrators and the run-time environments.

Several works addressed this challenge in the last few years. The most notable examples are PEGASUS [7], Rubik [8], and Copper [9]. PEGASUS [7] provides a feedback control loop to manage latency SLAs at scale for On-Line Data Intensive (OLDI) applications and actuates by reducing power consumption with Running Average Power Limit (RAPL) [10]. Rubik [8] introduces a statistical model to manage latency-critical workloads along with batch workloads without degrading tail latency. Finally, Copper [9] provides a control scheme to manage power consumption with RAPL providing guarantees on performance requirements. Unfortunately, they lack some aspects concerning the problem at hand. PEGASUS does not consider the heterogeneity of workload components within the same server. Rubik leverages Dynamic Voltage and Frequency Scaling (DVFS) that, according to [9], is now difficult to integrate with current hardware. Finally, Copper [9] instruments each workload component to monitor performance and requires to explicitly set the latency goal for each one.

Given the current research opportunities, in this paper, we present Power REGulator for Service Time Optimization (PRESTO). PRESTO takes into account microservices co-location and heterogeneity, provides a fully black-box approach that avoids modifying the user code, and leverages

modern power capping techniques. In particular, the contributions of this paper are the following:

- the design and development of a black-box monitoring infrastructure to measure performance, power consumption and latency of each microservice deployed in a Kubernetes cluster without user code instrumentation,
- the design and development of a graph-based analysis to attribute service time requirements to each microservice in the cluster starting from a single latency SLA that is enforced at the cluster entry-point level,
- the design and development of an Observe Decide Act (ODA) control loop able to monitor latency and power consumption, define service time goals based on the observed metrics, transform such goals in power budgets, and enforce them to all the machines in the cluster through the RAPL hardware power-capping interface.

The proposed methodology reduces the power consumption of the target workloads by 37.13% on average with a control error that is below 12.5% and below 1.5 ms on average.

The rest of this paper is organized as follows: Section II describes the main related works in the field, with a focus on microservice and Kubernetes based solutions. Section III details the step we followed to build PRESTO and its components. Section IV shows the experimental result we obtained with PRESTO using the *DeathStarBench* suite [11]. Finally, Section V concludes and derives future work.

II. RELATED WORK

Several works in the past addressed the challenge of minimizing power consumption while keeping workloads performance near a given SLA. If we consider latency as a SLA, the fundamental aspects to take into account are the definition of the correct latency target as well as the ability to maintain the target latency. *Brutlag* [12] showed that increasing the latency of a Google web search reduces the number of searches performed by each user from 0.2% to 0.6%. Although 0.2% seems negligible, at the Google scale it causes the loss of millions of searches per day with durable effects on user satisfaction. For this reason, business-critical application developers should carefully tailor the requested performance, while a power management system should be designed to precisely track those performance requirements.

For what concerns power management with latency guarantees, here we provide a brief view of the main works in the field. PowerNap [13] is an energy-conservation approach that eliminates expensive idle state with near-zero power idle states leveraging server usage traces. The authors showed that servers used for user-facing workloads typically have CPU usage below 30% or even below 10% in some cases. For this reason, the authors defined a scheme to sleep servers and activate them without significant degradation on the response time of the applications. This approach can no longer be used in the case of OLDP workloads [14], where coordinated full-system active low-power modes provide better results.

Rubik [8] is a fine-grain DVFS scheme for latency-critical workloads that adapts voltage and frequency depending on a

statistical model of the application performance. Rubik works at a sub-millisecond granularity, however, latency requirements must be explicitly set to all the applications being controlled. Moreover, according to [9], software DVFS is going to be replaced by hardware interfaces like RAPL [10].

PEGASUS [7] is a feedback-based controller that improves the energy proportionality of Warehouse Scale Computer (WSC). It adapts latency without violating SLAs and uses RAPL as an actuation system. Although PEGASUS provides interesting results with a single latency requirement for the whole cluster, it focuses on WSC, thus it does not support heterogeneity of workloads within the same server.

Wang et al. [15] provide power control with latency guarantees for Virtual Machines (VMs) by implementing a two-layers control loop where the first one balances requests among identical VMs while the second one maintains the requested latency target. SHIP [16], instead, provides a similar layered approach to achieve efficient power capping of the tenants while maximizing performance.

Copper [9] is a control scheme based on *Kalman* filtering that manages the performance of applications while reducing power consumption to achieve the requested performance. Performances are indicated as execution times and the adaptive controller approximates the non-linearities involved in this kind of power management technique. Copper is simple and reliable, but it needs to instrument the applications with the *Heartbeats* library [17]. Our approach, instead, does not need to instrument the applications' code.

To better analyze the current state of the art, here we provide the main works on power management with general performance guarantees in the context of microservices and application containers. Seer [18] is a performance debugging tool that employs deep learning to analyze the performance traces of all the microservices running in a cluster to provide predictions about SLA violations. The approach of this paper is extremely interesting for our case, as predicting the microservices' performance can improve the quality of the control activity and can further reduce the power consumption.

The work of Piraghaj et al. [19] is a framework for energy-efficient container consolidation in cloud data-centers. Containers are executed inside VMs and the goal is to minimize the overall power consumption guaranteeing a CPU-based SLA.

Dockercap [20] is an ODA control loop to manage power consumption guaranteeing a defined level of CPU usage within Docker containers. Performances are monitored parsing the output of various monitoring tools like *perf* [21], while control and actuation are limited to the single node level. Our approach scales to systems composed of several machines, supporting latency instead of CPU usage as the main performance metric. HyPPO [22] improves from Dockercap by defining an ODA loop that can scale to multiple machines, however, it does take into account only CPU usage.

The work of Townend et al. [23] studies the integration of Kubernetes clusters in the more complex data-center system, building a scheduler for the container orchestrator that takes into account both software and hardware models. Their

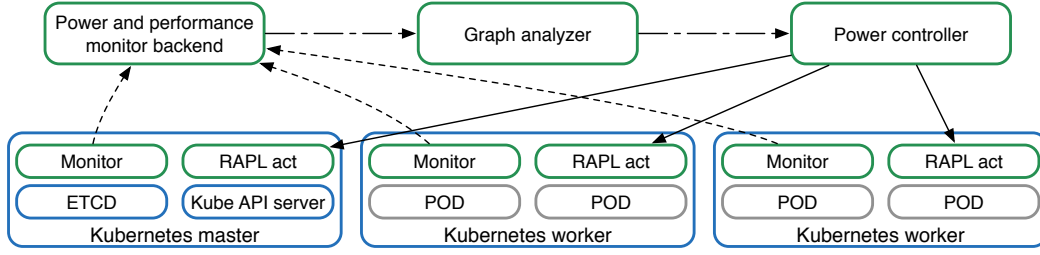


Fig. 1. PRESTO ODA control loop: we observe latency and power consumption of each Kubernetes pod in each server, centralizing metrics in a remote backend. Metrics are then passed to the Graph analyzer, which, starting from a latency requirement, defines the service times of each container. Then, the Power controller defines the power budget allocated to each server, which is enforced by the RAPL actuators.

approach reduces power from 10% to 20%. Our approach does not add components in the Kubernetes cluster, instead, it provides power capping after scheduling decisions.

Power Shepherd [24] leverages both RAPL and a CPU quota mechanism to actuate in a distributed fashion over a cluster of compute nodes. The authors state that Power Shepherd does not provide good results in the case of underutilized servers. Our power capping approach, instead, is designed to work on medium to low server utilization, while it releases the power cap in case of high utilization.

III. SYSTEM DESIGN

PRESTO is a power capping orchestrator whose goal is to reduce the power consumption of each server belonging to a Kubernetes cluster while maintaining a predefined SLA for the running cloud-native application. We decided to focus on microservices average latency as the target performance metric as it allows to express easily the performance the system should be able to achieve at any given time both from the developer perspective as well as from the end-user perspective. To have full control of the performance of each component, previous solutions required to set a latency goal for each of them. However, most of the microservice-based applications react to external inputs and show connected components that depend on each other operations [11]. For this reason, PRESTO allows setting just one latency requirement that will be translated dynamically and at run-time to latency requirements for all the components. The automatic definition of latency goals for each application component allows optimizing the power usage of each server when the application is underutilizing its resources, meaning that we will be able to slow down the application until its latency will be near the target SLA. To support this capability, we designed PRESTO following three main principles:

- 1) **Autonomicity**, as the proposed system should be able to operate with the least external intervention possible;
- 2) **Performance**, as microservices support business-critical applications and, as such, their functionality should not be compromised by power-saving systems;
- 3) **Transparency**, as workload components may vary frequently, and, as such, the use of PRESTO should not be limited to just a small set of instrumented workloads.

Figure 1 shows the main components of PRESTO, highlighted in green. From now on we will refer to microservices as *pods*, as they are the basic unit of work of Kubernetes and are defined as a collection of Docker containers providing a single functionality. To support *autonomicity*, we designed the proposed system leveraging the concept of ODA control loop. For each second, PRESTO *observes* the behavior of each pod running in each physical server without instrumenting the user code, supporting *transparency*. The *Monitor* component collects metrics about resource usage (e.g. CPU), low-level performance metrics (Performance Monitoring Counter (PMC) like cycles, Instruction Retired (IR), cache references, and cache misses), power consumption (for each container, pod, and physical host), network latency (e.g. average latency and from 50th to 99.9th percentile latency), and network bandwidth. All these metrics are sent to the *Power and performance monitor backend*, which groups them depending on the pod, the service, and the namespace. Metrics are then sent to the *Graph analyzer*, which is the first step in the *Decide* phase. The Graph analyzer builds a graph of the pods starting from their network connections and, given a latency requirement for the cluster entry-point, derives the network times and the service times each pod should provide at the next time interval to guarantee the *performance* principle. The *Power controller*, which is the second step in the *Decide* phase, takes the service times defined at the previous step and computes the power budget to be allocated to each physical server. Finally, the *RAPL act* components *actuate* for each host in the cluster the allocated power budget.

In the next sections, we will detail each step of the ODA loop, starting from the monitoring tool in Section III-A. Section III-B details how we define the service and network times starting from the single latency requirement. Then, Section III-C describes the heuristic control we perform to define the power budget, which is actuated on the system with the RAPL act component described in Section III-D.

A. Power and performance monitoring

To observe the behavior of the cluster we resorted to DEEP-mon [25], which is a monitoring tool that computes the fine-grain attribution of power consumption for each Docker container and Kubernetes pod. DEEP-mon operates by measuring PMC values with extended Berkeley Packet Filter (eBPF) [26],

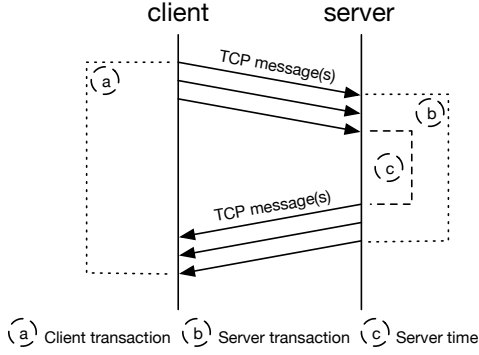


Fig. 2. Example of a network transactions over a TCP connection.

aggregating such values at the kernel level and exporting one sample to a user-space agent. The sample represents how the CPU was used in the last second by all the threads running in the system. RAPL core power values are then attributed proportionally to each thread using the weighted cycles measurement. This allows accounting for the concurrent execution of threads in Hyper Threads (HTs), which show a different power profile w.r.t. isolated execution on different physical cores [27]. Metrics are then aggregated by container within the user-space agent and then by pod, service, and namespace inside a remote backend, as shown in [22].

Unfortunately, DEEP-mon gives us just half of the information we need to build a latency-aware power-capping system. In particular, to enforce a latency requirement we first need to measure the network performance of each pod. Given that our goal is to let the user express a single requirement for the whole application, we also need to study the interactions between pods. Thus, we need to track the network performance across all the connections performed by the pods with the external world and between each other. To solve this issue, we leveraged eBPF to track all the network operations performed by the pods (both IPv4 and IPv6) and we integrated this capability within DEEP-mon. In particular, we instrumented the network stack leveraging the following Linux *kprobes*: *tcp_set_state*, *tcp_send_msg*, *tcp_rcv_msg*, and *tcp_cleanup_rbuf*. We use *tcp_set_state* to detect changes in the connection status. We use *tcp_send_msg* and *tcp_rcv_msg* kprobes to track incoming and outgoing Transfer Control Protocol (TCP) communications, while *tcp_cleanup_rbuf* is used to gather the data size read by a previous *tcp_rcv_msg* kprobe invocation.

All the kprobes we used provide just raw data about the network connection, its endpoints and the data size exchanged over TCP. To compute the network latency we resorted to the concept of *network transaction*. A network transaction is an exchange of data between a client and a server within a TCP connection, where the client starts by sending one or more messages and the server replies to them. A new transaction starts (and the previous one ends) when the client starts again to send messages after the reply from the server. The latency of

a request is the time elapsed from the first to the last message in the transaction. Figure 2 shows an example of a network transaction, where we have highlighted the execution time of such transaction from the client perspective and the server perspective. The time elapsed between the last client message and the first server message is the time required to create the response and we denoted it as *server time* (which is different from the service time if, to build the response, the server has to communicate with other application components).

Following the network transaction mechanism, we collect metrics for each network connection, where each connection is characterized by *source IP*, *source port*, *destination IP*, and *destination port*. When the connection is a plain HyperText Transfer Protocol (HTTP) connection, we collect the HTTP endpoint instead of the client port. We differentiate between client transaction and server transaction to collect different transaction times, as shown in Figure 2. Within the eBPF code, for each connection, we collect the bandwidth, the average latency and a sample of all the network latencies (currently a reservoir sampling of 240 items per second, configurable). On the user-space side of the monitoring agent, we collect those metrics and we compute the 50th, 75th, 90th, 99th, and 99.9th percentile latency and we send all the data to the remote backend. The connection data, as well as the performance data and the power consumption data, are used to build a graph of all the network interactions between pods and all the network interactions between the external world and the pods.

B. Graph-based service time estimation

The graph generated in the previous step is analyzed every 2 seconds in the *graph analyzer* component, which is deployed in the remote backend. If we focus on synchronous applicative protocols (e.g. HTTP, Remote Procedure Call (RPC) over TCP), for each pod we may find two different kind of queues: the TCP protocol queues and the client application queues. On the one hand, the TCP protocol queues provide too low-level information to be effectively used in the control of the average latency from the microservice perspective. On the other hand, client application queues are not available for the clients out of the cluster and in general cannot be accessed due to the *transparency* principle.

Given these limitations, we defined the average latency $L_i(t)$ of a given pod i at time step t according to Equation (1). The latency of a pod is the sum of its average service time $S_i(t)$ and the average time necessary to obtain a response from the pods connected to pod i (denoted by the set $dPOD_i$ that contains the downstream pods of pod i). For each pod j connected to pod i , the time necessary to obtain a response from j starting from i is the sum of the average latency $L_j(t)$ of pod j and the network time $N_{ij}(t)$ needed to reach it. Given that not all the downstream pods are always involved in the computation of the response of pod i , we weight each contribution by a factor denoted as α_j .

$$L_i(t) = S_i(t) + \sum_{j \in dPOD_i} \alpha_j(t) \cdot (L_j(t) + N_{ij}(t)) \quad (1)$$

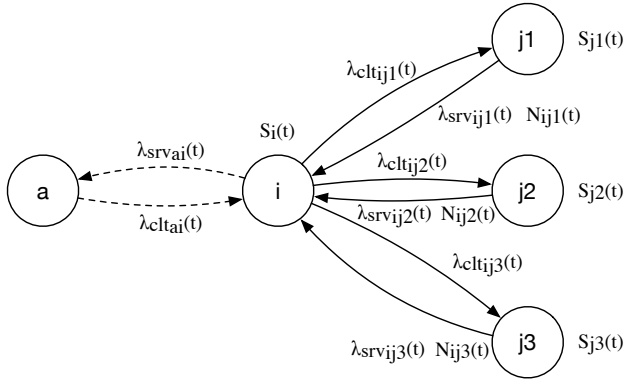


Fig. 3. Graph of microservices where pod i has a fan-out of 3 pods, each one with its own arrival rate, service time and network time.

Figure 3 helps to understand how we designed the α_j factor. First of all, we decided to define α_j as a function of time t , as the usage of the downstream pods can change depending on the loads and on the different requests coming to pod i . As we can see from Figure 3, pod i has 3 downstream pods ($j1$, $j2$, and $j3$), each one with a client connection and a server connection. Both connections are denoted by an average arrival rate and an average latency. For a given pair of connected pods (i, j), the network time $N_{ij}(t)$ can be computed as the difference between the average client latency and the average server latency. This is because on the client-side we measure from the first client request to the last server response for each transaction, while on the server-side we measure just the *server time* without the network overhead. At this point, two main cases may arise (other cases lies in the range between the two):

- 1) for each request coming to pod i , all the downstream pods are invoked to build the response of pod i ,
- 2) for each request coming to pod i , just one downstream pod is invoked to build the response of pod i .

In the first case, the arrival rate of each downstream pod will match the arrival rate of pod i . For this reason, to correctly weight the impact of each downstream pod, the sum of all the $\alpha_j(t)$ for pod i should be equal to $\frac{1}{|dPOD_i|}$. On the contrary, in the second case, the sum of the arrival rates of the downstream pods will match the arrival rate of pod i . Thus, the sum of all the $\alpha_j(t)$ for pod i should be equal to 1.

To enforce this behavior over the $\alpha_j(t)$ factor, we need to use the number of requests that are handled by pod i and by the downstream pods. In particular, we can denote $\lambda_{srv_i}(t)$ and $\lambda_{srv_j}(t)$ as the sum of all the server requests handled respectively by pod i and pod j . We can do the same for the client requests, where, for instance, $\lambda_{clt_i}(t)$ is the sum of all the requests generated by pod i as a client to all the downstream pods. Starting from these values, we define $\alpha_j(t)$ for a given pod j as in Equation 2, where $\alpha_j(t)$ is the product of two different contributions. The first contribution is the ratio between the requests served by pod i to other pods ($\lambda_{srv_i}(t)$) and the client requests generated by pod i to the downstream

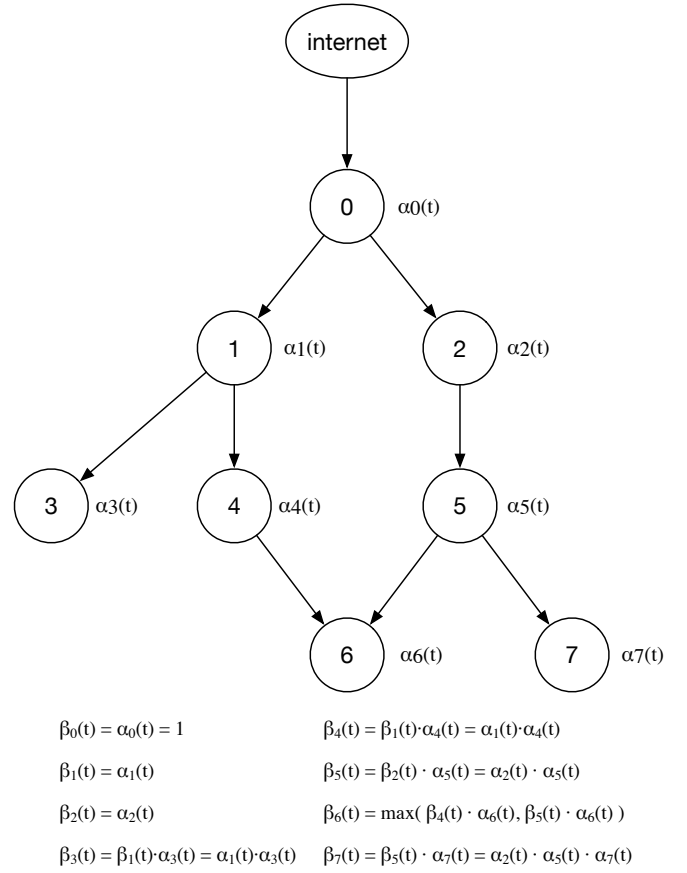


Fig. 4. Abstract representation of a microservice application. Vertices are pods, edges are connections. Each pod has a α coefficient computed according to Equation (2). β coefficients are computed following a breadth-first search of the graph starting from the internet vertex, which is outside of the cluster.

pods ($\lambda_{clt_i}(t)$). This contribution weights the impact of pod i w.r.t. the rest of the pods in the application. The second contribution is the ratio between the number of requests served by pod j to other pods ($\lambda_{srv_j}(t)$) and the client requests generated by pod i to the downstream pods. This second contribution weights the impact of the downstream pod j w.r.t. the requests performed by pod i to all the downstream pods.

If the pods are behaving as in case 1, the first contribution will weight $\frac{1}{|dPOD_i|}$, while the second contribution will weight $\frac{1}{|dPOD_i|}$ as well. If we sum up the $\alpha_j(t)$ we will obtain $\frac{1}{|dPOD_i|^2} |dPOD_i|$ times, which will lead to the expected $\frac{1}{|dPOD_i|}$. If we are in case 2, instead, the first contribution will fall down to 1, while the second contribution will lead to $\frac{1}{|dPOD_i|}$ when the $\alpha_j(t)$ factors are summed up for pod i .

$$\alpha_j(t) = \frac{\lambda_{srv_i}(t)}{\lambda_{clt_i}(t)} \cdot \frac{\lambda_{srv_j}(t)}{\lambda_{clt_j}(t)} = \frac{\lambda_{srv_i}(t) \cdot \lambda_{srv_j}(t)}{\lambda_{clt_i}(t)^2} \quad (2)$$

Once we defined the effects of $\alpha_j(t)$, we can estimate the latency of the whole microservice-based application to then try to enforce the new latency target. To do so, we need to solve Equation (1) for a given entry-point. Figure 4 shows

an example of a microservice-based application graph where vertices and edges are pods and client connections respectively. The *internet* vertex represents all the client endpoints outside of the cluster. The first pod in the path that starts from the internet vertex is our entry-point, for which we define the average latency of the microservice application. We can then explore the graph with a breadth-first search. Within this exploration, we compute a new factor that we denote $\beta_i(t)$ that keeps into account the path each request should follow from the entry-point, where i is a pod in the application. In the case of multiple visits of the same pod i , we select the highest $\beta_i(t)$ among the visits to enforce a harder requirement on the final average latency. After the exploration of the graph, we can estimate the latency $L_{out}(t)$ at the entry-point as in Equation (3). Given that we computed the $\beta_i(t)$ factors, the latency at the entry-point $L_{out}(t)$ is just the sum of two contributions. On the one hand, we have the sum of the service times $S_i(t)$ of all the pods i multiplied by $\beta_i(t)$ (this first contribution can be summarized as the total service time of the application $S_{out}(t)$). On the other hand, we have the sum of the network times $N_{ij}(t)$ between all pod pairs (i, j) that are connected by an edge of the graph, multiplied by the highest available $\alpha_j(t)$ (this second contribution can be summarized as the total network time of the application $N_{out}(t)$).

$$L_{out}(t) = \sum_{i \in POD} \left(\beta_i(t) \cdot S_i(t) \right) + \sum_{(i,j) \in C} \left(\alpha_j(t) \cdot N_{ij}(t) \right) \quad (3)$$

Once we have the estimation of the entry-point latency, we can compute the requirements each pod should satisfy to meet a given SLA. Unfortunately, we are not able to control the network times unless we get access to the network switches, as was done in [28] in the context of VM consolidation. Thus, we decided to focus on the service times of each pod to enforce the latency requirement. Within this context, if we define the target latency SLA as $\overline{L_{out}}$, the service time at the next time step $S_i(t+1)$ for a given pod i can be computed according to Equation (4). This equation simply scales the last value of the service time $S_i(t)$ of pod i by the ratio between the expected service time of the application and the current service time of the application $S_{out}(t)$. The expected service time is computed as the difference between the latency target $\overline{L_{out}}$ and the current network latency $N_{out}(t)$.

$$S_i(t+1) = S_i(t) \cdot \frac{\overline{L_{out}} - N_{out}(t)}{S_{out}(t)} \quad (4)$$

C. Latency-aware reactive control

Once we computed the service time each pod should guarantee at the next time interval, we can then try to enforce it through the *power controller*. We designed a heuristic controller, placed inside the cluster as a normal pod, that reacts to changes in the arrival rate of the application and to changes in the enforced service times. We decided to resort to the utilization based *Little's law* to translate the enforced service time into a requested utilization. For each physical server k

in the Kubernetes cluster, for each second, we select the pod with maximum utilization, as shown in Equation (5). We target the pod with maximum utilization because, otherwise, if we increase too much the power cap, such pod will be the first to saturate reducing the performance in an uncontrolled way.

$$U_c(k, t) = \max_{i \in POD} (U_i(k, t)) \quad (5)$$

Starting from the candidate utilization $U_c(k, t)$, we can define the control error over the utilization $e_u(k, t+1)$ as in Equation (6). We define this error as the difference between the current candidate utilization $U_c(k, t)$ and the target utilization that this pod should achieve, according to the *Little's law*.

$$e_u(k, t+1) = U_c(k, t) - \lambda_c(t) \cdot S_c(t+1) \quad (6)$$

We then define the latency error $e_l(t)$ of the overall application according to Equation (7). The latency error is the absolute value of the percentage error between the target average latency $\overline{L_{out}}$ and the current average latency $L_{out}(t)$.

$$e_l(t) = \left| \frac{\overline{L_{out}} - L_{out}(t)}{\overline{L_{out}}} \right| \quad (7)$$

Starting from the utilization error and the latency error we define the candidate power budget $P_c(k, t+1)$ of server k as in Equation (8). $P_c(k, t+1)$ is the sum of the current core power consumption of the server $P(k, t)$ and the actionable power $P_{act}(k, t)$ multiplied by the utilization error $e_u(k, t+1)$ and the latency error $e_l(t)$. The actionable power $P_{act}(k, t)$ is the power generated by the pods controlled by PRESTO.

$$P_c(k, t+1) = P(k, t) + e_u(k, t+1) \cdot e_l(t) \cdot P_{act}(k, t) \quad (8)$$

Finally, we define the final power budget for server k as in Equation (9). The power budget $P(k, t+1)$ is equal to $P_c(k, t+1)$ if the current latency of the microservice-based application $L_{out}(t)$ is less than $2 \cdot \overline{L_{out}}$, otherwise the maximum power budget $P_{max}(k)$ for the server is enforced. This condition is necessary to let the system to move away from the equilibrium in case of violation of the SLA requirement.

$$P(k, t+1) = \begin{cases} P_{max}(k) & L_{out}(t) > 2 \cdot \overline{L_{out}} \\ P_c(k, t+1) & otherwise \end{cases} \quad (9)$$

D. RAPL-based power allocation

Once the *power controller* has computed the power budgets for all the servers in the Kubernetes cluster, it makes this data available through a REST interface. Each *RAPL act* pod is executed as a Kubernetes *DaemonSet* on each host. To enforce the power cap, each pod is executed in privileged mode and we mount the */dev/cpu* folder to get access to the Model Specific Register (MSR) files. The *RAPL act* pod retrieves the data coming from the controller and transforms the power cap expressed in Watt into power units by reading the power unit MSR and applying the relative formula described in the Intel manual [29]. Then, the power budget is enforced on the *package* domain by writing on the proper MSR.

TABLE I

EXPERIMENTAL SETUP FOR THE SOCIAL-NETWORK BENCHMARK AND THE MEDIA-MICROSVCS BENCHMARK. FOR EACH WORKLOAD WE SHOW THE NUMBER OF PODS INVOLVED AND THE REQUEST RATES FOR LOW, MEDIUM, AND HIGH CONFIGURATIONS.

benchmark	workload	# pods	low $\lambda(t)$	mid $\lambda(t)$	high $\lambda(t)$	duration
social-network	compose post	21	300 req/s	400 req/s	500 req/s	300 s
social-network	read home timeline	5	5000 req/s	6000 req/s	8000 req/s	300 s
media-microsvc	compose review	27	250 req/s	300 req/s	350 req/s	180 s

IV. EVALUATION

Within this Section, we evaluate the performance of PRESTO w.r.t. the ability to keep the average latency of the applications near the latency target and w.r.t. the power savings that this activity can generate. Section IV-A will detail the experimental setup we built to validate the proposed methodology, while Section IV-B will describe the results obtained w.r.t. power savings and performance of the applications.

A. Experimental setup

The experimental campaign was conducted on a small cluster composed of two Dell PowerEdge r720xd equipped with 2x Intel Xeon E5-2680 Ivy Bridge with 10 cores each (20 HT) clocked at 2.80GHz and with 380GB of RAM. The host Operating System (OS) is an Ubuntu Linux 16.04 with kernel 4.15, eBPF, Docker 18.06.2 community edition, and Kubernetes v1.17.4. We tested the proposed methodology with two benchmarks from the *DeathStarBench* benchmark suite [11], an open-source¹ suite of microservice-based benchmarks. We chose the *social-network* and the *media-microsvc* benchmark because, at the time of writing, those benchmarks have a supported Kubernetes distribution. The *social-network* benchmark represents a small social-media application with microservices deputed to user management, user timeline management, home timeline management, post composition, media storage, social graph management, search, and URL shortening. When needed, microservices store data inside MongoDB² servers, Redis³ servers, and Memcached⁴ in-memory stores. The *media-microsvc* benchmark, instead, implements a media reviewing, renting and streaming platform and is composed of identification services (users and movies), a review composition service, a page composition service, services for users and reviews consultation, and a video streaming service. Also, the *media-microsvc* benchmark leverages MongoDB, Redis, and Memcached data stores. Both benchmarks combine an HTTP front-end with a Thrift-based RPC system between pods and TCP connections to data stores.

Table I shows how we set-up the tests for the experimental campaign. The *social-network* benchmark provides two different workloads: on the one hand, we have scripts for the post composition, while, on the other hand, we can read the home timeline of each user in the system. The *media-microsvc* benchmark, instead, has just the compose review workload.

For each workload, we set 3 different arrival rates, identified as low, mid, and high. For the compose post we set 300 req/s, 400 req/s, and 500 req/s for low, mid, and high respectively. For the read home timeline workload, we set 5000 req/s, 6000 req/s, and 8000 req/s for low, mid, and high respectively. Finally, for the compose review workload we set 250 req/s, 300 req/s, and 350 req/s for low, mid, and high respectively. The workloads for the *social-network* benchmark were run for 5 minutes, while the workload of the *media-microsvc* benchmark was run for 3 minutes. Due to a race condition in the UniqueID microservice, we had to limit the number of parallel connections to the compose post and the compose review workloads. The benchmarks were still able to generate significant load, but they were limited in scalability.

For each workload and each arrival rate, we perform 30 runs of the experiment both for the workload equipped with PRESTO, as well as the unconstrained workload (denoted as *alone*). Each run is executed thanks to a *wrk2*⁵ load generator that is launched from a third machine identical to the previous two. For each run, we collect the average latency, the power consumption, the energy usage throughout the experiment, and the percentile latencies of the workload. The experimental results are presented in Section IV-B.

B. Experimental results

Table II shows a synthetic view of the results we obtained running the experiments described in Section IV-A. If we consider the compose post workload, we can see that the unconstrained run generated an average latency of 10.08ms, 9.27ms, and 8.83ms for the low, mid, and high arrival rates respectively. Given the average latency showed by the benchmark, we decided to bring the latency target to 13ms, which represents an increase of $\simeq 40\%$ w.r.t. the normal behavior of the workload. Table II shows that the proposed methodology was able to increase the average latency, with a Relative Error (RE) of -16.44%, -13.21%, and 14.69% for the low, mid, and high arrival rates respectively. Although the RE is not negligible, the difference between the latency target and the achieved latency is bounded to $\simeq 2.6$ ms. Within this context, PRESTO was able to reduce power consumption by 7.01%, 35.15%, and 23.99% for the low, mid, and high arrival rates.

If we look at the read home timeline workload, we can see that the unconstrained execution has an average latency of $\simeq 2$ ms. In this case, we decided to bring the target latency to 8ms, which is a 5x increase on average latency. Table II shows that PRESTO was able to increase the average latency also in this case, with a RE of -11.15%, -21.65%, and -24.52% for the low, mid, and high arrival rates respectively. Again, if we look at the difference between the latency target and the achieved latency, we can see that it is bounded to $\simeq 2.6$ ms also in this case. If we consider instead the power savings, we can see that PRESTO reduces the power consumption of 46.26%, 42.52%, and 42.69% for the low, mid, and high arrival rates.

Finally, Table II shows also the results for the compose review workload. In this case, the benchmark exposes an

¹<https://github.com/delimitrou/DeathStarBench>

²<https://www.mongodb.com>

³<https://redis.io>

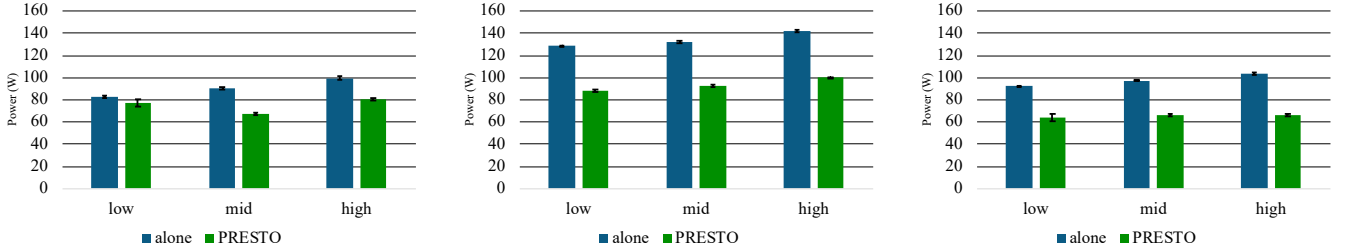
⁴<https://memcached.org>

⁵<https://github.com/giltene/wrk2>

TABLE II

EXPERIMENTAL RESULTS FOR THE 3 WORKLOADS WITH 3 DIFFERENT ARRIVAL RATES FOR EACH WORKLOAD. TABLE REPORTS THE AVERAGE LATENCY WHEN THE BENCHMARK IS NOT CONSTRAINED, THE LATENCY TARGET, THE AVERAGE LATENCY ACHIEVED BY PRESTO, THE LATENCY DIFFERENCE, THE LATENCY RELATIVE ERROR, AND THE POWER SAVINGS W.R.T. THE NOT CONSTRAINED EXECUTION.

	configuration	$\lambda(t)$	alone avg latency	latency target	avg latency	Δ latency	relative error	power saving
compose post	low	300 req/s	10.08 ms	13 ms	15.66 ms	-2.66 ms	-16.44%	7.01%
	mid	400 req/s	9.27 ms	13 ms	15.25 ms	-2.25 ms	-13.21%	35.15%
	high	500 req/s	8.83 ms	13 ms	11.43 ms	1.57 ms	14.69%	23.99%
read home timeline	low	5000 req/s	1.92 ms	8 ms	9.04 ms	-1.04 ms	-11.15%	46.26%
	mid	6000 req/s	1.89 ms	8 ms	10.22 ms	-2.22 ms	-21.65%	42.52%
	high	8000 req/s	2.10 ms	8 ms	10.62 ms	-2.62 ms	-24.52%	42.69%
compose review	low	250 req/s	9.72 ms	15 ms	14.54 ms	0.46 ms	5.94%	43.93%
	mid	300 req/s	10.24 ms	15 ms	15.51 ms	-0.51 ms	-2.70%	47.53%
	high	350 req/s	10.32 ms	15 ms	14.83 ms	0.13 ms	2.14%	45.09%
Averages (on absolute values)						1.50 ms	12.49%	37.13%



(a) Compose post power consumption with arrival rate of 300 req/s, 400 req/s, and 500 req/s. (b) Read home timeline power consumption with arrival rate of 5000 req/s, 6000 req/s, and 8000 req/s. (c) Compose review power consumption with arrival rate of 250 req/s, 300 req/s, and 350 req/s.

Fig. 5. Comparison of power consumption of the 3 workloads with different arrival rates. Error bars represent 95% confidence interval. Lower is better.

average latency of $\simeq 10$ ms and we enforce a less strict latency target: 15ms. Within this case, the proposed methodology achieves a RE that is below 6% in all cases, with power savings that goes from 43.93% for the low arrival rate, to 47.53% for mid and 45.09% for high.

Although we obtained good results on the power savings side, the RE exposed by PRESTO is not negligible in some cases. This is mainly due to the actuation mechanism. Given that we leverage RAPL, the power cap enforced is valid for all the cores in the system. This is taken into account by the controller that defines the power cap depending on the pod with the highest utilization within each physical server. However, this poses some challenges: first of all, the pod with the highest utilization within a server often works near its saturation, thus, small and frequent changes in the power cap lead to high oscillations of the performance of the pod. To solve this issue, we decided to change the power cap of the server only if the new power cap has a difference of at least 1W. This, of course, results in the possibility to reach the equilibrium at a value that is slightly higher or smaller than the target latency. Read home timeline and compose post exposes this behavior, although it is bounded to $\simeq 2.6$ ms, while compose review reaches the equilibrium with a smaller error.

To further analyze the results we obtained, Figure 5 shows the power consumption of the three workloads when executed with and without PRESTO. Figure 5(a) shows the power consumption of the compose post workload for different arrival rates. As we can see, the difference between the power consumption increased from low to mid, but then slightly reduces

from mid to high. This is because in the low case PRESTO hits the minimum of the RAPL actuator for one server, which is set to 30W. Then, in the high case, power savings decrease because the controller sets a slightly lower latency w.r.t. the target. In Figure 5(b) the power consumption for the read home timeline workload is reported. This workload is the one that has the highest power consumption among the three analyzed workloads. As we can see, the power consumption increases for both the unconstrained execution as well as for PRESTO when we increase the arrival rate. Power savings for this case remain similar among the different arrival rates. Finally, Figure 5(c) shows the power consumption of the compose review workload for the different arrival rates that we described in Table I. Here we can see a slight increase in power consumption for both the unconstrained execution as well as the one managed by PRESTO. This is because the increase in the arrival rate is less steep w.r.t. the other cases. Again, the power savings remain fairly similar across the different runs, indicating a system that is stable and that can control the performance accurately within this case.

To better analyze the effects of PRESTO on performance, we decided to collect data also on the latency percentile provided by the workloads during the experiments. In particular, we leverage the *uncorrected latency* measurements of *wrk2*. Here we report only the box plots for all the workloads we analyzed for the mid arrival rate case, as similar considerations can be made for the other configurations. Figure 6 shows the 50th, 75th, 90th, and 99th percentile latency for the compose post workload for the unconstrained execution (Figure 6(a))

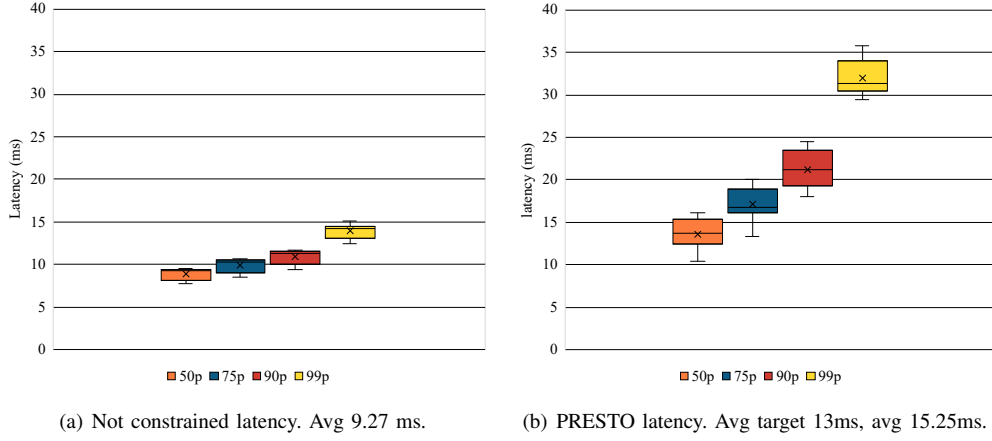


Fig. 6. Box plots of the percentile latency (50th, 75th, 90th, and 99th) of the compose post workload with 400 req/s across 30 runs.

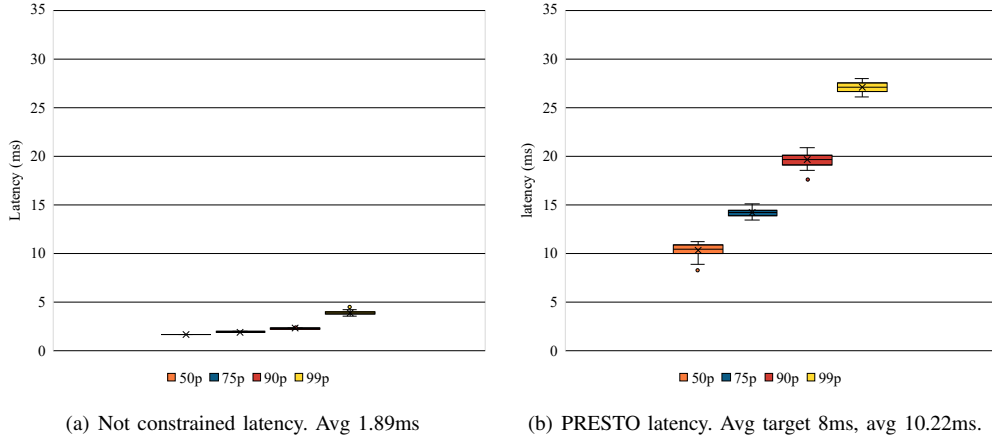


Fig. 7. Box plots of the percentile latency (50th, 75th, 90th, and 99th) of the read home timeline workload with 6000 req/s across 30 runs.

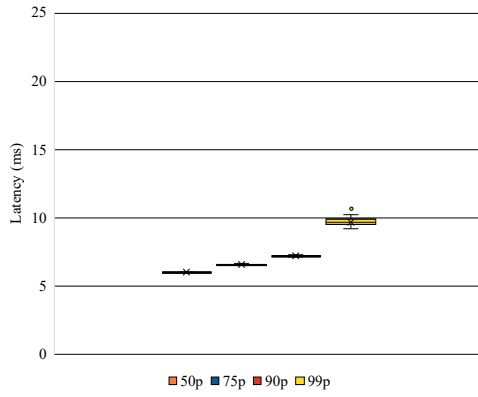
and the one managed by PRESTO (Figure 6(b)). The power capping imposed by PRESTO increased not only the average latency but also all the latency percentiles. In particular, we can see that the average and the median of the 50th percentile are in line with the latency target, while the other latency metrics increase with similar paces w.r.t. the unconstrained ones. As expected, PRESTO slightly increases the variability of the latency results due to the reactive control activity. Similar considerations can be made for the latency percentiles of the read home timeline workload (shown in Figure 7). Within this case, however, the 50th percentile is always higher than the latency target (except for one outlier). This is due to the lower latency of the workload w.r.t. the compose post one and the 5x latency increase required by the latency target that poses more challenges in the control activity. The variability induced by PRESTO is lower w.r.t. the compose post case, because the workload involves fewer microservices, as indicated in Table I. Finally, Figure 8 shows the results for the compose review workload. As we can see, the 50th percentile and the 75th percentile latency metrics are below the latency target. This indicates that the average latency is heavily affected by the tail latency. Within this case, PRESTO correctly increases the

average latency, modifying the latency percentiles accordingly without abrupt changes in the application behavior.

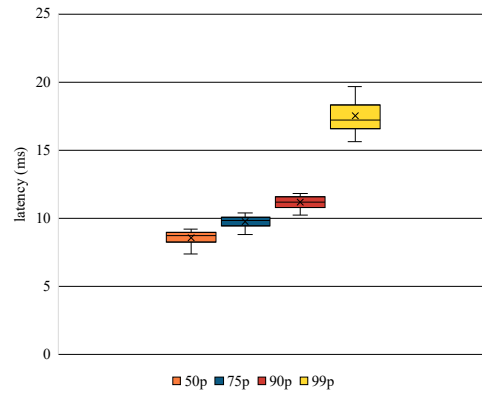
V. CONCLUSION AND FUTURE WORK

Within this paper, we presented PRESTO, a latency-aware power capping orchestrator specifically designed to manage the heterogeneity of microservice-based workloads. The proposed ODA control loop is based on the transparency, performance, and autonomicity principles, and provides a fine-grain monitoring tool combined with a graph-based analysis of the running workloads, a reactive and heuristic controller, and a fast actuation based on RAPL. The proposed approach reduces the power consumption of 37.13% w.r.t. an unconstrained execution, with a RE in the control activity that is 12.49% on average. The absolute error on average remains below 1.50ms.

Future works will investigate the scalability of the proposed approach, addressing also the tail latencies of the controlled workloads and considering also DRAM power consumption. We will then extend PRESTO by adding tuning knobs like CPU quota and pinning, by introducing prediction mechanisms within the controller and by improving the resiliency of the system in case of diurnal traffic patterns.



(a) Not constrained latency. Avg 10.24ms



(b) PRESTO latency. Avg target 15ms, avg 15.51ms.

Fig. 8. Box plots of the percentile latency (50th, 75th, 90th, and 99th) of the compose review workload with 300 req/s across 30 runs.

REFERENCES

- [1] "Docker containers," <https://www.docker.com>, [Online; accessed 29-Mar-2020].
- [2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.
- [3] A. S. Andrae and T. Edler, "On global electricity usage of communication technology: trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.
- [4] Y. Cui, C. Ingaz, T. Gao, and A. Heydari, "Total cost of ownership model for data center technology evaluation," in *Thermal and Thermo-mechanical Phenomena in Electronic Systems (ITherm)*, 2017 16th IEEE Intersociety Conference on. IEEE, 2017, pp. 936–942.
- [5] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.
- [6] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, 2007.
- [7] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Computer Architecture (ISCA)*, 2014 ACM/IEEE 41st International Symposium on. IEEE, 2014, pp. 301–312.
- [8] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 598–610.
- [9] C. Imes, H. Zhang, K. Zhao, and H. Hoffmann, "Copper: Soft real-time application performance using hardware power capping," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2019, pp. 31–41.
- [10] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, Mar. 2012. [Online]. Available: <http://dx.doi.org/10.1109/MM.2012.12>
- [11] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.
- [12] J. Brutlag, "Speed matters for google web search," 2009.
- [13] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 205–216.
- [14] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 319–330.
- [15] Y. Wang, X. Wang, M. Chen, and X. Zhu, "Power-efficient response time guarantees for virtualized enterprise servers," in *2008 Real-Time Systems Symposium*. IEEE, 2008, pp. 303–312.
- [16] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller, "Ship: Scalable hierarchical power control for large-scale data centers," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 91–100.
- [17] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th international conference on Autonomic computing*, 2010, pp. 79–88.
- [18] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 19–33.
- [19] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A framework and algorithm for energy efficient container consolidation in cloud data centers," in *Data Science and Data Intensive Systems (DSDIS)*, 2015 IEEE International Conference on. IEEE, 2015, pp. 368–375.
- [20] A. Asnaghi, M. Ferroni, and M. Santambrogio, "Dockercap: A software-level power capping orchestrator for docker containers," in *Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, 2016 IEEE Intl Conference on. IEEE, 2016, pp. 90–97.
- [21] A. C. De Melo, "The new linux'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.
- [22] M. Arnaboldi, R. Brondolin, and M. D. Santambrogio, "Hyppo: Hybrid performance-aware power-capping orchestrator," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2018, pp. 71–80.
- [23] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu, "Improving data center efficiency through holistic scheduling in kubernetes," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 156–161.
- [24] J. Krzywda, A. Ali-Eldin, E. Wadbro, P.-O. Östberg, and E. Elmroth, "Power shepherd: Application performance aware power shifting," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019, pp. 45–53.
- [25] R. Brondolin, T. Sardelli, and M. D. Santambrogio, "Deep-mon: Dynamic and energy efficient power monitoring for container-based infrastructures," in *Parallel and Distributed Processing Symposium Workshops, 2018 IEEE International*. IEEE, 2018, pp. 676–684.
- [26] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *USENIX winter*, vol. 93, 1993.
- [27] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars, "Happy: Hyperthread-aware power profiling dynamically," in *USENIX Annual Technical Conference*, 2014, pp. 211–217.
- [28] K. Zheng, W. Zheng, L. Li, and X. Wang, "Powernets: Coordinating data center network with servers and cooling for power optimization," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 661–675, 2017.
- [29] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.