Unleashing the Scalability Potential of Power-Constrained Data Center in the Microservice Era

Xiaofeng Hou, Jiacheng Liu, Chao Li, Minyi Guo
Department of Computer Science and Engineering, Shanghai Jiao Tong University
{xfhelen,liujiacheng}@sjtu.edu.cn, {lichao,guo-my}@cs.sjtu.edu.cn

ABSTRACT
Recent scale-out cloud services have undergone a shift from monolithic applications to microservices by putting each functionality into lightweight software containers. Although traditional data center power optimization frameworks excel at per-server or per-rack management, they can hardly make informed decisions when facing microservices that have different QoS requirements on a per-service basis. In a power-constrained data center, blindly budgeting power usage could lead to a power unbalance issue: microservices on the critical path may not receive adequate power budget. This unavoidably hinders the growth of cloud productivity.

To unleash the performance potential of cloud in the microservice era, this paper investigates microservice-aware data center resource management. We model microservice using a bipartite graph and propose a metric called microservice criticality factor (MCF) to measure the overall impact of performance scaling on a microservice from the whole application’s perspective. We further devise ServiceFridge, a novel system framework that leverages MCF to jointly orchestrate software containers and control hardware power demand. Our detailed case study on a practical microservice application demonstrates that ServiceFridge allows data center to reduce its dynamic power by 25% with slight performance loss. It improves the mean response time by 25.2% and improves the 90th tail latency by 18.0% compared with existing schemes.

ACM Reference Format:

1 INTRODUCTION
In recent years, microservice architecture has become an important trend in deploying cloud computing applications. Microservice transfers a monolith containing the entire service’s functionality in a single program to tens or hundreds of lightweight and loosely-coupled mini services [22]. Some key attributes of microservice such as domain-driven design, on-demand virtualization and infrastructure automation [4] make microservice fit nicely to the model of container-based computing environment. Today, many mainstream cloud providers such as Alibaba, Google, Microsoft and Amazon have adopted this application model, as shown in Table 1.

A key benefit of the microservice architecture is the scale-out capability it enables. For traditional online data intensive (OLDI) applications (such as web search and social network services), data centers usually provide excessive computing resources and extra power budget for reducing the tail latency, which causes low system utilization [20]. The typical utilization of data centers hosting online services is often less than 50% [2, 24]. With microservice architecture’s unique two-tier topology [47], user queries always pass a specific API layer and access many mini services. By dividing a monolithic application into process-level services, microservices greatly facilitate tail request scheduling. Meanwhile, since heavy queries are served with multiple separate services, one can avoid local power peaks induced by traffic flood. It has been shown that Tmall (the world’s second biggest e-commerce website operated by Alibaba) based on microservice is able to withstand tens of thousands of requests from global users [5].

Although microservice architecture offers new opportunities for accommodating ever-growing workloads in the cloud, its true scalability potential has not been exploited yet. The reason behind this is two-fold. Firstly, the heterogeneity of microservices is never well-exposed to the data center management layer, unavoidably causing power allocation imbalance and power capacity waste. Today’s data centers make their power management decisions mainly based on application level [29, 39] or server level [14, 28] activities. Oftentimes, they overlook the sensitivity of performance to power budgeting of each individual microservice. As a result, it could waste precious power budget on some less critical microservices while leaving inadequate power budget to the most important ones. Secondly, existing power management schemes lack the agility to react at per-service granularity, further exacerbating the above imbalance issue. Representative optimization schemes such as per-server voltage/frequency tuning [12, 15, 36], rack-level thermal/cooling optimization [35, 37] and battery-based power peak shaving [12, 25, 30] are too coarse-grained to fulfill a service-oriented control. Designing aggressively fine-grained strategy on top of existing power management frameworks often incurs high software control overhead, which can compromise the benefits that the optimization may provide.

<table>
<thead>
<tr>
<th>Enterprise</th>
<th>Container</th>
<th>Product</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alibaba</td>
<td>Docker</td>
<td>Dubbo</td>
<td>All except Taobao</td>
</tr>
<tr>
<td>Google</td>
<td>Docker</td>
<td>Kubernetes</td>
<td>Google Clouds</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Docker</td>
<td>.Net Framework</td>
<td>Azure Core, Skype(Business)</td>
</tr>
<tr>
<td>Amazon</td>
<td>Docker</td>
<td>ECS</td>
<td>Amazon.com</td>
</tr>
</tbody>
</table>

Table 1: Many mainstream cloud providers are tapping into microservice architecture today.
In this paper we show how to further scale out power-constrained data center in the microservice era by matching server-level power budget variation to fine-grained microservice heterogeneity. The first challenge is to develop the mechanism for identifying critical microservices who dominate the performance of the entire application. It depends on both the organization of the microservices that form the application and the dynamic power behavior of each microservice that must be analyzed during run-time analysis. Another obstacle before us is the power management overhead that comes with per-service control and cross-service coordination. Designing a completely new framework that fits the needs of microservice-level power allocation optimization requires significant changes to existing power management schemes. Thus, we focus our attention on efficiently adapting exiting server-level power management schemes to microservices and exploiting a medium-grained scheduling strategy for a desired performance-power trade-off.

We first propose microservice criticality factor (MCF), a comprehensive metric for evaluating the importance of each microservice to the performance of the entire application. MCF models the two-layer microservice architecture as a bipartite graph. It takes into account both how much time a microservice is performing useful work and how performance scaling affects the running microservices. Each microservice’s MCF is calculated based on three static factors and one dynamic factor. The static factors include execution duration, call times and performance-power characteristics of a microservice - all of them determine the weight of an edge in the graph. The dynamic factor reflects the variation of the incoming requests. It determines the indegree of a microservice, i.e., vertex in the graph. MCF depicts the likelihood that power budget capping on that particular microservice will result in severe QoS violation. High criticality often means more sensitive to power variation. In contrast, slowing down non-critical microservices has negligible impact on performance, which allows for a more aggressive execution mode for energy saving.

Based on our characterization of microservice, we propose a novel power management coordination framework named ServiceFridge. The major feature of our framework is that it can synergistically combine different server-level power management schemes to adapt power-hungry data centers to the microservice environment. In addition to coordinate power management schemes, ServiceFridge jointly manages the container orchestration system (a system for deploying, scaling and managing containerized applications, such as kubernetes [45], docker swarm [49]) as well. In order to provide service-oriented power allocation (but not too fine-grained), ServiceFridge logically partitions a data center into three different zones: a hot zone, a warm zone, and a cold zone. The cold zone targets microservices that require guaranteed QoS; servers in this zone are configured without any power budget limiting operations to guarantee high performance. The hot zone allows aggressive power capping on low MCF services to squeeze further energy savings from the power-hungry data centers. Finally, a warm zone hosts services with uncertain criticality. It serves as a buffer between the hot and the cold zones. During runtime, ServiceFridge isolates microservices into different zones and dynamically swaps microservices between different zones for ensuring optimal quality-of-service as well as meeting power budget.

This paper makes the following contributions:

- We investigate emerging microservice architecture and discuss its implication on power-limited data centers.
- We propose microservice criticality factor (MCF), a metric for identifying the critical microservices. It allows data centers to evaluate the necessity of power allocation for a microservice with the static and dynamic factors.
- We propose ServiceFridge, a cross-layer power management coordination framework adapted to the microservice environment. It isolates the power management of different types of microservices for better performance-power trade-off.
- We implement our design as a proof-of-concept system and conduct a case study with real-world microservice applications containing more than 42 microservices. We show that ServiceFridge could reduce dynamic power range by 25% with the slightest performance loss.

The rest of this paper is organized as follows. Section 2 introduces microservice in this paper. Section 3 discusses critical microservice. Section 4 proposes a new metric called MCF to measure microservice criticality. Section 5 describes ServiceFridge power coordination framework for microservice. Section 6 presents experimental methodology and evaluation results. Section 7 discusses related work and Section 8 concludes this paper.

2 BACKGROUND AND MOTIVATION

2.1 The Microservice Revolution

Microservice architecture is appealing to cloud providers for its agile software development, high service quality, and scalable service deployment. Microservice architecture presents application as a suite of services [22]. Unlike monolith encompassing the entire application’s functionality into a huge block of code, each microservice runs as a single process. These small processes communicate with each other through lightweight mechanisms such as an HTTP resource API [48]. Therefore, it is easier and faster to deploy and remove a small service with microservice architecture. It allows a system to conveniently dispatch computation resources according to the real-time demand. Even if a failure occurs, a microservice based application can continue running with graceful degradation.

The benefit of microservice architecture mainly comes from decomposition. As shown in Figure 1, microservice architecture decomposes a monolithic application into two layers [47]. The API layer acts as a service-access portal and the service layer contains massive loosely-coupled microservices. The clients always access a
specific service via an API and is responded by several microservices. In this work we refer to the API and its corresponding microservices as a microservice region. Microservice regions that employ different numbers/types of microservices can realize different functions. This topology enables fine-grained dispersion and isolation of different functions, which brings new opportunities to scale out the data center. For example, it allows one to spatially disperse the explosively high-volume query floods into multiple microservice regions located on different physical servers. One can also balance resource allocation among different functions to accelerate program execution.

### 2.2 Power Management Challenge

Power and energy resources significantly limit the scalability of data centers today [10–12, 15, 27, 36, 39]. As cloud applications proliferate and data-analyzing demands continue to increase, it is important to improve data center utilization and smartly use power budget. For conventional online data-intensive (OLDI) workloads in monolithic deployment mode, power-saving techniques such as workload consolidation and performance scaling can be problematic [13, 15, 20]. When running iterations of the full application across thousands of servers, it is difficult to determine which iteration can cause unacceptable SLAs (Service-Level-Agreements) violations under power shortage.

Although the microservice architecture shows great promise in alleviating the above issue, realizing the true power saving potential in a microservice based data center can be challenging. Existing power management schemes, no matter server-level [14, 30] or OS-level [23], are unaware of the topology and heterogeneity of microservices (detailed in Section 3). Consequently, they often lead to sub-optimal power allocation or even unbalanced power allocation with significant power waste. In addition, even if we expose detail microservice statistics to a global data center power controller, one needs to determine appropriate control granularity for performing power capping. Extremely fine-grained control (e.g., per-microservice basis) unnecessarily increases control overhead, and therefore is not suitable for microservices that exhibit extremely short request service time.

### 3 ANALYZING MICROSERVICE SYSTEMS

In this section, we investigate the topology of microservices. We show that microservice based design manifests remarkable heterogeneity. It is beneficial if one can exploit it for better performance-power design trade-offs in power-constrained data centers.

#### 3.1 Methodology

Although there are a few microservice applications online [6, 48], most of them are just toy benchmarks only containing limited microservices. Recently, Gan et al. present the first comprehensive microservice benchmark called DeathStarBench [43]. However, it is not publicly available at this moment. Thus, we conduct our case study mainly using TrainTicket [40], a railway ticketing system implemented based on microservice design principles. The number of microservices in TrainTicket is more than any other existing benchmarks [40]. It contains more than 40 microservices including 24 microservices related to business logic.

Figure 2 shows the two-layer topology of TrainTicket. The upper-level microservices in the API layer not only perform their own tasks, but also wait for the return of the lower-level microservices in the service layer. In Figure 2, we show a widely used function called Advanced Search. The red bold lines indicate its microservice region which contains many microservices. Different microservice regions in TrainTicket may interact with each other, resembling microservice systems in industrial practices. These services can be deployed on server with docker swarm [49] and kubernetes [45] for simulating a public cloud application. In this study, we deploy TrainTicket on a cluster with 6 server nodes with docker swarm. Each container only executes a single microservice. Docker swarm leverages a fair docker scheduling algorithm (round-robin) to deploy
all the related microservice dockers among server nodes. We write a
Python program to continuously access the Advanced Search service
via the 80 port of the manager node. By analyzing the request-tracing
data with Zipkin [50], we can obtain the request response time and
execution time of each microservices. We repeat our experiment for
1000 times and report the average results.

Our cluster contains 1 manager node and 4 worker nodes with
100 watts nameplate power per server. As shown in Table 2, the man-
ger node provides web interfaces for gathering timing data of each
microservice along user's request links. We deploy the observed
microservice on the power worker apart from others for debugging
specified microservice properties. Besides, the 3 normal workers
and the manager node host the remaining microservice for ensur-
ing functionality integrity of TrainTicket. We deploy the observed
microservice on the power worker apart from others for debugging
specified microservice properties.

As for hardware specifications, each server has a 6-core CPU
(Intel Xeon E5-2620, 2.4GHz) and Ubuntu 14.04 installed as the
operating system. With the advanced configuration and power inter-
faces (ACPI), their processors support operating frequencies from
1.2GHz to 2.4 GHz at the intervals of 0.1GHz. With linux tool
turbostat, we can read the dynamic power of every server. All the
servers are connected to a FAST FSG116 network switcher to ensure
high-speed network among microservices.

3.2 Basic Properties of Microservices
We begin by examining the relationship among user request, mi-
croservice region, as well as various microservices. Figure 3 presents
the distribution of execution time for each related microservice. The
x-axis is the time interval. Each colored rectangle represents the
frequency of the execution time that falls into an given time interval.
The darker the color, the higher the frequency. For a given microser-
vice in our experiment, it shows almost the same execution time
under 1000 trials and report the average results.

Even if a microservice has a short execution time, it can still be a
critical component for the entire program. A microservice may be
frequently called in the process of responding to one request. Taking
Advanced Search request for example, when a client clicks search
button, this service will return multiple records showing train infor-
mation. Meanwhile, each return calls train service more than price
since different trains between two stations usually have the same
price. Each record accesses train service for obtaining the informa-
tion of different trains, but it only calls price service in accordance
with the train types. It is possible that train service demands more
computation power. We observe that every microservice has distinct-
call times. Figure 4 shows the call times of all the microservices.
Microservices marked with red bar are called more frequently. Therefore, when determining the criticality of microservices, we should also take into account the call times of each microservice per request.

### 3.3 Performance-Power Characteristics

The criticality of a microservice in power-constrained data centers is not only related to its execution time and call times, but also its sensitiveness to performance to power capping. We enable dynamic voltage and frequency scaling (DVFS) in our experiment for power capping. We characterize the variation of microservice's execution time under different power supply conditions.

In Figure 5, we present the cumulative distributions of execution time of 4 types of microservices when responding to 1000 requests. From the figure we can see that route has short execution time which is almost irrelevant with performance scaling. In contrast, price is more sensitive with performance. Therefore, price microservice is more likely to become the bottleneck of the program than the route microservice.

By comparing Figures 5-(c) and (d), we can draw similar conclusions. For long-running microservices, some microservices (e.g., seat) are more sensitive to performance scaling. However, in terms of travel, it is hard to determine whether it is sensitive to power variation or not. Therefore, we need a metric to quantitatively evaluate the static and dynamic behaviors of different microservices.

### 3.4 The Effect on Entire Application’s QoS

From the above study we can see that the criticality of a microservice is closely related to its execution time, call times, and performance-power profile. To better understand the impact of these factors on the performance of the entire application, we further compare the performance impact of critical and non-critical microservices under power capping. Based on our offline analysis, we eventually select station, ticketinfo and travel as the critical microservices. We compare them with basic and seat, which are all among the non-critical microservices.

We separately run the selected five microservices on Server B as described in Table 2 at different frequencies of 1.8GHz and 2.4GHz. The other microservices are always hosted on Server A, C1, C2 and C3 at the frequency of 2.4GHz. We evaluate the impact on the QoS of the entire application with regards to the percentile latency and mean response time. We choose deploying TrainTicket with docker swarm’s default configuration as the baseline.

Figure 6-(a) shows the result at the frequency of 2.4GHz. It is surprising that when we isolate the critical microservices, the mean response time is higher than the baseline’s, but the percentile latency is much better. This is because isolating the critical microservice can accelerate its execution, thus lower latency. When reducing the frequency to 1.8GHz of the server running critical microservices, the latency increases a lot as shown in Figure 6-(b).

The above experiment further motivates that critical microservice can be identified with its running time, call times and its performance-power characteristics. In the following paragraph, we will discuss how to coordinate these three factors to formally evaluate the criticality of a microservice.

### 4 MICROSERVICE CRITICALITY FACTOR

In this study we propose Microservice Criticality Factor (MCF), a new metric for evaluating the priority of each microservice when allocating power.

Identifying the critical microservices is non-trivial. The reason is that our analysis on microservices shows that their criticality changes dynamically during runtime. Figure 7 shows an example of four microservices represented by different shapes. The digit on each microservice represents its execution time and the number of times the microservice appears presents its call times. Microservice a has the largest running time but its total execution time is less than microservice c which has the most running instances. When changing the running frequency from 2.4GHz to 2.0GHz, microservice c has the same total execution time as b. Therefore, it is important to find a way to compute the MCF, which synthesizes all the relevant factors.

We model the relationship of the API layer and function service layer using a bipartite graph as shown in Figure 8. The API layer is one of the disjoint and independent vertex set in the bipartite graph. The pairs of a function microservice and its corresponding database microservice constitute another vertex set. Generally, functional services are services that support specific business operations or functions, whereas a database service only maintains the private data for a function service and is not shared with any other services.

In Figure 8, each microservice’s MCF is calculated with its running time (edge weight), call times (indegree) and performance-power characteristic (variance coefficient) based on the bipartite graph model.
According to the computation mode of the influential vertex in a graph [34], we can identify the critical microservice. The execution time, call times, as well as QoS-power profile of a service are treated as static factors that can be obtained through offline profiling. The ratio of different request quantities are dynamic behaviors of a microservice. Namely, the edge weight of a microservice is the production of its execution time, call times and QoS-power relationship. Eventually, like locating the influential vertex in a graph [34], we can identify the critical microservice through comparing the production of their edge weight and indegree. Thus, the MCF of a microservice (vertex) is computed with its request ratio (indegree) and per-request execution time (weight).

\[
MCF_i = FUCN(In_i, w_i) = In_i \times W_i
\]

(1)

where

\[
W_i = call_{ts_i} \times exec_{t_i} \times \beta_i
\]

(2)

and

\[
In_i = \frac{\text{res}_j}{\sum_{i=1}^{m} \text{res}_i}
\]

(3)

With the above equations, we can calculate the MCF for different microservices varying with the incoming request.

5 SERVICEFRIDGE: MCF-DRIVEN POWER MANAGEMENT COORDINATION

Based on the MCF, we propose ServiceFridge, a differentiated power management approach by the consolidation of container orchestration and power controller. As shown in Figure 9, the key idea behind ServiceFridge is to extract the critical microservices and to strictly guarantee their QoS requirement with full power supply.

5.1 Overview

ServiceFridge has three distinctive features:

1. Cross-layer Scheduling Support. ServiceFridge establishes a cross-layer framework that spans both docker orchestration and power controller. It inserts a scheduling engine into the docker orchestration layer, which uses a MCF Calculator to compute the criticality of microservices and to classify them into different levels. Then the power controller assigns the classified microservices to server nodes with different budget.

2. Differentiated Power Management. ServiceFridge partitions a data center into a hot zone, a warm zone and a cold zone. To ensure QoS, the cold zone does not have any software power limits when running highly-critical services. Differently, the hot zone allows today’s aggressive capping schemes on low-criticality services to save power budget. The warm zone host services with uncertain criticality.
Unleashing the Scalability Potential of Power-Constrained Data Center

ICPP 2019, August 5–8, 2019, Kyoto, Japan

5.2 MCF-Driven Power Allocation

As shown in Figure 9, ServiceFridge obtains the execution time of each microservice and their call times in different microservice regions through an offline analysis. The MCF Calculator maintains a dynamic bipartite graph for capturing the varying MCF of a microservice affected by users’ behaviors. MCF Calculator initiates the bipartite graph with parameters analyzed offline. According to Equation (1), (2) and (3), the MCF of a microservice is a function value of its execution time, per-request call times, QoS-power profile and the request ratio. While some factors are static, the MCF is determined by the users’ request types and quantities as well as timely power supply. As shown in Figure 10, each vertex in the graph maintains a counter to count its present indegree. It is the sum of edges incurred by the remaining requests in former time interval and the incoming requests in current time interval. Digit surrounded by the red circles presents the completed edges in the former time intervals.

The ServiceFridge framework uses a normalized MCF to classify microservices. The MCF is normalized to the required response time of the whole application. Generally, the required response time of an application is no less than the completion time of any small microservice. The required response time of an application is an empirical value or a standard limit. It can always be determined by running some benchmarks and finding the maximum response time [47]. Nevertheless, the standard value is always proposed by a reputed organization and is pervasively accepted, such as the response times is no more than 100 ms for interactive services [21, 46]. If no power capping is used, the normalized MCFs of all the microservices are no more than 1. A larger value means more critical. When limiting the power consumed by a microservice, the MCF varies with the QoS-power relationship. If the MCF of a microservice at the lowest power states is still less than 1, MCF Calculator marks it as the low-criticality microservice. The normalized MCF of highly-critical microservices exceeds 1 even if power changes slightly.

Table 3: Evaluated power management schemes.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>Without any capping</td>
</tr>
<tr>
<td>ServiceFridge</td>
<td>The MCF-driven management</td>
</tr>
<tr>
<td>Capping</td>
<td>Managing peak power based the utilization of servers</td>
</tr>
<tr>
<td>P-first</td>
<td>Fine-grained and high-power-as-first power management</td>
</tr>
<tr>
<td>T-first</td>
<td>Fine-grained and time-driven power management</td>
</tr>
</tbody>
</table>

Table 4: Offline analysis of edge weight.

<table>
<thead>
<tr>
<th></th>
<th>ticket</th>
<th>basic</th>
<th>seat</th>
<th>travel</th>
<th>station</th>
<th>route</th>
<th>config</th>
<th>train</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET</td>
<td>A</td>
<td>12.2</td>
<td>9</td>
<td>25.7</td>
<td>22.5</td>
<td>1.3</td>
<td>1.5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4.1</td>
<td>2.8</td>
<td>0</td>
<td>0</td>
<td>1.2</td>
<td>1.4</td>
<td>0</td>
</tr>
<tr>
<td>CF</td>
<td>A</td>
<td>44</td>
<td>44</td>
<td>16</td>
<td>10</td>
<td>70</td>
<td>34</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td>A</td>
<td>536.8</td>
<td>396</td>
<td>411.2</td>
<td>225</td>
<td>91</td>
<td>51</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>8.2</td>
<td>5.6</td>
<td>0</td>
<td>0</td>
<td>2.4</td>
<td>1.4</td>
<td>0</td>
</tr>
</tbody>
</table>

The practical boundary between uncertain-criticality microservices and high-criticality ones is dynamically decided by the available power resources, which will be discussed in the next part.

5.3 Collaborative Power Management

To separately support high-/uncertain-/low-criticality microservices, ServiceFridge logically groups server nodes into a hot zone, a warm zone, and a cold zone, as shown in Figure 9. It implements a centralized controller to manage the power consumption of servers in each zone. As the cold zone hosts high-criticality microservices, server nodes always run at full speed. The controller regulates the power consumption of servers in the other two zones for limiting the total power budget via multiple power tuning knobs integrated into servers such as p-states [44], DVFS [14]. ServiceFridge uses the same power capping strategy on servers belonging to the same zone.

Since the total power of microservices changes with users’ behavior during an operating process, the power peak to be capped also varies. To fully utilize power resource, ServiceFridge always tries to maximize the average and tail response time without violating the power constraint. Therefore, one must dynamically adapt the power management to the variation of request number. Although MCF Calculator has classified the microservices into three levels, ServiceFridge Controller can promote or demote the criticality of a microservice timely based on the available power resources. For example, when the power is abundant, it can promote the uncertain-criticality microservices as high-criticality ones and enlarge the cold zone. Taking the warm zone as an example, ServiceFridge adopts an auto-scaling algorithm as shown in Algorithm 1.

The promotion or demotion of a microservice is achieved by the coordination of the container orchestration and the ServiceFridge Controller. MCF Calculator first delivers a MCF list of all the microservices to the ServiceFridge Controller. Then the controller automatically reuses the microservices classified into different levels based on the original MCF list as well as the above auto-scaling algorithm. After that, it returns the modified list to the docker orchestration and adjusts the servers in different zones. Meanwhile, it regulates the power consumption of each zone in case that their total power usage exceeds the budget power. Eventually, the docker
We randomly select 1000 requests and analyze their execution trace. We examine the MCF of the evaluated 8 microservices. We first experiment with the testbed detailed in Section 3.1. We compare our design with three types of data center power management schemes as summarized in Table 3.

Among those, Capping is a representative peak power management technique similar to prior work [14], which only scales down the overall servers’ active power to shave peak power. P-first and T-first represent a group of schemes that pay more attention on every single application or task [23, 39]. P-first is a high-power-as-first management approach. It firstly limits the power usage of power-consuming microservices. T-first slows down the execution of fast microservice to meet the power constraint.

In the following experiment, we consider two microservice regions, i.e., Advanced Search service region and Basic Ticketing service region. We use A to present Advanced Ticketing service region while B for Basic Search service region. Both A and B contain microservice ticket, basic, station and route and only A invokes microservice seat, travel, config and train. We select 8 representative microservices involved in A and B. We write Python programs to adjust the ratio of requests accessing A and B.

6.2 Analysis of Criticality

We examine the MCF of the evaluated 8 microservices. We first discuss the static and dynamic factors (defined in Section 4) determining the MCF of a microservice. Then, we analyze the impact of applying MCF to power management of different microservices.

We send requests accessing A and B for 5 minutes respectively. We randomly select 1000 requests and analyze their execution trace. In Table 4, we show the call times (CT) and average execution time (ET) of the microservices. The CT and ET of a microservice keeps constant within the same service region (as depicted in Section 3 and Section 4). W means the weight (per-request completion time) of an edge linked to a microservice. Microservices spend more time to complete the request of A. Since B does not contain service seat, travel, config and train, the corresponding values are zeros.

The MCF of microservices also varies with the incoming request types and quantities as well as the available power resources. We consider 4 different access scenarios, i.e., the ratios of requests accessing A and B are 30:0, 30:20, 20:30 and 0:30. We obtain the execution time of the services under 7 voltage/frequency settings through offline profiling. We normalize the MCF to 100ms according to Section 5.2. Figure 11 illustrates the MCF of microservices in these situations. Larger data, i.e., darker color means higher criticality. There are three rankings. The darkest black marks the highly-critical microservices. The lighter and lightest black labels the uncertain-criticality and non-criticality ones separately. We can see that a microservice can be classified into different ranking layers. For example, when the ratio of A and B transfers from 30:0 to 30:20, travel becomes a uncertain-criticality microservice from highly-critical one. Generally, the MCF of microservices decreases when the percentage of requests accessing B increases. When focusing on every single microservice, their MCF declines with the reduction of power supply. It is remarkable that the effect of power reduction on one microservice’s MCF differs from the other shown as the uneven color variation. To summarize, the MCF of microservice is determined by multiple factors particularly the dynamic factors. Taking basic as an instance, its weight is larger than the seat, nevertheless, the situation reverses when the ratio of A and B becomes 20:30.

6.3 Impact on Each microservice’s Execution

In Figure 12, 13 and 14, we show how ServiceFridge adjusts the execution state of each microservice with its MCF variation. In Figure 12, we demonstrate the operating frequency of each microservice in the above four request ratio scenarios when the overall power supply
Comparison with the Present Schemes

Finally, we compare ServiceFridge with traditional power management designs in terms of application QoS and their effects on different microservices. We first evaluate both the mean response time and percentile tail latency. In this experiment, we access both A and B with 25 paralleling workers at the same time. We observe the results under different power budget scenarios from 100% to 75% percent of the maximum required power.

Figure 15 shows our results. The y-axis presents response time normalized to the normal execution time, which is measured without any power throttling. As the power budget decreases, conventional schemes affect the mean response time as well as the 90th, 95th and 99th percentile latency. With ServiceFridge, the system can still maintain desirable service quality when the power budget is low. Compared with the other schemes, it improves the average response time by 25.2% and improves the 99th tail latency by 18.0%. This is because ServiceFridge allows the data center to trade off high power budget of non-critical microservices for better overall tail latency of the entire application.

In Figure 16, we further evaluate the impact of various power management schemes on three representative microservices. From Figure 16-(a), we can see that conventional designs such as Capping and P-first severely reduce the average response time of ticketinfo, which belongs to the highly-critical microservice set. This is because they overlook the QoS of critical services when distributing the power resource. Compared with these mechanisms, our ServiceFridge always maintains a better QoS for high-criticality microservice like ticketinfo. It limits the total power consumption by decreasing the power of non-critical microservice like station and train. Thus, in 16-(a) and (b), the mean response time of ServiceFridge is lower.

7 RELATED WORK

Microservice: In recent years, microservice software architecture is proposed to solve several problems [4, 17, 47, 48] of deploying monolithic applications in data centers. Most of them emphasize designing and implementing microservice applications [40, 41, 43] or verifying and enhancing the robustness of this software architectures itself [19, 31]. No prior works consider the power management of microservices. A few proposals have focused on using microservices to improve the performance and QoS of cloud computing service. Yu et al. focus on anticipating QoS violations in cloud settings to mitigate it performance unpredictability [41, 42]. Marcelo et al. compare the CPU and network performance of implementing microservice with docker and virtual machines [26]. These prior works mainly focus on how to deploy microservices platforms or how to enhance the performance of microservices. Only Chih-Hsun Chou et al. [8] propose a power-saving mechanism under the architecture of microservice through prolonging the execution of microservices to its maximum time limit.

Data Center Power Management: There have been substantial works related to power management in an power-constrained data center. To ensure that the power dissipation stays below a given budget, aggressive power control strategies such as power/performance...
state tuning [12, 15, 36], thermal/cooling optimizing [35, 38] and battery-based peak power shaving [12, 25, 30] are employed. Performance-preserving aggressive power capping framework has been deployed in the industry [28]. However, current works focus on data center power management at the server level. They are coarse-grained and insufficient for managing the power for microservices. Some works focus on managing data center’s peak power at fine-grained granularity [39]. However, these works are mainly history-data-driven, making them too slow to allocating power at per microservice level. Criticality and Latency Analysis: There have been prior works on recognizing the critical path [16, 32, 33] in a process pipeline to enhance the performance of a system. For example, Srinivasan et al. [32] define an alternative measurement of the critical path for characterizing performance of memory system. Tune et al. [16] provide the first exploration of heuristics-based critical path predictors. Different from these prior works, our work focuses on identifying the critical microservices at service level. There also exists some works on improving the tail latency of cloud service. Adrenaline [9] identifies latency-critical operations in an application and boosts their computation. However, it cannot evaluate the criticality in quantity. Other proposals like Rubik [18] and Pegasus [13] permit adjusting frequency every few seconds to keep tail latency on server level. They are insufficient in data center environments.

8 CONCLUSIONS

Unleashing the power saving potential of microservices allows data center to better scale out. In this paper, we propose to adapt existing power management schemes to OLDI applications with the emerging microservice architecture. We show that by taking into account the heterogeneity of microservices and offering a differentiated power capping service, one can greatly save power capacity in a power constrained data centers, with minor design overhead.

ACKNOWLEDGMENTS

We thank all the reviewers for their valuable comments and feedbacks. This work is sponsored by the National Natural Science Foundation of China (No. 61502302). Corresponding author is Chao Li from Shanghai Jiao Tong University.

REFERENCES

data
[27] P. Pan et al. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In ICCD.
[38] X. Gao et al. 2018. Reduced cooling redundancy: A new security vulnerability in a hot data center. In NDSIS.