Improving the Efficiency of Serverless Computing via Core-Level Power Management

Du Liu, Jing Wang, Xinkai Wang, Chao Li*, Lu Zhang, Xiaofeng Hou, Xiaoxiang Shi, Minyi Guo
Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: {sjtu_ld, jing618, unbreakablewxk, luzhang, lambda7shi}@sjtu.edu.cn, {lichao, hou-xf, guo-my}@cs.sjtu.edu.cn

***Abstract*—Serverless computing has recently become a significant application paradigm in data centers. However, existing power management methods focus on optimizations at the coarse-grained server level, making them unable to handle the characteristics of these short-lived, dynamic serverless functions. In this context, the unawareness of function-level characteristics by the existing power management systems can severely degrade the energy efficiency of the data centers. To address this challenge, we design a function-level power management system. Instead of relying on server-level schedulers, we propose a novel core-level scheduling policy for serverless functions that can efficiently allocate functions to the most suitable CPU core. Additionally, we propose a power management mechanism for serverless computing that can reduce system power consumption with functions' QoS guaranteed. Our evaluation shows that our system achieves a maximum power saving of 8.5% and an average power saving of 8% across the majority of loads without incurring any loss in tail latency, as compared to the conventional server-level scheduling system.**

***Index Terms*—serverless computing, power efficiency, power management, core-level scheduling**

I. Introduction

Energy efficiency is one of the most critical metrics for data centers in addition to computing performance. Energy consumption is reported to account for 25% $\sim$ 40% of the total cost of ownership (TCO) of a data center [1]. Even incremental energy efficiency improvement can bring huge cost savings in a cloud data center. In addition, achieving zero carbon emissions and carbon neutrality is a global mandate under the Paris Agreement Roadmap [2], [3]. All these incentives have led researchers to work towards improving the energy efficiency of data centers. Cloud companies also have a strong desire to limit system power consumption to cut down energy bills and achieve global environmental goals. Generally, there are two ways to achieve this goal: increasing utilization and making servers energy-proportional [4]. The first way is to increase workload consolidation by co-locating different workloads which can be divided into two representative classes: latency-critical and best-effort. The other way is to make data center servers energy-proportional, i.e., scale the servers' power consumption to match the instantaneous load.

Serverless computing has emerged as a significant paradigm in the domain of cloud computing, which garners considerable attention from both academia and industry, such as OpenWhisk [5], OpenFaaS [6], Amazon Lambda [7], Google Cloud Functions [8], and Microsoft Azure Functions [9]. In contrast to the conventional cloud computing paradigm, serverless computing conceals the underlying infrastructure and platform from the client. Rather than requiring customers to lease virtual machines to execute their applications, this paradigm enables them to focus solely on the functions necessary for their applications instead of the infrastructure. The service provider is responsible for scheduling these functions in an efficient and auto-scaling manner, while also guaranteeing the clients' quality of service (QoS).

However, previous work mainly focuses on improving serverless computing performance [10]–[12]. Very limited work has been done to optimize the serverless system's energy efficiency with clients' QoS guaranteed. Functions are assigned to worker nodes or servers as opposed to individual cores in current serverless systems [13]. Most serverless systems delegate scheduling to the process scheduler of the OS such as Linux's Completely Fair Scheduler. We call it the scheduling policy of intra-node processor sharing [10]. Given the power budget, the scheduler co-locates functions together and assigns CPU resources according to the calculated maximum execution time. When the process reaches its maximum execution time, the scheduler will stop the task and reschedule it. Such coarse-grain policy results in the sharing of physical and virtual resources among function invocations, thereby causing performance variability.

In addition, it's non-trivial to achieve function-level performance tuning under the server-level scheduling policy. Most existing power management methods [4], [14] depend on customized performance tuning for each application with mechanisms like Dynamic Voltage and Frequency Scaling (DVFS). In the context of serverless computing which adopts the policy of intra-node process sharing, we cannot build our power management methods based on these mechanisms since the scheduling system has no idea which core the function runs on when we want to adjust its performance.

In this work, we take the first step to schedule serverless functions in a core-level way to excavate the potential of fine-tuning power management in serverless environments. We delve into the power and performance features of different functions and train a linear regression model with high accuracy to predict their execution time under different function inputs. We define priority for the invoked functions to maintain better core allocation with functions' QoS guaranteed. Furthermore, we introduce a serverless computing scheduling system

*Corresponding author

with core-level resource binding for each function. Overall, our work provides a better trade-off of power and performance by fully considering function behavior and resource conditions. Through extensive evaluation of realistic prototypes, we show that our designs can make the FaaS system more sustainable by reducing power consumption by 8% on average without harming the tail latency performance.

In summary, this paper makes the following contributions:

- We build a QoS-aware request queue to prioritize the function requests with profiled function characterization, including function latency prediction and request prioritization with high accuracy and efficiency.
- We propose a novel core-level serverless scheduling system, that can manage functions in a fine-grained way. Furthermore, we design a frequency adaption mechanism for dynamic trade-offs of serverless function performance and power consumption with QoS guaranteed.
- We conduct a real system experiment to evaluate the effectiveness of our core-level scheduler. Our system achieved 8.5% maximum power saving and 8% average power saving across most loads while maintaining the functions' tail latency within QoS constraints compared to the server-level one.

The rest of this paper is organized as follows: Section II provides background and current challenges. Section III presents function characterization. Section IV proposes the system design. SectionV specifies the implementation details of the system. Section VI describes experimental methodologies and presents experimental results. Section VII introduces related work. Section VIII concludes the paper.

II. Background & Challenge

In this section, we introduce the conventional server-level serverless function management system and further specify core-level optimization challenges.

A. Server-Level Serverless Function Management

Current serverless computing systems adopt the server-level function management method as shown in Fig. 1. The function requests from the clients are scheduled by the server-level function scheduler. There are three primary strategies for the scheduler [10]: Shortest-Remaining-Processing-Time (SRPT), Processor Sharing (PS), and First-Come-First-Serve (FCFS). Simulation experiments have shown that PS outperforms the other strategies, making it the most effective server-level scheduling strategy. As a result, both in academia and industry, PS has become the preferred strategy for server-level scheduling in the majority of serverless computing platforms. In the context of PS, each function receives an equal share of the processor's capacity, which does not apply to real systems. Therefore, most serverless platforms end up using Linux's Completely Fair Scheduler (CFS) as an approximation of PS. As a result, functions that run as containers will be automatically allocated computing resources by CFS.

In detail, when a container process begins to be scheduled by CFS, CFS calculates its virtual runtime, which is the number of CPU time slices the container has used. Then, CFS inserts the container into the red-black tree scheduler and sorts it according to its virtual runtime. CFS chooses the container with the lowest virtual runtime to run and inserts it at the end of the red-black tree. When the container is running, it uses CPU time slices, and its virtual runtime increases. When the container finishes running and relinquishes the CPU, CFS recalculates the container's virtual runtime and reinserts it into the red-black tree, until all processes finish.

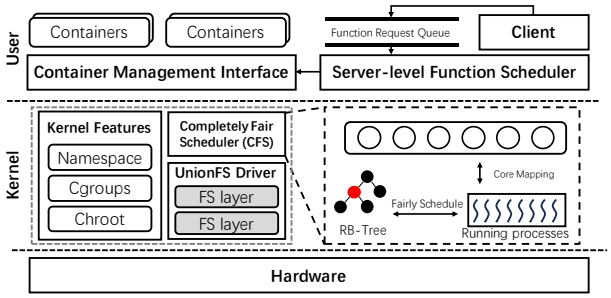

Fig. 1. An overview of the server-level serverless computing system.

However, during the scheduling process, prevailing serverless computing systems do not explicitly delegate cores for each function, making it difficult to determine which core the function is actually running on. It is impossible to achieve per-function performance tuning in this situation. Thus, it is effective to simply adopt PS in server-level scheduling but not optimal. If we want to achieve maximal energy efficiency, we need to step further making scheduling more granular. Our research explores optimizations along with this idea.

B. Challenges of Core-Level Optimizations

Challenge 1: The complexity of co-located serverless functions makes it challenging to seize the core-level efficiency opportunities. Although operating system abstractions can expose power capping and provide fine-grained control mechanisms, such as power containers [15] and sandboxes [16], [17], existing works mainly leverage straight-forward methods, e.g. Linux CFS [18], to schedule the large amounts of arrived functions with extremely short execution time. They often adopt server-level scheduling policies, which assign available sockets (instead of physical or logical CPU cores) to a batch of arrived functions. Furthermore, existing works pay more attention to function interference handling when assigning them to the same socket. For example, Kaffes [13] proposes a centralized and core-level scheduler to reduce resource interference of functions, while missing the precious opportunity of saving power.

Challenge 2: Core-level power management cannot be directly implemented in current serverless computing platforms. Power monitoring at the hardware core level is difficult to configure, as it requires hardware support in computer systems. The frequency of a CPU core is an effective indicator of its power consumption because they are positively correlated. However, when the fair method is applied to scheduling

TABLE I
S,M,L INPUTS FOR EACH FUNCTION

Function	Input	Small	Medium	Large
Compression	file size	500KB	1000KB	1500 KB
Chameleon	# rows and columns	100,100	200, 200	300, 300
Download	file size	25MB	50 MB	75MB
Upload	file size	25MB	50 MB	75MB

functions, it's hard to indicate functions' power consumption simply by frequency, since a function can be assigned to any core in the server and we cannot locate which core it runs on. Therefore, the development of a robust core-level scheduling mechanism is crucial to enable effective serverless power management.

Summary: We seek to design a fair serverless function core-level scheduling algorithm and implement it with a QoS-aware CPU frequency tuning mechanism to achieve power saving with QoS guaranteed.

III. Preliminary of Function Characterization

In this section, we characterize four representative functions to find underlying power-saving chances for serverless computing. The functions are selected from typical serverless benchmarks [19]–[21]. The functionalities of them can be referred to in Table II. Characterizing the behaviors of functions under certain CPU settings becomes crucial. The relationship between power consumption and performance can guide the performance decline within QoS constraints. We manipulate various aspects of the execution environment to characterize the performance of functions by assigning different numbers of CPU cores, and CPU frequencies. We plot the curves of serverless execution latency with different scales of inputs and assigned CPU cores over growing CPU frequency from 0.8GHz to 2.2GHz. Detailed information about their settings is presented in Table I.

A. Analysis on Core Allocation

Overall, we observe that all the serverless functions maintain a performance improvement when increasing the CPU frequency. However, different function types perform slightly different trends as CPU frequency changes. One can make a better trade-off between the CPU frequency and execution runtime. In addition, the normalized latency curves of the same input but different core numbers overlap completely in Fig. 2-(a) and (b). This tells us that functions have the same trend under various CPU cores quantity. This phenomenon suggests that these functions show insensitivity to the number of CPU cores, in which case there is no need to allocate additional core resources. Primarily single-threaded mode is sufficient. Generally, serverless functions maintain few intra-function parallelisms. In the philosophy of serverless computing, developers are encouraged to implement parallelism between different functions rather than in an individual function [13].

Observation 1: Functions hardly suffer performance loss when assigned to only a single CPU core due to limited intra-function parallelism.

B. Analysis on Various Function Inputs

Known from Fig. 2 that the performance of functions is improved with the increase of CPU frequency, we further normalize the performance by performance on 2.2GHz with single-core configuration. We plot the variation of the normalized execution latency with different scales of inputs, as shown in Fig. 3. It shows that each function has similar normalized performance trends over different function inputs. This means that we can build a certain mapping between normalized latency and allocated CPU frequency. Then we can further predict the overall latency of the function by analyzing inputs. Furthermore, different functions exhibit unique performance curves. For example, the execution latency of `Compression` at 1.4GHz rises to about 150% of that at 2.2GHz, while the execution latency of `Upload` at 1.4GHz rises to just 120% of that at 2.2GHz. Thus, distinct tuning strategies are necessary for different functions.

Observation 2: Different functions have specific mappings of normalized latency and allocated frequency, which remain consistent across diverse inputs.

C. Analysis on Function Co-location

Core-granular scheduling involves two basic core-binding approaches: allocating individual functions to unique cores to mitigate scheduling delays caused by time-sharing and underlying last-level cache pollution and packing multiple functions to a single core to attain higher utilization. To determine the appropriate core-binding strategy for a given function, it is necessary to evaluate the performance behavior of all functions under co-location. In Fig. 4, it indicates that CPU-intensive functions such as `Compression` and `Chameleon` tend to fully utilize the available CPU time, thereby making co-location of these functions ineffective in raising utilization. Conversely, functions like `Download` and `Upload`, which do not heavily rely on CPU usage, have the potential to effectively increase core utilization through co-location.

Observation 3: We need to co-locate non-CPU-intensive functions to leverage available CPU resources and execute CPU-intensive functions independently by assigning unique CPU cores to maintain QoS.

IV. System Design

In this section, we propose our core-level power management mechanism which is orthogonal to the prevailing system. Therefore, we can implement our mechanism to most intra-node serverless systems to enable power efficiency. The key contribution of our work is to design a fair serverless function core-level scheduling algorithm and implement it with a QoS-aware CPU frequency tuning mechanism to achieve power saving with QoS guaranteed.

Our system consists of four key components: (1) function level tuning table, (2) function latency predictor, (3) QoS-aware request queue, and (4) core-level scheduler. Fig. 5 shows the architecture of our system in detail. Leveraging serverless function features, we organize function requests in a QoS-aware manner. The scheduler is responsible for handling

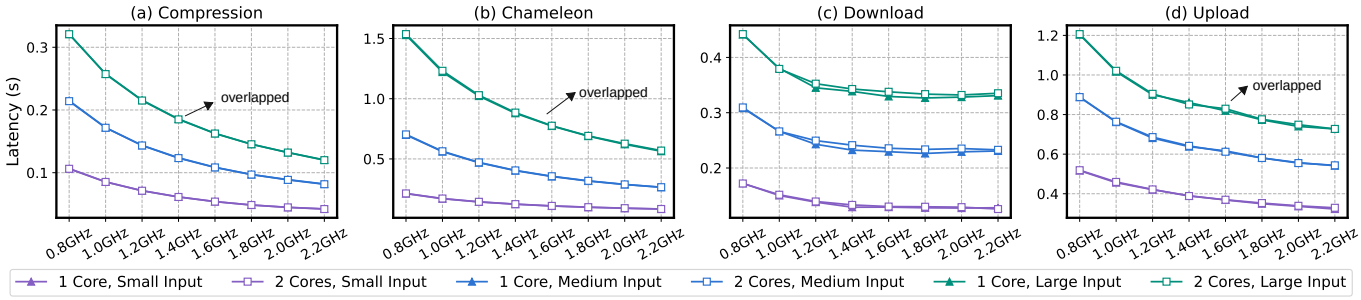

Fig. 2. Execution latency of evaluated functions on growing CPU frequency.

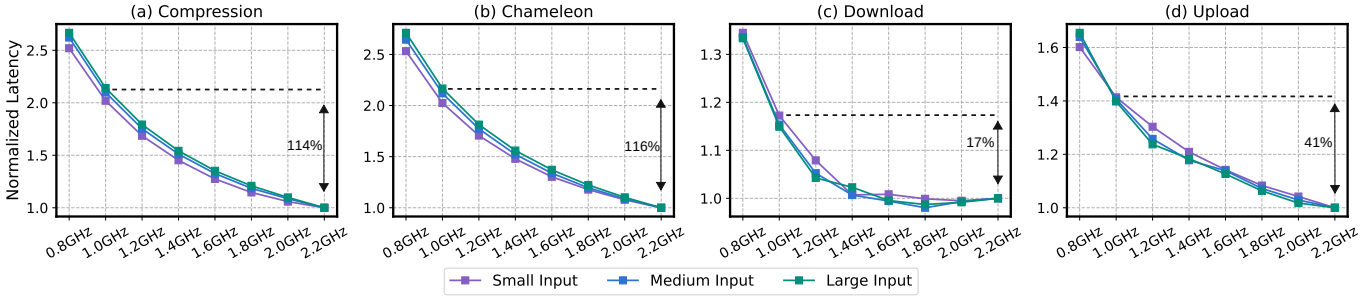

Fig. 3. Normalized latency of evaluated functions on growing CPU frequency.

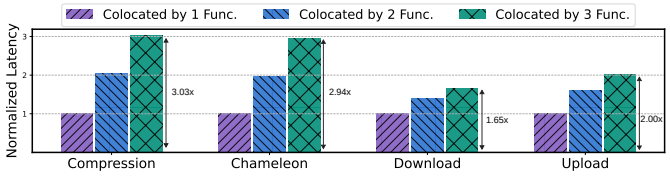

Fig. 4. Performance comparison of co-location serverless functions.

requests and assigning functions to suitable containers. During the scheduling process, adjustments to the CPU frequency are made to save power, while also ensuring that QoS requirements are met.

A. System Overview

The overall structure of our system is shown in Fig 5. Before scheduling, we need to do profiling for new functions to learn functions' performance features. A function predictor will be trained to estimate the latency of each function under different CPU frequencies with the help of profiling information. The process of training the latency prediction model and function profiling is conducted offline.

Once a request is received, it is checked if it has been previously profiled. If it has not, the request is sent to the function profiler and executed on an exclusive CPU core with the highest frequency until profiling is complete. If the request has already been profiled, it is directly sent to the latency predictor for estimation of its execution time.

The request is then added to the QoS-aware request queue, where it is dispatched to the scheduler based on priority. The core-level scheduler manages the requests and allocates CPU resources to functions by binding CPU cores to proper containers. The frequency estimator determines the feasible CPU frequency by referring to the function level tuning table. The frequency tuner then applies the setting to the corresponding CPU core. The core-level status record keeps track of the container status to guide the container allocator. Synchronously, the state monitor records the list of functions being executed and the length of the request queue and determines the busy or idle state of the workload based on current resource pressure. This procedure enables dynamic and accurate power management while guaranteeing QoS.

B. Function Level Tuning Table

A smart scheduler and power tuning module require additional knowledge of function requests, including execution time and performance features of each function, which is kept track of by **function level tuning table (FLTT)**. The function profiler is responsible for profiling new functions. Specifically, we execute functions at different CPU frequencies to obtain a normalized performance curve for each type of function. Thus we can build a mapping of the CPU frequency and the normalized overall execution latency, which helps the actual latency prediction.

We also analyze the function behavior by collecting the CPU utilization rate when co-location. The CPU utilization rate of the container is used to define a function's quota, which helps to make the decision of assigning cores. Those functions with high quotas can monopolize a core while functions with low quotas will co-locate with other functions per core. All these

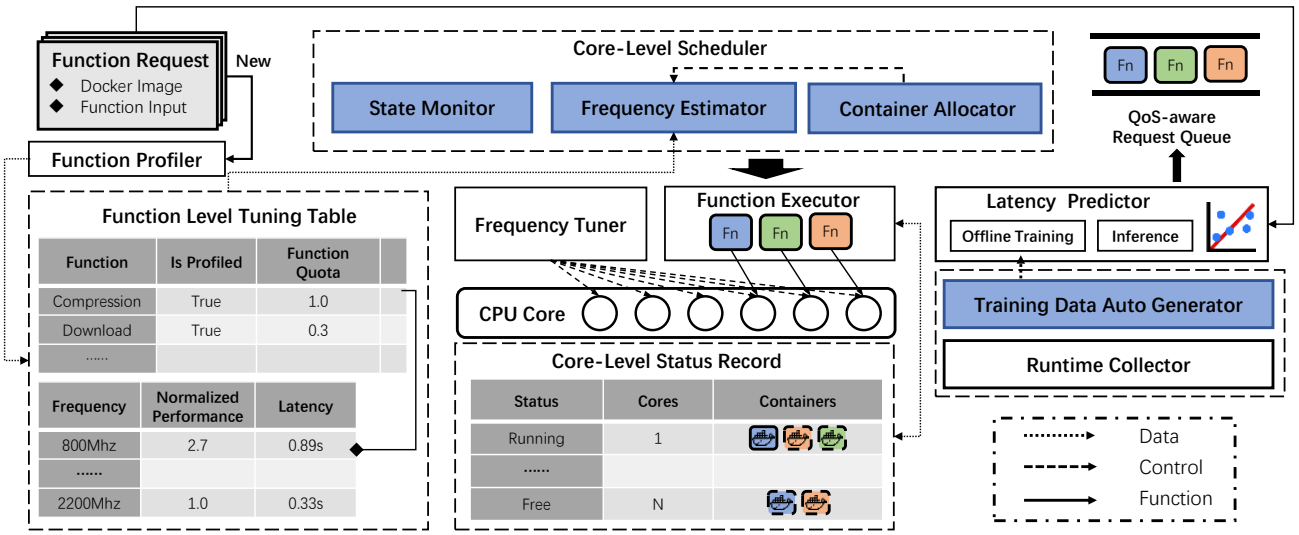

Fig. 5. Overall structure of the core-level serverless scheduling system.

profiling results will be updated in the table, which provides useful knowledge for core-level scheduling.

C. Function Latency Predictor

The **function latency predictor** estimates function execution time according to a specified set of inputs, differing among each type of function. We design an algorithm that can estimate latency accurately in a low inference overhead.

Obtaining training data is the primary step in developing a predicting model. By referring to several representative serverless function benchmarks [19], [20], we classify the input of the functions into two types, numeric input and composite input. For example, in the case of the function `chameleon` used for HTML/XML table generation, the input parameters are numeric values representing the number of rows and columns in the HTML table. On the other hand, functions used for image processing typically take an image as input, which can be seen as an aggregate of several attributes such as pixel height and width.

When dealing with functions that have numeric inputs, it is straightforward to define the training data format. When dealing with functions that have composite inputs, simply obtaining properties such as file size, which are directly available, is often not sufficient to train a highly accurate model. Therefore, it's necessary for us to define the features of some composite inputs that are prevalent in the production environment where our system will be deployed. For example, we can define pixel height, pixel width, image size, and image format as features of the image.

To obtain training data, a common approach is to collect data during the execution of the functions. Based on observations from the characterization of the FaaS workload of Azure Functions [22], we learn that 18.6% functions generate 99.6% invocations, which means that FaaS workloads follow the Pareto principle in the production environment. For the frequently invoked functions, training data will accumulate quickly to train the predicting model so that the system can turn on performance tuning on these popular functions. For the rarely invoked functions, the power savings of our system won't be significantly reduced since they bring a limited number of function invocations.

In particular, we add a **training data auto generator** in our system to speed up training data collecting for functions with numeric inputs. It will autonomously generate a range of function inputs and run functions offline to obtain data for latency predictor to train the model. We use the linear regression model as our latency prediction model and adopt the R^2 score (a coefficient of determination) and Root Mean Square Error (RMSE) as metrics to evaluate our models. The definition of these two metrics is shown in Equ. 1.

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2} \tag{1a}$$

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2} \tag{1b}$$

Generally, we train specific models for different types of serverless functions so that the latency can be quickly inferred when scheduling. The training overhead of each function is acceptable, and we show the results in our later evaluation.

The latency predictor is convenient to infer an estimated result, i.e. the latency at the highest frequency setting. We predict the actual execution time under the maximum CPU frequency as the basic normalized latency. Then we can calculate any actual latency on each CPU frequency by referring to FLTT. This step can significantly reduce profiling overhead, especially when latency prediction inference is a critical path of scheduling.

Algorithm 1: Core-Level Scheduling Process

Input: Fair request queue: Q, Free core list: free_core_list

```
1  is_busy ← False;
2  executed_num ← 0;
3  while True do
4      if Q is empty() then
5          continue;
6      r ← Dequeue(Q);
7      cont ← FindFreeContainer(r.function);
8      if r.func is CPU-intensive then
9          if cont.core is not in free_core_list then
10             cont.core ← free_core_list.pop();
11             UpdateCore(cont, cont.core);
               // Core Rebinding
12         if executed_num + Q.length + r.quota ≤ free_core_list.length then
13             is_busy ← False;
14             SetFrequency(cont.core, QoS2Freq(r.RR, QoS));
15     else
16         if cont.core.quota + r.quota ≤ 1 then
17             cont.core.quota ← cont.core.quota + r.quota;
18         else
19             cont.core ← free_core_list.pop();
20             cont.core.quota ← r.quota;
21             UpdateCore(cont, cont.core);
22         SetFrequency(cont.core, max(QoS2Freq(r.RR, QoS),
             cont.core.freq));
23     executed_num ← executed_num + r.quota;
24     cont.send_request(r.inputs);
25     executed_num ← executed_num - r.quota;
26     if r.func is CPU-intensive then
27         free_core_list.add(cont.core);
28     else
29         if cont.core.quota == 0 then
30             free_core_list.add(cont.core);
31     if executed_num + Q.length + r.quota > free_core_list.length and not
         is_busy then
32         is_busy ← True;
33         SetFrequency(all_cores, 2200);
```

D. QoS-Aware Request Queue

In the scenario of core-level scheduling, if all cores are occupied by invoked functions, any subsequent requests must be placed in a queue and await the release of a free core before being serviced. We design a **QoS-aware request queue** to handle upcoming requests and fairly dispatch them to the core-level scheduler.

Our priority design of the request queue depends on the waiting time and service time. The waiting time can be collected by detecting hardware information, such as the real-time value of the process collected by time tools in Linux OS. The service time is equal to the estimated latency in our latency predictor. We use a metric that is Response Ratio (RR) to represent the priority of each task, as shown in Equ. 2. The design is inspired by the Highest Response Ratio Next (HRRN) scheduling algorithm [23].

$$\mathrm{RR} = \frac{\text{Waiting Time} + \text{Service Time}}{\text{Service Time}} \tag{2a}$$

Generally, the request with a higher response ratio(RR) will be prioritized. If two requests have the same waiting time, the request with a shorter service time will be prioritized. If two requests have the same service time, the request with a longer wait time will be prioritized. This method combines the advantage of Shortest Job First and First Come First Serve algorithms to fairly handle both short tasks and long tasks.

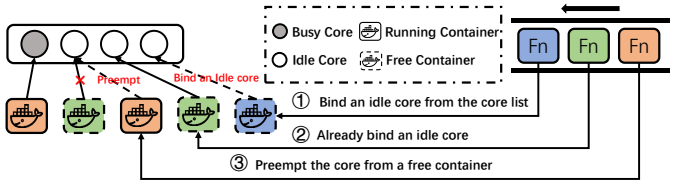

Fig. 6. Cases of container selection.

E. Core-Level Scheduler

The scheduler receives requests from the QoS-aware request queue and allocates resources for each function request in a core-granular manner. Furthermore, it performs function-level frequency adaption to trade-off between power saving and acceptable performance within QoS constraints. The algorithm of the scheduling procedure is shown in Alg.1. The major components of the scheduler are illustrated as follows.

The **container allocator** takes responsibility for allocating functions containers and performs core binding if needed. To make the allocation more efficient and avoid unnecessary core binding, we design the **core-level status record (CLSR)** to maintain lists of containers for each core and mark whether cores are occupied or not. When a request is popped from the QoS-aware request queue, the container allocator will find a free container by referring to CLSR. To explain the rule of the container selection(Line 7-22), we demonstrate three cases in detail as shown in Fig 6.

1) If the CPU cores bound by an available idle container are all in use and there are still free CPU cores not bound by other containers, the container allocator will bind an idle container to that CPU core and allocate the function to it.
2) If there is a CPU core bound by an available idle container that is free, the container allocator will directly allocate the function to it.
3) If the CPU cores bound by an available idle container are all in use and there are no CPU cores that have not been bound to a container, the container allocator will preempt a core from other functions, bind an idle container to it, and run the function.

After the container and its bound core are both confirmed, we need to update the remaining resources of it, if the function is non-CPU-intensive. We define a function's quota by CPU utilization rate which is profiled by `docker stats` (Line 17). The **frequency estimator** determines the best CPU frequency by referring to the tuning table. We define the best degradation rate (BDR) in Equ. 3, which indicates the performance degradation that the current function can afford.

$$\mathrm{BDR} = \frac{\mathrm{QoS}}{\text{Service Time}} - \frac{\text{Waiting Time}}{\text{Service Time}} \tag{3a}$$

$$= \mathrm{QoS\ Ratio} - \mathrm{RR} + 1 \tag{3b}$$

The corresponding frequency can be found by searching from the lowest frequency to the highest frequency in FLTT until you find a frequency under which the normalized performance is greater than BDR. If the core is shared by several functions. the frequency adaptor selects the highest frequency among them (Line 22).

The **state monitor** collects state information from the QoS-aware request queue and scheduler at regular intervals and it checks whether functions that are being executed or in the queue exceed the limit that the server can handle. Usually, the system does not need to run processes in the highest frequency, which we name as the *idle mode*. If there are too many waiting tasks in the queue, the system will be switched to the *busy mode* in which all cores work at the highest CPU frequency (Line 31-33).

V. Implementation Details

We implement a proof-of-concept system to show the effectiveness of our system. The implementation is mainly written in Python, which includes the base scheduler, the core-level scheduler, the power management module, and the container management module. Each component is independently developed. Besides, all functions selected are modified to the format defined by our system. Thus, they can be easily invoked with different input arguments.

We use Docker containers as functions' executing environments. Similar to the watchdog architecture of OpenFaaS [6], we implement an HTTP server using Flask, which is built into the container image as an initialization process. It provides a common interface between the external environment and functions' functionality, enabling the analysis of HTTP requests received on the API gateway and the execution of binary files to invoke corresponding functions. When a container starts, it will init the server and expose a port, which will be managed by the scheduling system. Through this port, the system can monitor and query the running status of the container and process new function requests. Besides, we also integrate a remote storage API inside the container image, since some functions might need to upload data to the remote storage or download data from the remote storage. We implement a Minio [24] storage system locally in the container to avoid the impact of network fluctuations on the results.

VI. Evaluation

In this section, we evaluate our design. Specifically, we want to answer three questions:

1) How does our design improve the power efficiency of serverless computing systems?
2) How does our design affect the function performance?
3) How is the overhead and accuracy of our introduced optimization modules?

A. Experimental Environment

Experimental Setup. The evaluation of our system was conducted on an Intel Xeon Silver 4114 platform, comprising 2 sockets, each with 10 cores, and 64GB DDR4 memory, running on Ubuntu 16.04.5 LTS. The processor supports per-core DVFS with operating frequencies from 0.8GHz to 2.2GHz at an interval of 0.1GHz. The frequency driver is ACPI with the "userspace" governor.

TABLE II
Evalauted Functions

Function	Description	Benchmark
chameleon	Render HTML/XML file	FB
linpack	Run linpack benchmark	FB
json dump	Deserialize and serialize json file	FB
upload	Upload to the remote storage	FB
download	Download from the remote storage	FB
dynamic HTML	Render templates by jinja2	SeBS
compression	Run file compression	SeBS
bfs	Run breadth-first search algorithm	SeBS
image resize	Resize a image into the thumbnail	SeBS
DNA visualization	Process DNA sequence data	SeBS

TABLE III
Evalauted Systems

Systems	Scheduling Method	PM
PSS (Baseline)	First In First Processing; server-level	No
CS w/o PM	Prediction-based HRRN; core-level	No
CS (Ours)	Prediction-based HRRN; core-level	Yes

System Configuration. We conduct our experiments by running our monitor and scheduling system on socket 0. We keep function containers on socket 1 to evaluate their energy consumption accurately. We use pyRAPL [25] to measure the entire energy consumption, including the energy consumption of the CPU socket package and DRAM, to reveal all functions' energy consumption. We disable Hyper-Threading to evaluate our system accurately due to the per-physical-core frequency adjustment instead of the per-logical-core basis.

Benchmark and Metrics. The evaluated serverless functions are shown in Table II, which are selected from SeBS [19], FunctionBench (FB) [20]. We develop an open-loop load generator capable of emulating the fluctuations of the coming requests, including both the high-traffic periods (peaks) and the low-traffic intervals (valleys). We vary the load scales by controlling the requests per second (RPS) from 4 to 10. Furthermore, we monitor the 95th percentile latency (p95 latency) of functions and present them in the results.

Baselines. We evaluate the effectiveness of our Core-level System (CS) compared with the baseline Processor Sharing Scheduling (PSS). In practice, this policy can be implemented by delegating CFS to perform intra-node scheduling, which is adopted by most serverless systems [10]. Besides, we also compare the system with our Core-level Scheduler without Power Management (CS w/o PM). The detail of evaluated systems is shown in Table III.

B. Result of System Performance

Evaluation of QoS Guarantee. We first present the mean and tail latency of each application that is under the QoS limitation. As shown in Fig. 7, CS with no power management

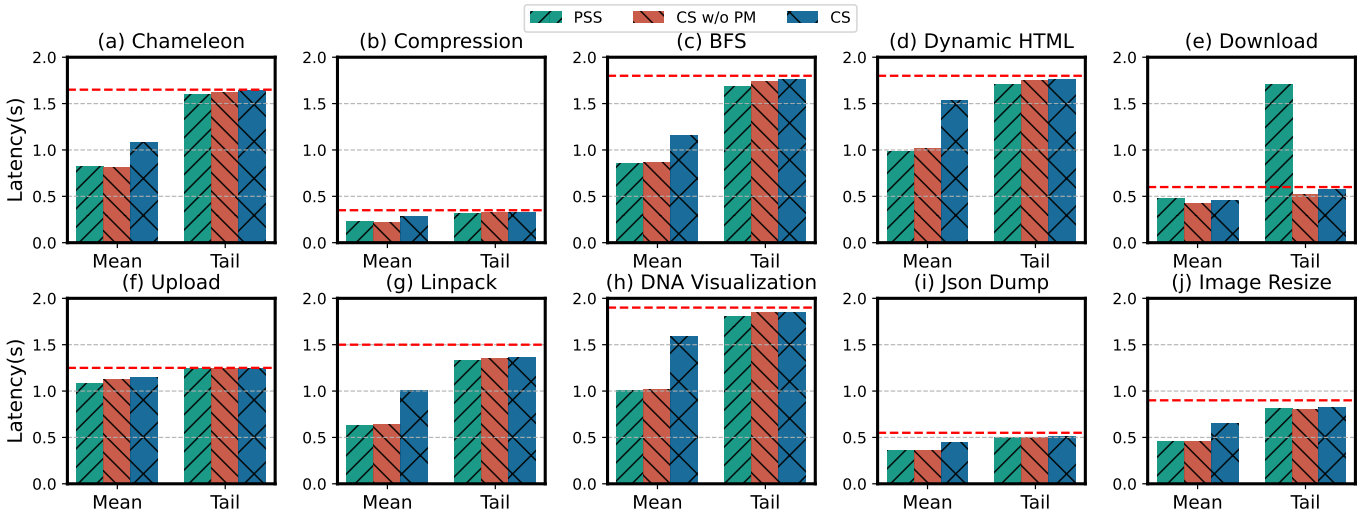

Fig. 7. Mean and tail latency under different scheduling methods. The red dashed lines mean QoS constraint.

TABLE IV
THE OVERHEAD AND ACCURACY OF THE LATENCY PREDICTOR

Function	Overhead		Accuracy	
	Training	Inference	R^2	RMSE
Upload	2.0 ms	0.19 ms	0.988	0.028
Download	2.0 ms	0.19 ms	0.988	0.023
Chameleon	3.8 ms	0.19 ms	0.894	0.111
BFS	2.2 ms	0.18 ms	0.998	0.007
Compression	2.9 ms	0.18 ms	0.999	0.001
Dynamic HTML	1.2 ms	0.18 ms	0.999	0.012
Linpack	2.1 ms	0.20 ms	0.996	0.045
Json dump	2.1 ms	0.18 ms	0.995	0.010
Image resize	4.2 ms	0.25 ms	0.960	0.021
DNA visualization	2.3 ms	0.19 ms	0.997	0.009

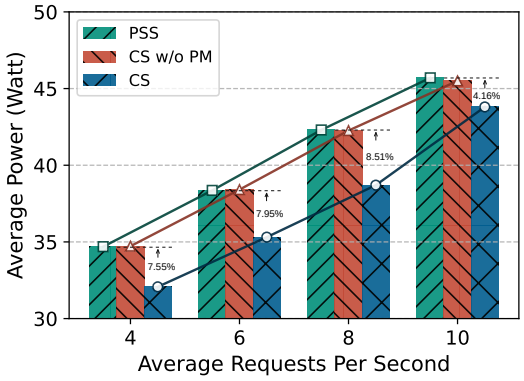

Fig. 8. The average power consumption comparison on evaluated systems with different scales of deployed functions.

and PSS have the same performance on both the mean latency and tail latency in most of the functions except `Download`, which suffers from severe degradation in PSS. This result reveals that CS introduces less or similar scheduling overhead than PSS and to some extent mitigates resource contention due to its core-binding mechanism. After adding power management back to CS, functions' QoS is still met. Compared with PSS, CS has higher mean latency since it can perform dynamic performance tuning for each function, which causes degradation within QoS constraints to save power.

Performance of Latency Predictor. The experiment shows that linear regression achieves great predicting accuracy with low training and inference overhead, which fulfills the requirements of our system. We evaluate the 10 types of serverless functions and present the detailed training inputs, the predictor runtime, and execution accuracy. The result is shown in Table IV. Our method performs high accuracy of latency prediction, which helps to determine the priority of requests in a sophisticated way. The extra overhead including training duration and inference runtime is acceptable in our scheduling system. In addition, We also conduct comparative experiments on the random forest model, which achieves a similar predicting performance with 2.1x-3x inference latency and 73x-91x training latency.

C. Result of Power Efficiency

Result of Power Saving. To evaluate our system's scalability, we set the average RPS of the load generator from 4 to 10 to simulate different scales of workload. As shown in Fig. 8, The power behavior between PSS and CS without PM is the same as the performance behavior in section VI-B. This result is expected since core-level scheduling itself has no ability to save power. With power management implemented, CS achieves an average of 8% power saving under the most workload. we notice that CS only saves 4% power when RPS is set to 10. This is because when the load on the server starts to approach its tolerance limit, CS will also increase the ratio of time the CPU works at the highest frequency to cope with excessive requests. If the load level reaches or exceeds the

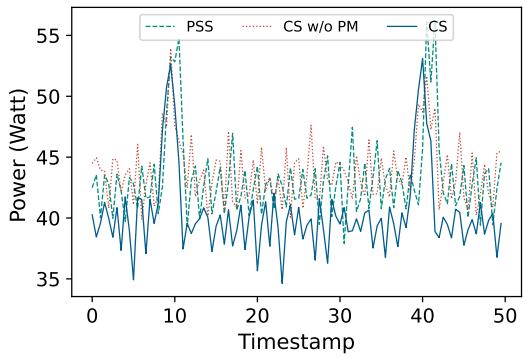

Fig. 9. The real-time power consumption comparison of functions on each timestamp on evaluated systems.

threshold, The power of CS and PSS will converge. In fact, this situation should be avoided by the inter-node scheduler that takes charge of load balancing. Thus, our system achieves significant power saving in general cases.

Efficiency of Power Adaption. In Fig. 9, we monitor the real-time power consumption of evaluated systems under the same workload. When the request peak occurs, the power of all systems rises dramatically. This situation reveals that our system remains sensitive to load fluctuations. When the load level decreases, it's observed that CS runs at lower power than CS without PM and PSS, which indicates that the power management mechanism turns our system to a power-saving mode that functions are slower while with QoS guaranteed.

VII. Related Work

Task Co-location. Generally, typical scheduling methods tend to improve resource utilization and accelerate application execution to improve efficiency. First, co-locating best-effort (BE) workloads with latency-sensitive (LS) workloads can improve energy efficiency by increasing resource utilization [26], [27]. Second, dealing with resource interference can improve resource utilization by accelerating function execution [28], [29]. When co-locating applications, the contention of resources including CPU, memory, cache, and bandwidth can affect the performance of LS loads.

Some previous works propose various non-partitioning methods to co-locate workloads [30]–[32]. They generally share the same idea of co-locating workloads that don't contend for the same hardware resources. However, the primary concern lies in the substantial decrease in the variety and quantity of workloads that can be co-located due to active inter-job interference. Thus, the QoS cannot be fully guaranteed for co-located LS workloads.

In addition, many works propose their resource isolation schemes [26], [33]–[35]. For example, PerfIso [26] serves as a performance isolation framework, utilizing idle resources for the execution of batch jobs while ensuring no impact on the primary tenant's operations. This framework employs CPU blind isolation, catering to the needs of commercial services that are sensitive to latency variations. Demeter [33] can automatically classify black-box workloads in virtual machines as either LS or BE based on the correlation analysis between network throughput and CPU resource utilization without prior knowledge about them or offline profiling. It then applies different CPU management strategies including core allocation and frequency scaling to LS and BE workloads to achieve power saving.

Resource Management. Researchers tend to optimize resource management to make computing systems more efficient [36]–[39]. For example, dynamically adjusting the CPU frequency gives the chance to achieve optimal resource management. One can use DVFS to dynamically adapt to the load variation. ReTail [14] proposes an automated solution of request-level power management for LC workload, which uses a systematic process to select workloads' features and predicts the latency of requests with a simple machine learning model. To achieve high energy efficiency with no QoS violations, it proposes a dynamic frequency tuning mechanism that can give an appropriate frequency with queueing delay considered. AlphaR [40] proposes a learning-powered resource management system tailored to the microservice environment. It devises a bipartite feature inference approach named Bi-GNN to extract the temporal characteristics of microservices and select an appropriate resource-managing policy to improve microservices' response time.

Frequency adjustment is not the only way to manage resources. For example, NEMO [41] provides a novel platform designed to facilitate the efficient deployment of serverless edge functions within the network function virtualization environment. NEMO intelligently leverages the unused computational cycles of network functions to pre-warm serverless functions and speed up the function invocation in an agile manner.

Energy-efficient Serverless Scheduling. To achieve energy-efficient function scheduling, one possible approach is to place inactive containers or execution environments in a state of low energy consumption. However, this may result in delays during the invocation of the corresponding function, potentially violating QoS requirements.

To address this issue, Ensure [42] proposed to actively reserve additional containers in a warm state to prevent cold starts and smoothly handle workload variations. Fifer [43] undertakes a similar approach, actively creating containers to avoid cold starts. Leveraging these optimizations in schedulers can save resource costs as well as reduce energy for container-based serverless. In addition, MicroFaaS [44] builds a serverless computing platform on new hardware architecture, replacing a few x86-based rack servers with hundreds of ARM-based single-board computers. FIRST [45] finds that the optimal operating point (OOP) for energy efficiency cannot be attained without synthesizing the multi-dimensional attributes of functions and introduces a lightweight internal representation and meta-scheduling layer for collecting the maximum potential revenue from the servers. It analyzes functions from different angles to avoid OOP divergence.

VIII. Conclusion

In this paper, we design an efficient function-level power management system. We build a latency predictor to estimate functions' execution latency and build a QoS-aware request queue. We propose a novel core-level scheduling scheme that can directly allocate functions to given CPU cores and apply function-level power management to achieve power savings within QoS constraints. The experiment shows that our system obtains a maximum power saving of up to 8.5%, with an average power savings of around 8% across the majority of loads, while maintaining the functions' tail latency within QoS constraints. Furthermore, our power-aware function scheduling system has great scalability, which acts as complementary to existing serverless computing systems.

Acknowledgment

We thank all the reviewers for their valuable comments. This work is supported by the Shanghai S&T Committee Rising-Star Program (No.21QA1404400) and by Alibaba Innovative Research Program. The corresponding author is Chao Li.

References

[1] S. Singh, A. Swaroop, A. Kumar *et al.*, "A survey on techniques to achive energy efficiency in cloud computing," in *2016 International conference on computing, communication and automation (ICCCA)*. IEEE, 2016, pp. 1281–1285.

[2] "Working guidance for carbon dioxide peaking and carbon neutrality in full and faithful implementation of the new development philosophy," https://en.ndrc.gov.cn/policies/202110/t20211024_1300725.html, 2021.

[3] "Pathways to net-zero greenhouse gas emissions by 2050," https://www.whitehouse.gov/wp-content/uploads/2021/10/US-Long-Term-Strategy.pdf, 2021.

[4] K. Kaffes, D. Sbirlea, Y. Lin, D. Lo, and C. Kozyrakis, "Leveraging application classes to save power in highly-utilized data centers," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 134–149.

[5] "Apache openwhisk," https://openwhisk.apache.org/, 2023.

[6] "Serverless functions, made simple." https://www.openfaas.com/, 2023.

[7] "Aws lambda," https://amzn.to/34IBiNv, 2023.

[8] "Google cloud functions," https://bit.ly/31DYgDR, 2023.

[9] "Microsoft azure functions," https://bit.ly/3lt92Vg, 2023.

[10] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 289–305.

[11] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[12] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 768–781.

[13] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM symposium on cloud computing*, 2019, pp. 158–164.

[14] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, "Retail: Opting for learning simplicity to enable qos-aware power management in the cloud," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 155–168.

[15] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power containers: An OS facility for fine-grained power and energy management on multicore servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 65–76, 2013.

[16] L. Gerhorst, S. Reif, B. Herzog, and T. Hönig, "Energybudgets: Integrating physical energy measurement devices into systems software," in *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, 2020, pp. 1–8.

[17] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: A platform for os-level power management," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 289–302.

[18] M. Dong, T. Lan, and L. Zhong, "Rethink energy accounting with cooperative game theory," in *Proceedings of the 20th annual international conference on Mobile computing and networking*, 2014, pp. 531–542.

[19] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 64–78.

[20] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[21] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 30–44.

[22] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX annual technical conference (USENIX ATC 20)*, 2020, pp. 205–218.

[23] "Highest response ratio next scheduling algorithm," https://en.wikipedia.org/wiki/Highest_response_ratio_next.

[24] "The object store for ai data infrastructure," https://min.io/, 2023.

[25] "pyrapl 0.2.3.1," https://pypi.org/project/pyRAPL/, 2023.

[26] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang *et al.*, "Perfiso: Performance isolation for commercial latency-sensitive services," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 519–532.

[27] J. Wang, C. Li, J. Mei, H. He, T. Wang, P. Wang, L. Zhang, M. Guo, H. Wu, D. Chen *et al.*, "Hyfarm: Task orchestration on hybrid far memory for high performance per bit," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 33–41.

[28] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 107–120.

[29] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[30] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 248–259.

[31] S. Blagodurov, A. Fedorova, E. Vinnik, T. Dwyer, and F. Hermenier, "Multi-objective job placement in clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[32] X. Wang, C. Li, L. Zhang, X. Hou, Q. Chen, and M. Guo, "Exploring efficient microservice level parallelism," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 223–233.

[33] W. Tang, Y. Ke, S. Fu, H. Jiang, J. Wu, Q. Peng, and F. Gao, "Demeter: Qos-aware cpu scheduling to reduce power consumption of multiple black-box workloads," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 31–46.

[34] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[35] A. Suresh and A. Gandhi, "Servermore: Opportunistic execution of serverless functions in the cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 570–584.

[36] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "Ant-man: Towards agile power management in the microservice era," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[37] X. Hou, J. Liu, C. Li, and M. Guo, "Unleashing the scalability potential of power-constrained data center in the microservice era," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[38] X. Hou, L. Hao, C. Li, Q. Chen, W. Zheng, and M. Guo, "Power grab in aggressively provisioned data centers: What is the risk and what can be done about it," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 26–34.

[39] X. Wang, H. He, Y. Li, C. Li, X. Hou, J. Wang, Q. Chen, J. Leng, M. Guo, and L. Wang, "Not all resources are visible: Exploiting fragmented shadow resources in shared-state scheduler architecture," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023, pp. 109–124.

[40] X. Hou, C. Li, J. Liu, L. Zhang, S. Ren, J. Leng, Q. Chen, and M. Guo, "Alphar: Learning-powered resource management for irregular, dynamic microservice graph," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 797–806.

[41] L. Zhang, W. Feng, C. Li, X. Hou, P. Wang, J. Wang, and M. Guo, "Tapping into nfv environment for opportunistic serverless edge function deployment," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2698–2704, 2021.

[42] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 1–10.

[43] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 280–295.

[44] A. Byrne, Y. Pang, A. Zou, S. Nadgowda, and A. K. Coskun, "Microfaas: energy-efficient serverless on bare-metal single-board computers," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 754–759.

[45] L. Zhang, C. Li, X. Wang, W. Feng, Z. Yu, Q. Chen, J. Leng, M. Guo, P. Yang, and S. Yue, "First: Exploiting the multi-dimensional attributes of functions for power-aware serverless computing," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 864–874.