SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2025, Vol. 68, Iss. 3, 132101:1–132101:19
https://doi.org/10.1007/s11432-022-3882-y

Power synchronization: taming massive diversified serverless functions under power constraints

Du LIU[1†], Lu ZHANG[1†], Yechen XU[1], Xinkai WANG[1], Lingyu SUN[1], Yifei PU[1], Xiaofeng HOU[2], Chao LI[1*] & Minyi GUO[1]

[1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai* 200240*, China;*
[2]*AI Chip Center for Emerging Smart Systems, The Hong Kong University of Science and Technology, Hong Kong* 999077*, China*

Received 20 December 2022/Revised 28 July 2023/Accepted 20 October 2023/Published online 27 November 2024

Abstract Carbon neutrality has become an important design objective worldwide. However, the on-going shift to cloud-naive era does not necessarily mean energy efficiency. From the perspective of power management, co-hosted serverless functions are difficult to tame. They are lightweight, short-lived applications sensitive to power capping activities. In addition, they exhibit great individual and temporal variability, presenting idiosyncratic performance/power scaling goals that are often at odds with one another. To date, very few proposals exist in terms of tailored power management for serverless platforms. In this work, we introduce power synchronization, a novel yet generic mechanism for managing serverless functions in a power-efficient way. Our insight with power synchronization is that uniform application power behavior enables consistent and uncompromised function operation on shared host machines. We also propose PowerSync, a synchronization-based power management framework that ensures optimal efficiency based on a clear understanding of functions. Our evaluation shows that PowerSync can improve the energy efficiency of functions by up to 16% without performance loss compared to conventional power management strategies.

Keywords serverless computing, variability, energy efficiency, power management, power synchronization

Citation Liu D, Zhang L, Xu Y C, et al. Power synchronization: taming massive diversified serverless functions under power constraints. Sci China Inf Sci, 2025, 68(3): 132101, https://doi.org/10.1007/s11432-022-3882-y

1 Introduction

Serverless functions represent a significant and increasing workload on cloud servers [1–4]. Many cloud providers have offered serverless computing (function as a service) such as AWS, IBM, and Azure. According to Google Trends, serverless computing has garnered in industry tradeshows, meetups, blogs, and the development community [5]. This model allows users to focus on the task logic while abstracting away the low-level complexity from users. In contrast to the traditional model of virtual machines (VMs), cloud providers that offer the serverless services are responsible for managing functions in a transparent and auto-scaling manner. To achieve high server utilization, vendors tend to co-locate thousands of functions concurrently [6,7] or co-locate both VMs and serverless functions complementally [8,9]. The light-weight nature of serverless functions allows them to be co-hosted on a single processor core [10–12]. Due to its great elasticity and flexibility, going serverless has become a key move in the cloud-native era.

Much research has been done to boost serverless computing's performance [13,14]. Besides, to make the serverless model more cost-effective and profitable, the underlying cloud servers should also be carefully managed for greater energy efficiency [15,16]. Further, green consciousness is quickly making its way into cloud. IT companies now have strong incentives to limit system power to improve sustainability.

While there are many recent papers on improving serverless computing performance [3, 17, 18], very limited studies have been done with respect to power management optimization for serverless functions [19]. Managing co-hosted applications under the given power budget has long been an important topic. However, existing power management mainly emphasizes power allocation for traditional applications under power and latency constraints [20–22]. Providing customized performance/power levels

* Corresponding author (email: lichao@cs.sjtu.edu.cn)
† Liu D and Zhang L have the same contribution to this work.

for each application is appealing [20, 21, 23], but it cannot be directly applied to serverless functions. Adjusting the power in a fine-grained manner for short-lived functions faces significant control overhead with current power scaling speed [20].

Managing the power of serverless functions is not straightforward. We show that serverless functions may have different power-performance behaviors depending on their runtime environment and hardware resource dependencies. The fact that a large amount of functions are co-located at the same time further exacerbates the problem even more. Another key challenge of serverless function power management is that they exhibit great variability in the temporal domain. Typically, a serverless function experiences multiple phases throughout its life [2], requiring different optimal performance/power settings. Besides, their whole duration can also vary greatly depending on parameters. In such a highly dynamic environment, serverless platforms may end up with frequent CPU frequency/voltage adjustment or function reshuffle to restore the optimal operating state of the server. This could result in significant overhead, including transition delay of frequency [20, 24], function migration delay, and other non-deterministic factors [25].

In this work, we aim to gain a deep understanding of power management for co-located serverless functions. We argue that serverless platforms must carefully consider the variety of functions and make informed power management decisions. We propose power synchronization, a new power management model allowing better workload-hardware mapping. A key feature of our model is that it puts an emphasis on functions with synchronized power behavior, i.e., having similar best-suited performance/power settings and well-aligned life cycles, for the achievement of common efficiency goals.

We introduce PowerSync, a novel power management framework that provides high power efficiency when functions co-locate on the same processor core. Specifically, PowerSync adopts a two-pronged strategy when performing power synchronization. First, PowerSync could accurately estimate the optimal operational settings of various functions including its best-suited frequency and duration in different phases. It synchronizes the power management of various functions by deploying functions with the same optimal operational setting on the same processor core. Additionally, PowerSync has a built-in frequency/phase cooperative switch mechanism which is able to restore the best-suited frequency of functions with low overhead as their phases move forward.

This paper makes the following contributions:

- Our characterization of functions' power behavior, for the first time, reveals that individual and temporal variability could greatly affect the power efficiency of the underlying computing hardware that supports serverless functions.
- We propose power synchronization, a novel power management mechanism for dispatching functions under a consistent best-suited frequency to pursue optimal efficiency. Furthermore, we design PowerSync, a serverless computing power management framework that implements power synchronization with minor design overhead.
- We perform extensive experiments to evaluate the effectiveness of PowerSync from different perspectives. Compared to the state-of-the-art power management strategies, PowerSync can improve energy efficiency by up to 16% without performance loss.

The rest of this paper is organized as follows. Section 2 provides the background and related work. Section 3 introduces key design considerations. Section 4 proposes power synchronization. Section 5 presents the system design of PowerSync. Section 6 describes the experimental methodologies. Section 7 details experimental results. Section 8 analyzes PowerSync on real machines and the future work. Lastly, Section 9 concludes this paper.

2 Background & related work

In this section, we introduce the background and related work of serverless functions and discuss different dimensions of application co-location management.

2.1 Resource management under co-location

We classify existing resource management studies as either exclusive or inclusive.

Managing exclusive resource. We term processor core, memory cache, and network resources, as exclusive resource. This is because the quantity of resources such as cores, memory, and network are all limited. Co-locating multiple applications with similar resource preferences may cause interference, even

with multiplexing. To resolve this problem, one needs to consider the co-relation of applications and try to deploy heterogeneous applications that have different resource requirements on the same node [26–28].

Managing inclusive resources. Power and thermal management are different from exclusive resource management. Power configuration like frequency settings of a processor core is shared by all applications running on it. We refer to power/cooling as inclusive resources. By allocating applications with the same power/frequency, we can meet the requirements of all the tasks. Note that the above discussion is based on the premise that there is no severe contention with regard to the exclusive resources. We maintain an appropriate level of server utilization and do not aggressively co-locate services.

Many prior studies [21,22,29,30] proposed to reduce power/energy consumption while guaranteeing the QoS of traditional applications. Adrenaline [22] and EEPL [30] target the tail latency of latency-critical applications while optimizing energy efficiency. Ant-Man [20] provides highly efficient power management in the microservice environment. Additionally, the energy-proportional design aims to improve the energy efficiency of servers at low utilization [31, 32]. However, these studies all lead to sub-optimal energy efficiency in the serverless computing era, since they do not consider the inherent behavior of functions co-located on the same processor core.

2.2 Variety of serverless functions

Serverless computing is a rapidly growing cloud application model. A serverless function is a bundle of code written by high-level languages, such as Python and Node.js [2]. Since a serverless function only handles the specific piece of computation logic and functions are typically live-shorted(O(100 ms)) [2–4]. Moreover, to achieve high server utilization and make the serverless model profitable, cloud vendors may colocate thousands of functions concurrently [6, 7]. To improve the performance of functions, prior works schedule functions according to their resource requirement. GrandSLAm [33] guarantees the SLA for Microservice/FaaS by identifying slack in individual queries of different applications. FnSched [34] considers the diversity in resource consumption and lifetime of functions.

FaaS generally makes use of containers, virtual machines, or other isolation methods to deploy functions. When functions are invoked, serverless platforms firstly initialize the runtime environment and booting functions [2, 17, 18]. Thus, the duration of functions includes different phases, mainly including the initialization and execution phase [35, 36]. In the initialization phase, serverless computing platforms prepare the runtime environment of functions according to the function code. In the execution phase, serverless functions process the logic with the given parameters. Due to the variety of phases' processing flows, the performance and energy cost of different functions would be also various.

To our knowledge, prior studies do not look into the power management issue of serverless functions. We take the first step to investigate the power-saving potential of serverless platforms with a novel mechanism: power synchronization.

3 Power implications of function's individual and temporal variability

While there are many recent studies on improving the performance of serverless function invocation, the power behavior of serverless functions is still not well-understood.

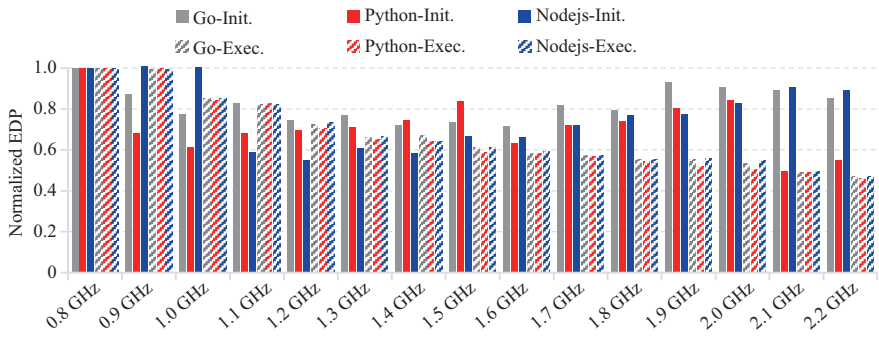
In this section, we show that traditional function invocation methods incur sub-optimal efficiency due to two types of workload variability. We first observe that diversified serverless functions have different best-suited CPU frequencies, which we refer to as individual variability. Second, we find that temporal variability, i.e., the fact that serverless functions have asynchronous life cycles, makes power efficiency optimization even more challenging. We present a detailed evaluation methodology in Section 6.

3.1 Individual & temporal variability

We thoroughly examine the power-performance trade-off in function power management and summarize the results into individual variability and temporal variability.

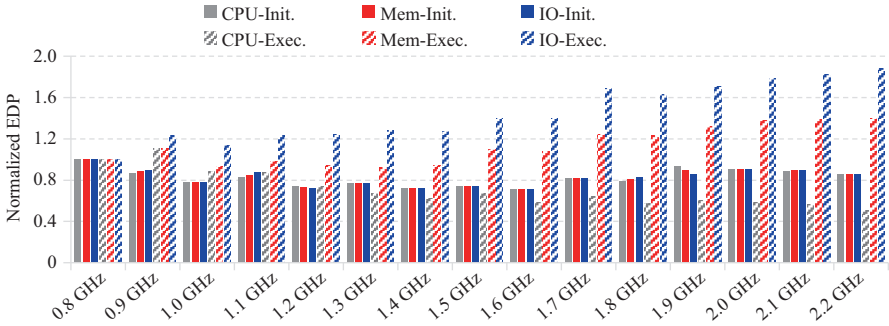
3.1.1 *Individual variability*

Note that developers generally upload functions written in a variety of programming languages (such as Python, Go, and Node.js, etc.) to serverless platforms [2, 4]. In addition, functions can also be classified according to their resource requirement characteristics (CPU-intensive, Memory-intensive, and



	Go-Init	Python-Init	Nodejs-Init	Go-Exec	Python-Exec	Nodejs-Exec
Best-suited frequency	1.6 GHz	2.1 GHz	1.2 GHz	2.2 GHz	2.2 GHz	2.2 GHz

Figure 1 (Color online) Comparison of ALU functions written in various languages.



	CPU-init	Mem-init	IO-init	CPU-exec	Mem-exec	IO-exec
Best-suited frequency	1.6 GHz	1.6 GHz	1.6 GHz	2.2 GHz	1.0 GHz	0.8 GHz

Figure 2 (Color online) Comparison of different resource-intensive functions written in Go.

IO-intensive). Therefore, in Figure 1 we first experiment with a representative function [37] that can be implemented with different languages, and in Figure 2 we further evaluate different types of resource-intensive functions written in the same language.

In Figures 1 and 2, we use energy-delay product (EDP) to quantify the efficiency of function invocation. EDP is the product of energy (Joule) and delay(s) which offers equal weight to energy consumption and performance degradation [20]. We show the EDP of both the initialization phase and execution phase of each function under different frequencies (from 0.8 to 2.2 GHz). We look for the best-suited frequency which means that each function can achieve its lowest EDP under the frequency. Since serverless functions present various power-performance behaviors, if providers focus more on performance, they can increase the weight of performance to look for the best-suited frequency of each function.

Our characterization has identified interesting individual differences in terms of the best-suited frequency. Note that we only consider the case when the function is cold-started in our characterization. For a cold-start function, its duration can be divided into the initialization phase and the execution phase. For a warm-start function, only the execution phase needs to be considered. Thus, the case of cold starting can be regarded as a superset of that of warm starting. When we do the measurement and design our system in the environment of cold starting, the result can also be applied to warm-start functions. In Figure 1, while CPU frequency scaling affects function EDP greatly in the initialization phase, it hardly makes any effect in the execution phase. The result implies that different runtime environments often lead to different best-suited frequencies when functions are initializing. For example, the best-suited initialization frequency for ALU (Go), ALU (Python), and ALU (Node.js) are 1.6, 2.1, and 1.2 GHz, respectively. On the contrary, resource requirement characteristics play a dominant role in the best-suited frequency selection during the execution phase but have negligible influence during

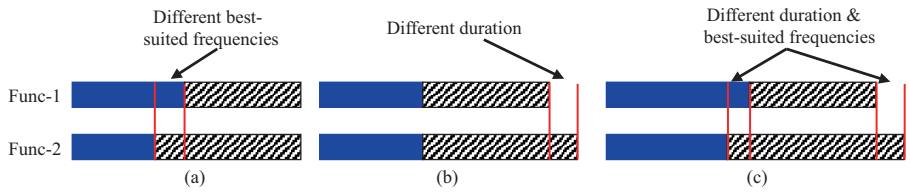


Figure 3 (Color online) Different cases of asynchronization when only considering best-suited frequency of functions (blue bars represent the initialization phase and black bars represent the execution phase). (a) Phase Async.; (b) life Async.; (c) life Async.

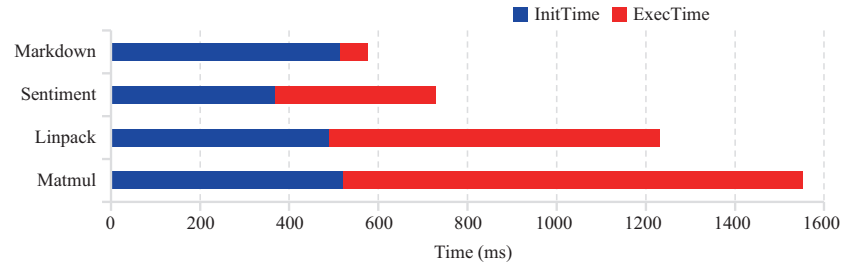


Figure 4 (Color online) Duration breakdown of functions of the same best-suited frequency (2.1 GHz in the init. phase and 2.2 GHz in the exec. phase).

Table 1 Ratio of phase Async. and life Async. for co-hosted functions[a)]

	Mat &Lin	Mat &Sen	Mat &Mar	Lin &Sen	Lin &Mar	Sen &Mar
Phase Async. (%)	3	21	1	16	2	20
Life Async. (%)	21	53	63	41	53	21

a) Mat: matmul, Lin: Linpack, Sen: sentiment, Mar: markdown.

the initialization phase, as shown in Figure 2. The best-suited execution frequency for CPU-intensive, Mem-intensive, and IO-intensive are 2.2, 1.0, and 0.8 GHz, respectively.

3.1.2 *Temporal variability*

Once we capture the individual variability of serverless functions, we could co-locate functions of similar power-performance behaviors on a processor core and set the best-suited CPU frequency thereafter.

However, best-suited frequency matching is not the only consideration when invoking serverless functions. The optimal efficiency cannot be achieved if the co-located functions have asynchronous life cycles.

In Figure 3, as an example, we consider two functions with the same best-suited initialization frequency and the same best-suited execution frequency. We list three possible scenarios that may lead to compromised efficiency.

(1) Phase Async. The two co-located functions have different initialization durations. In this case, there is an interim period when func-2 is executed while func-1 still stays in the initialization phase. The processor cannot guarantee the best-suited frequency for both functions during the interim period, thereby causing sub-optimal energy efficiency.

(2) Life Async. Although the two functions have the same initialization duration, func-1 finishes earlier than func-2. In this case, a CPU slack is created, which is a waste of resources and power. It is often infeasible to harvest the CPU slack by inserting special functions of specific frequency requirements.

(3) Mixed Async. It is the worst case. The co-located functions incur both phase Async. and life Async.

The temporal variability issue can be significant. In Figure 4, we invoke four functions with the same best-suited frequency (2.1 GHz for initialization and 2.2 GHz for execution). We measure the initialization duration under 2.1 GHz and execution duration under 2.2 GHz for each function. Apparently, there is a large difference in both the initialization duration and the total duration among these functions. Table 1 shows the ratio of phase Async. and life Async. if we select two functions and co-locate them on the same core. In the experiment, phase Async. accounts for 1%–21% of the total CPU time and the ratio of life Async. ranges from 21%–63%.

Summary. It is desirable to assign the best-suited frequency for each serverless function. However,

functions have various language runtime and resource requirements, affecting best-suited frequency selection in different ways (individual variability). Besides, the duration of functions also plays a key role in managing functions with power efficiency (temporal variability). To make the serverless computing more efficient, the power management system should take both those individual and temporal differences seriously.

3.2 Limitations of existing solutions

Our work is mainly driven by this question: can a cloud-native infrastructure be more efficient if its task scheduler has more visibility into the property of functions? Particularly, we focus our attention on a more challenging scenario: power-constrained servers with heavy FaaS query traffic. Due to a lack of appropriate tuning knobs, we mainly consider processor power in this paper and leave full-server power management in future work. We examine the individual differences of best-suited CPU frequency and how much time it spends initializing functions and executing instructions. We want to understand how co-hosted serverless functions with individual and temporal variability should be managed as a whole to best utilize the limited power budget.

There are a few compelling reasons that classical power management models in the literature cannot solve the problem.

• At the workload-level, fine-grained scheduling policies [38] that coalesce idle and busy periods have efficiency issues, since in the field of serverless computing, there are not many resource slacks to utilize. Besides, the functions are highly sensitive to scheduling due to their short lives so that the fine-grained scheduling might severely degrade their performance.

• At the system-level, per-application hardware adaptation [23] incurs large control overhead. Servers can be overwhelmed by massive small stateless services with such an aggressive power management method.

4 Power synchronization

We propose power synchronization, a novel mechanism for dispatching serverless functions efficiently.

4.1 Definition

Power synchronization is a special power-driven batch processing designed specially for serverless functions. It is a unification process through which various functions are gracefully mapped to processor cores for the achievement of common energy efficiency goals.

No power synchronization. This is the default setting for conventional serverless platforms. A function can be optionally put on any idle processor core, as long as the necessary data fit in the local memory. Various functions might request drastically different power/performance settings, and consequently the processor core could be left in an inefficient state. In other words, functions' design goals (best-suited frequency) are at odds with one another and must be traded off.

Power-synchronized state (PSS). Better energy efficiency would be satisfied by co-locating functions with the same performance-power behaviors on a single core. Specifically, two conditions must be met: (1) for any given phase of its life cycle, each co-located function demands the same best-suited CPU frequency; (2) every function moves forward at the same pace — their phases are well-aligned.

4.2 Benefits of power synchronization

In Figure 5 we show the influence of power synchronization on the energy efficiency of functions. We invoke a group of functions (matmul, linpack, sentiment, and markdown). There are six cases of co-location shown in Figure 5. Take Mat & Lin as an example, we run the two functions in two ways. (1) Sync.: we colocate two Mat functions on one core and two Lin on another. (2) Async.: we run two sets of Mat, Lin on two cores. Note that the core binding here can be achieved by cgroup-based techniques like Docker. As shown in Figure 5, without power synchronization, the overall processor EDP would increase by 13%–89%.

Note that power synchronization complements, not replaces, the existing function scheduling mechanism. A key benefit of our design is that it offers a way to get functions running at their best-suited frequency under a stringent power budget.

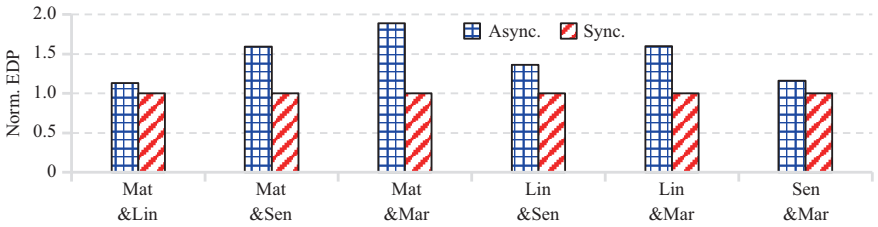


Figure 5 (Color online) Normalized EDP of two functions invoked with power Sync. or Async. (Mat: matmul, Lin: Linpack, Sen: sentiment, Mar: markdown).

In addition, power synchronization intends to be non-intrusive and convenient. It neither places extra requirements on function implementation nor uses expensive locks to proactively synchronize functions' execution phases. Its design considerations are discussed in Section 5.

4.3 Relaxed policy

Considering various non-deterministic factors in the real world, strictly enforcing power synchronization can be difficult if not impossible. In practice, a relaxed policy can be much easier to implement while preserving the benefits of power synchronization. We define relaxed power synchronization as follows:

$$\begin{cases} \forall i, j \in F,\ \forall p \in P,\ |\mathrm{Freq}_{i,p} - \mathrm{Freq}_{j,p}| < F_t, \\ \mathrm{FDD} < D_t. \end{cases} \tag{1}$$

In (1), $\mathrm{Freq}_{i,p}$ is the power requirement of the function i in phase p. F_t and D_t are differentiation threshold values related to frequency and duration, respectively. They are user-defined parameters that imply the degree of inaccuracy that one can tolerate. FDD is the function duration difference of functions co-located on the same processor core given by

$$\mathrm{FDD} = \frac{1}{N_P} \sum_{p}^{P} \sqrt{\frac{\sum_{i}^{F} (D_{i,p} - \hat{D}_p)^2}{N_F \hat{D}_p{}^2}}, \tag{2}$$

where $D_{i,p}$ is the time of function i in the phase p, $\hat{D}_p$ is the average duration of all functions in the phase p, F indicates all functions on the same core, N_F is the total number of F, P is phases of functions and N_p is the total number of P.

5 PowerSync design

In this section, we propose PowerSync, a novel serverless computing power management framework that keeps power synchronization in mind. Our goals are to manage co-located functions in a way that automatically adapts to the given power budget and to improve the power efficiency of server clusters by dynamically configuring the PSS.

5.1 Overview

In general, PowerSync is a light-weight patch that allows a serverless platform to run serverless functions with the PSS. It is designed to be a piece of system components that can be added to the existing serverless platforms like OpenWhisk.

Figure 6 depicts the system architecture for PowerSync. In the central controller, PowerSync adds a function unification module and a function dispatch module to synchronize the power behavior of massive serverless functions. In the serverless invoker, PowerSync also uses a in-node manager module to provide fine-grained function management. We design our system towards the batch invocation. In the setting of the batch invocation, a large group of functions are invoked simultaneously.

The function unification module is designed to analyze both the individual and temporal variability on the serverless platform, thereby creating a group of unified functions ready to be invoked. We refer to such a bundle of functions in the PSS as a synchronized function set. It is the basic unit for function invocation using PowerSync.

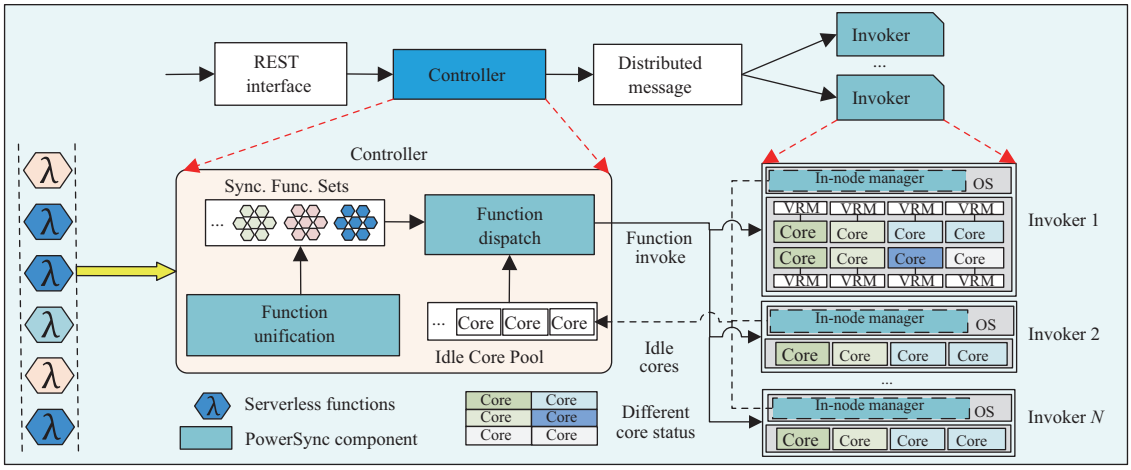


Figure 6 (Color online) Architecture of PowerSync design.

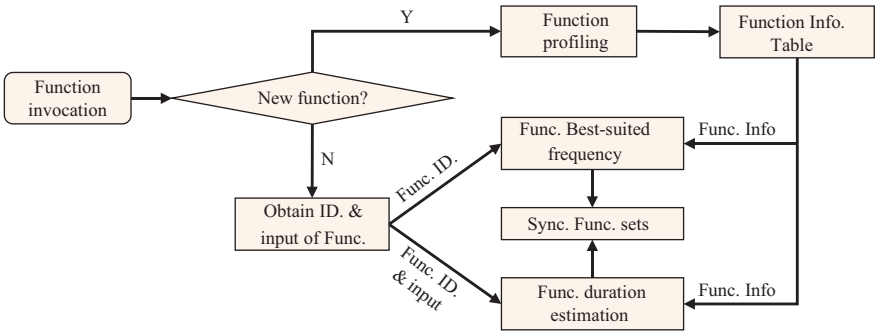


Figure 7 (Color online) Workflow of function unification.

Afterwards, the function dispatch module aims to dispatch the synchronized function set to an idle core. Note that the sets are determined algorithmically at the beginning of each batch invocation. Since the assignment is largely predetermined, it does not incur much function orchestration overhead at runtime. Functions do not need to communicate with one another to achieve power synchronization.

After dispatching, PowerSync adds a in-node manager to each OpenWhisk invoker to monitor and fine-tune functions' status during runtime. This local process can be regarded as an add-on module of the invoker. It can adjust CPU frequency as functions' phase changes to maintain the best-suited frequency. Meanwhile, it can also timely identify possible CPU slacks and check the effectiveness of power synchronization to further calibrate the entire process.

5.2 Function unification

The first step of PowerSync is to quickly identify two important properties of each function: (1) the best-suited frequency of its initialization phase and execution phase, and (2) the duration of the two phases respectively. Using this information, PowerSync can group functions into synchronized function sets. The workflow of function unification is shown in Figure 7.

Best-suited frequency analysis. As discussed earlier, the best-suited frequency of each phase heavily relies on the language type and resource type of serverless functions. When functions are invoked, we can search historical data and use a lookup table to obtain functions' best-suited frequencies in their initialization and execution phases.

PowerSync stores all the collected information in a PSS profiling table showing in Figure 8. For a newly created unknown serverless function, we run the very first invocation of it on a dedicated server for characterization. Subsequent invocations do not need to be profiled again. Note that the profiling here is mainly about the best-suited frequency of the function under different phases. In the meantime, we will also invoke the new function under different parameters to collect training data.

Today, major serverless providers such as AWS have already offered various tools to monitor functions.

Function	Freq. Req.	Duration-related Info.
$Func_1$	<$Freq_{init}$, $Freq_{exec}$>	<Freq, input_size, T_{init}, T_{exec}>
$Func_2$	<$Freq_{init}$, $Freq_{exec}$>	<Freq, input_size, T_{init}, T_{exec}>
...	...	...
$Func_n$	<$Freq_{init}$, $Freq_{exec}$>	<Freq, input_size, T_{init}, T_{exec}>

Figure 8 PSS profiling table.

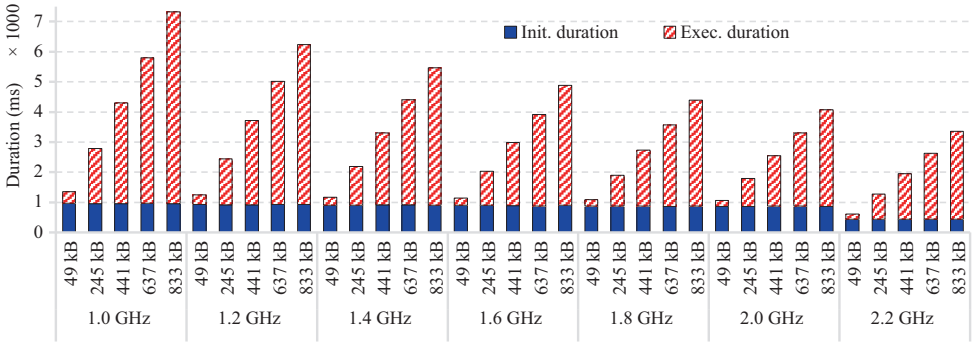


Figure 9 (Color online) Example of duration change under different file size.

Since functions are managed in Docker containers in our system, tools like docker stats and `Intel's PQoS` are available. When functions are first created, we use docker stats and `Intel's PQoS` to profile the language and determine the resource type of serverless functions. We then use `cpupower` [39] to adjust the frequency of cores to obtain the duration of each serverless function. If the already profiled functions exhibit significantly different characteristics at runtime, we will update the information or re-profile functions.

Function duration estimation. Besides the best-suited frequency of functions, another important factor that affects PSS is the initialization and execution time (ET) of functions. However, estimating the duration of functions is not straightforward since the actual ET would be relevant to the parameter of functions. As the parameter varies, the amount of computation actually performed by the function also changes, which is reflected in the change in latency.

To understand the impact of the parameters on the duration of the given function, we experiment with the function Markdown2html, a representative serverless function for rendering markdown text to HTML in cloud [2]. Figure 9 shows the function's initialization and execution duration with different parameter settings under a number of frequency levels. It is intuitive that function execution duration increases as the file size becomes larger, which implies that the execution duration is input-dependent. However, the initialization duration is largely independent of the file size as it keeps stable when the file size increases. Moreover, we can also see that the initialization duration is much less sensitive to different frequency levels, compared with the execution duration.

Understanding the lifecycle of co-located serverless functions is crucial for maintaining PSS. PowerSyncSince the function duration cannot be directly determined based on historical data, we need to build an estimation model to estimate the duration of each function for all available frequencies. The duration of a function includes the initialization time and ET.

In this work, we use a linear regression model to estimate the ET since the prediction time is short and it can achieve high accuracy for duration estimation. We train a specific model for each function under different processor frequencies to predict the duration of a given function under different parameters.

$$\mathrm{ET} = a + bS, \tag{3}$$

where S is the parameter vector to invoke functions which is an independent variable. a and b are the intercept and slope vectors of the regression model. The initialization time is determined by the processor's frequency which can be obtained from the PSS table.

To construct the training and testing datasets, we have collected a large number of traces of functions with various parameters under different processor frequencies. We randomly choose 80% of the function

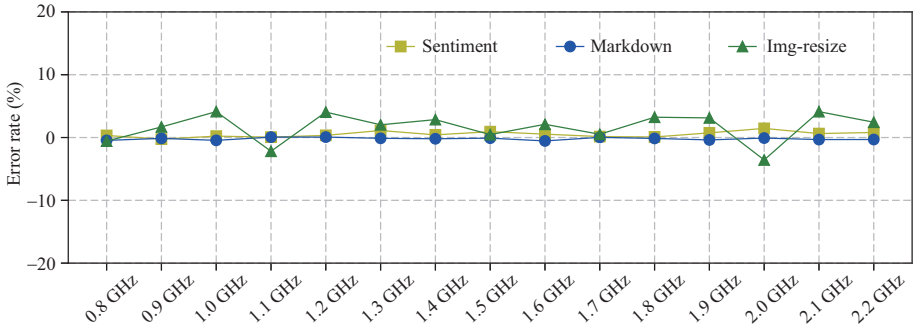


Figure 10 (Color online) Error rate (%) in predicting ET under each frequency level.

traces as the train datasets and the rest for testing. The error rate is defined as

$$\mathrm{Err} = \frac{\mathrm{ET}_{\mathrm{predicted}} - \mathrm{ET}_{\mathrm{measured}}}{\mathrm{ET}_{\mathrm{measured}}}. \tag{4}$$

Figure 10 shows the error rate of predicting the ET of different functions under different frequency levels. As we can see, the error rate in predicting the ET from our model is small. The largest error rate is about 5% when predicting the ET of img-resize function. To improve the accuracy of the predicted model, the predicted model is designed to self-optimize over the runtime. To be specific, the model compares the estimated values with the real function duration during runtime. If the error is large (>5% [20]), PowerSync can incrementally update the model.

Synchronized function sets. Given the best-suited frequency and the duration of functions, each function can be represented by a quadruple $\langle \mathrm{Freq}_{\mathrm{init}}, \mathrm{Freq}_{\mathrm{exec}}, T_{\mathrm{init}}, T_{\mathrm{exec}} \rangle$. The four values in the quadruple are quantified based on thresholds set in (1). In this case, functions with similar best-suited frequency and duration would lead to PSS and hence will be grouped together to form synchronized function sets. Afterwards, PowerSync can deploy functions in the same synchronized function set to the same processor core to achieve power synchronization.

5.3 Function dispatch

Having identified the synchronized function sets, our next job of PowerSync is to dispatch them to idles cores of invokers. We devise a two-step control for function dispatch.

Step 1. At the beginning of each control period, PowerSync fetches functions from each synchronized function set according to the co-location limit of processor cores. For example, supposing that each core can co-locate at most M functions, while the size of one synchronized function set is N, at this period, PowerSync can only dispatch $\lfloor N/M \rfloor$ sets of functions to idle cores. The remaining $N\%M$ functions have to wait for the incoming functions. Otherwise, these functions would be processed in an asynchronized way. In this step, PowerSync obtains the function information including the PSS quadruple $\langle \mathrm{Freq}_{\mathrm{init}}, \mathrm{Freq}_{\mathrm{exec}}, T_{\mathrm{init}}, T_{\mathrm{exec}} \rangle$.

Step 2. PowerSync further searches the idle core pool to determine which invoker the synchronized functions should be dispatched to based on other design considerations such as load balance or warm start. The PSS quadruple $\langle \mathrm{Freq}_{\mathrm{init}}, \mathrm{Freq}_{\mathrm{exec}}, T_{\mathrm{init}}, T_{\mathrm{exec}} \rangle$ will be sent along with the dispatch message to instruct the invoker to set the right frequency at the right time. As in OpenWhisk, new fields can be added to the ActivationMessage API to carry the quadruple.

Step 3. For each synchronized function set, there might be $N\%M$ functions left at the current dispatch period. In this case, the left functions incur certain queuing delays. To minimize the negative impact, PowerSync records the queuing delay of each function. Once the queuing delay exceeds 10% of the functions' total duration, PowerSync would dispatch these functions to any of the idle cores with the highest frequency which aims to compensate for their long queueing delay.

5.4 In-node manager

Finally, PowerSync uses an In-Node Manager in each invoker to dynamically manage function processing. The In-Node is a daemon thread in each invoker. Figure 11 shows its responsibility, which consists of three sub-tasks: (1) function invocation, (2) system re-sync, and (3) data feedback.

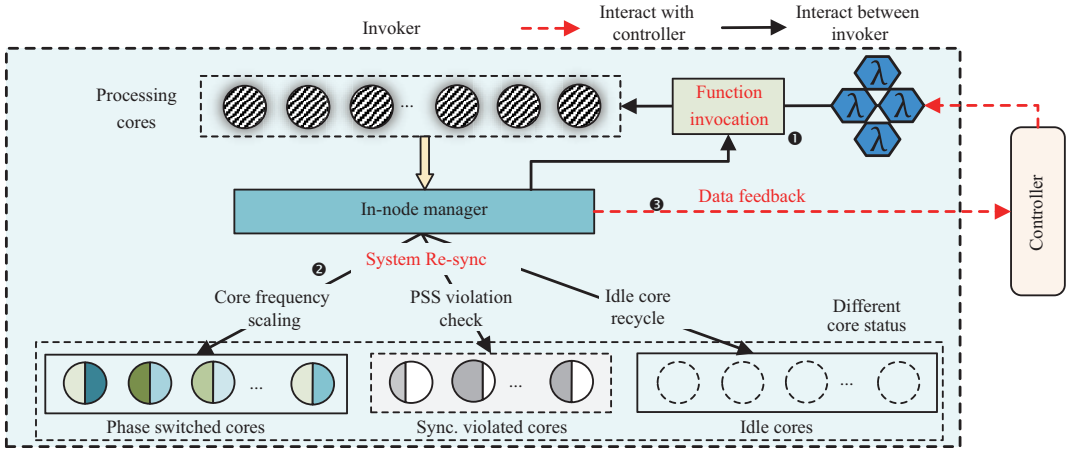


Figure 11 (Color online) In-node manager in each invoker.

Function invocation. Once PowerSync dispatches a bundle of synchronized functions to invokers, the in-node manager start to acquire invocation requests from the controller. It parses the invocation request to obtain $\langle \text{Freq}_{\text{init}}, \text{Freq}_{\text{exec}}, T_{\text{init}}, T_{\text{exec}} \rangle$ of the synchronized functions. It then informs the host processor to select the best-suited frequency for the invoked functions.

System re-Sync. Another key duty of the in-node manager is to re-sync the functions, namely, adjust the core frequency as functions' phases change. It first checks the initialization time of functions to see if their duration equals the stored historical data. Then, it obtains the phase information by analyzing enriched container logs. After the verification, the in-node manager adjusts processor cores with the new best-suited frequency. Note that cores may face asynchronized functions due to certain non-deterministic factors. We need to check the proportion of phase Async. and life Async. on each processing core periodically to record the violation statistics. Finally, the in-node manager recycles idle processor cores. It puts idle cores into sleep states for further power saving.

Data feedback. At last, the in-node manager sends the necessary log data of each invoker to the controller. For example, the controller can update the idle core pool using the information for further function dispatch. It can also re-profile the functions or update the duration prediction model.

6 Experimental methodology

To thoroughly analyze the benefits of PowerSync using the realistic traces, we evaluate our design with a trace-driven approach as shown in Figure 12. The evaluated server has 20 physical cores (Intel Xeon Silver 4114) with Ubuntu 16.04.5 LTS installed. The processor supports per-core DVFS with operating frequencies from 0.8 to 2.2 GHz at an interval of 0.1 GHz. The frequency driver is ACPI with the "userspace" governor. With Linux tool turbostat, we can record dynamic power and energy consumption. We use Openwhisk as our serverless platform.

We set up experiments with serverless functions shown in Table 2, which are from FaaS-Profiler [2] and FunctionBench [40]. In addition, we implement the ALU algorithm (a CPU-intensive function used in ServerlessBench [37]) and FileIO function (IO-intensive). We evaluate our design using different function pools as shown in Table 3. SPF includes functions whose best-suited frequencies are synchronized but with asynchronized duration ($\text{Freq}_{i,p} = \text{Freq}_{j,p}$, FDD > 0). Functions in SPD all have the same duration while the best-suited frequencies are asynchronized ($\text{Freq}_{i,p} \neq \text{Freq}_{j,p}$, FDD $= 0$). Lastly, ASP contains all functions combined in Table 2 to show the variability ($\text{Freq}_{i,p} \neq \text{Freq}_{j,p}$, FDD > 0).

We invoke each function pool using different invocation patterns as shown in Figure 13 to generate different request sets. The invocation patterns are all generated from a real cluster. The flat and periodical patterns follow the trace from Alibaba. To represent the production-live serverless invocation pattern, we use the Azure Function trace which consists of serverless invocation data of two weeks. The generated trace and the serverless function information (the best-suited frequency and duration of each function) are fed into PowerSync.

Table 4 summarizes the evaluated schemes. There are three important baselines that do not consider PSS: NPS, IL, and Per-APP. NPS and IL represent a widely used application deployment mechanism,

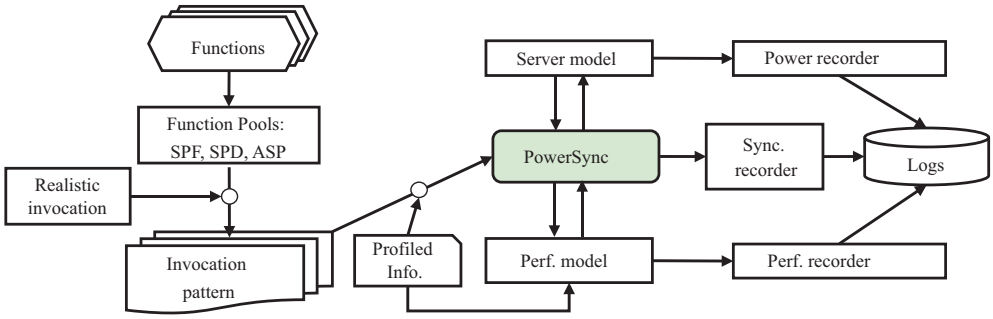


Figure 12 (Color online) Workflow of a trace-driven evaluation.

Table 2 Serverless functions in our experiment

Function	Description	Runtime
Markdown	Markdown to HTML	Python
Img-resize	Resize image	Nodejs
Sentiment	Sentiment analysis	Python
Ocr-img	Find text in images	Nodejs
Autocomplete	Autocomplete string	Nodejs
Matmul/linpack	CPU/Mem-intensive	Python
FileIO	IO-intensive function	Go
ALU	CPU-intensive function	Python/go/ruby/ swift/php/nodejs

Table 3 Evaluated serverless function pool

Function pool	Functions
Synchronized Freq. pool (Abbr. SPF) ($\text{Freq}_{i,p} = \text{Freq}_{j,p},\ \text{FDD} > 0$)	Functions with the same best-suited Freq.
Synchronized Dura. pool (Abbr. SPD) ($\text{Freq}_{i,p} \neq \text{Freq}_{j,p},\ \text{FDD} = 0$)	Functions with the same duration
Asychronized pool (Abbr. ASP) ($\text{Freq}_{i,p} \neq \text{Freq}_{j,p},\ \text{FDD} > 0$)	All functions combined

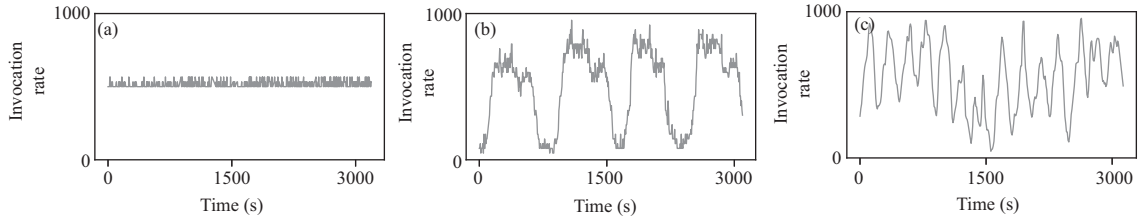


Figure 13 Function invocation patterns in the realistic cluster. (a) and (b) from Alibaba trace; (c) from Azure Functions trace [3]. (a) Flat pattern; (b) periodical pattern; (c) fluctuate pattern.

Table 4 Evaluated power management schemes

Mechanism	PSS	Description
NPS	No	Round-robin invocation w/ no power synchronization
IL	No	Round-robin invocation w/ ideal latency
Per-APP	No	Considering function Sync. as a whole application
PS&Freq	Partly	Considering best-suited frequency synchronization
PS&Dura	Partly	Considering duration synchronization
PS&Opt	Yes	Optimized management of PowerSync

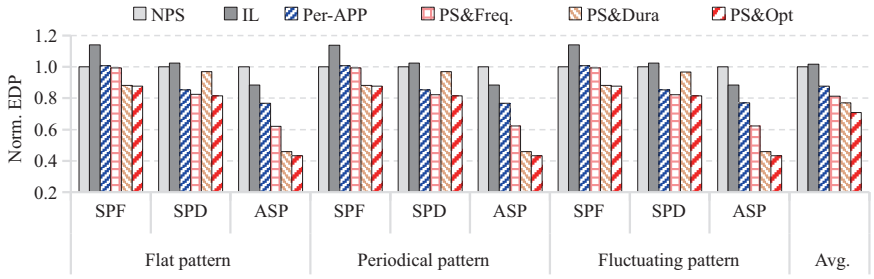


Figure 14 (Color online) Normalized EDP of different schemes.

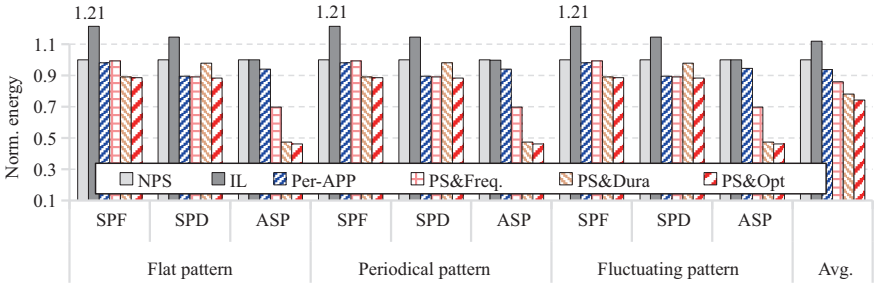


Figure 15 (Color online) Normalized energy of different schemes.

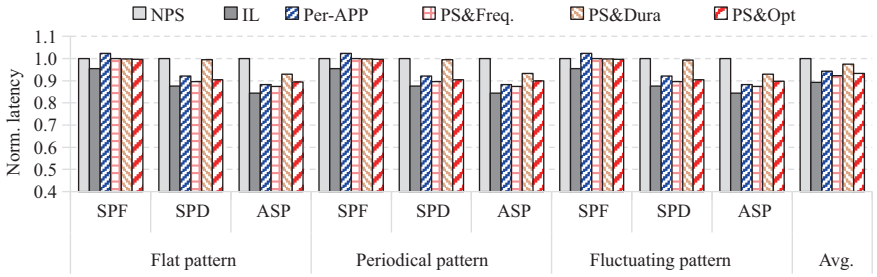


Figure 16 (Color online) Normalized latency of different schemes.

which invokes functions in a round-robin fashion. The frequency of cores in NPS is set as the average value of co-located functions' best-suited frequency while IL sets the highest frequency of cores to pursue ideal latency. Per-APP represents the state-of-the-art application-level power management [20, 21, 23]. It considers functions as a whole application but ignores functions' inherent phases. It can synchronize the best-suited frequency of functions. In addition, we also evaluate the optimization effectiveness of PowerSync. To be more specific, PS&Freq deploys functions considering the synchronization of best-suited frequency. PS&Dura tends to deploy functions with the duration synchronization.

7 Evaluation results

This section quantifies the performance and efficiency of PowerSync with a wide range of configurations.

7.1 Effectiveness of PowerSync

To understand the overall efficacy of our design, we first measure the EDP of different schemes as shown in Figure 14. All results are normalized to NPS. Compared to NPS, the mean EDP of PS&Freq, PS&Dura and PS&Opt is reduced by 19%, 22%, and 29%, respectively. Since IL pursues high performance but ignores the energy efficiency, it presents the highest mean EDP of all schemes (31% higher than PS&Opt). PS&Opt can also reduce 16% EDP compared to Per-APP. Since PS&Freq and PS&Dura only support part of power synchronization, their EDP values are larger than PS&Opt.

In addition to EDP, we further evaluate the average energy and latency (which includes the queue time and duration of functions) of different schemes as shown in Figures 15 and 16. We observe that the energy consumption of IL is higher than other schemes. However, PS&Opt only increases the latency by 4% while

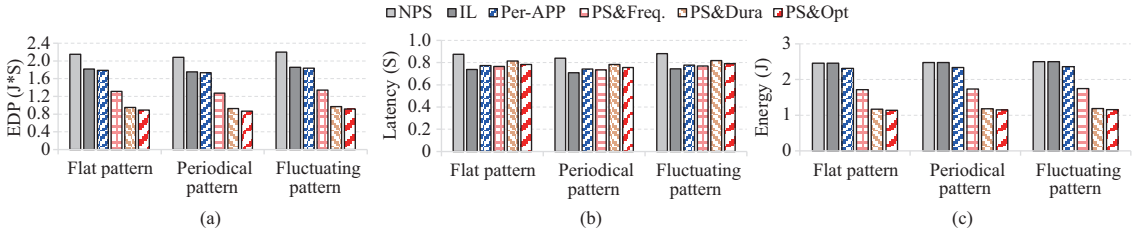


Figure 17 (Color online) Comparison of the energy, latency, and EDP of different schemes. (a) EDP; (b) latency; (c) energy.

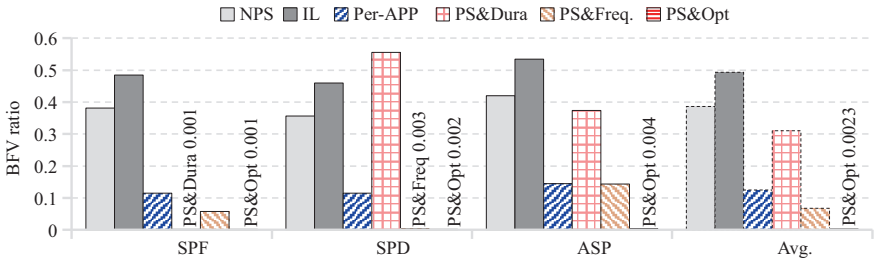


Figure 18 (Color online) Optimization effectiveness under fluctuating invocation pattern.

achieving 37% energy saving compared to IL. Unlike Per-APP, PS&Opt can achieve both lower latency and energy efficiency. On average, PS&Opt have 1% performance improvement and 19% energy saving as well. Since PS&Dura only synchronizes function duration, it would select a sub-optimal frequency; it shows about 3% more latency compared to Per-APP. In Figure 16, we can see that the latency of PS&Opt is increased by less than 10% (which is an acceptable latency degradation in cloud [8]) compared with the best case IL. In addition to the normalized value, we also present the absolute value of the average latency, energy, and EDP using the ASP function pool as shown in Figure 17.

7.2 Best-suited frequency violation

Figure 18 plots the best-suited frequency violation (BFV) ratio of functions. It is defined as the duration that a function is processed without the best-suited frequency divided by the total duration of the function. It is a lower-is-better metric.

As shown in Figure 18, PS&Opt can achieve a best-suited frequency violation ratio of nearly zero with different function pools — lower than all other schemes. The results show that PowerSync can always ensure that functions are processed with the best-suited frequency. Since functions in the SPF pool all have similar best-suited frequencies and PS&Dura co-locates function with synchronized duration, PS&Dura shows similar results as PS&Opt. Thus, the BFV of PS&Dura is also nearly zero under the SPF pool. Moreover, the BFV ratio of PS&Dura (0.31) is higher than PS&Freq (0.07) and Per-APP (0.12) since PS&Freq and Per-APP can select the most suited frequency for functions.

7.3 A glance at asynchronous power

Even if functions with the same best-suited frequency co-locate on the same core, phase Async. and life Async. can still result in low power efficiency. In our evaluation, only PS&Freq and PS&Opt invoke function according to the best-suited frequency. Thus, we record the proportion of phase Async. and life Async. of both schemes as shown in Table 5. We can hardly see phase and life asynchronization with PS&Opt, since it ensures the near-optimal PSS for functions. On average, the proportion of phase Async. and life Async. of PS&Freq are 7% and 13%, respectively. Since the FDD of functions in the SPD pool is zero, PS&Freq group functions by best-suited frequency which can ensure the exact PSS as PS&Opt.

7.4 Impact of threshold on PowerSync

As shown in (1), the setting of differentiation threshold D_t affects the PSSs of functions. To understand the impact of the threshold on PowerSync, we invoke functions in the ASP pools with the fluctuating invocation pattern. Figure 19 shows the EDP and latency under various thresholds. All results are normalized to the value with 0-threshold. It shows that increasing the duration differentiation threshold

Table 5 Ratio of switch Async. and finish Async. (%)

	Phase Async. ratio		Life Async. ratio	
	PS&Freq	PS&Opt	PS&Freq	PS&Opt
SPF	6	**0.12**	9.2	**0.03**
SPD	0.3	**0.2**	0.8	**0.025**
ASP	14	**0.4**	31	**0.04**
Average	7	**0.23**	13.6	**0.032**

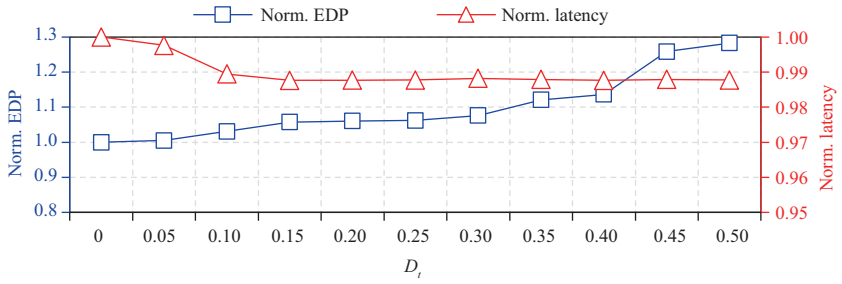


Figure 19 (Color online) Normalized EDP and latency under different FDD thresholds.

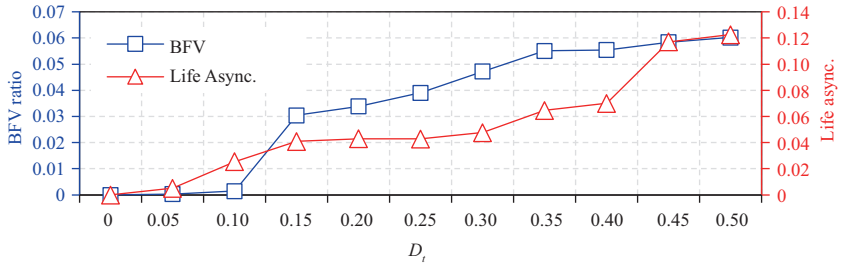


Figure 20 (Color online) Ratio of BFV and life Async. under different FDD thresholds.

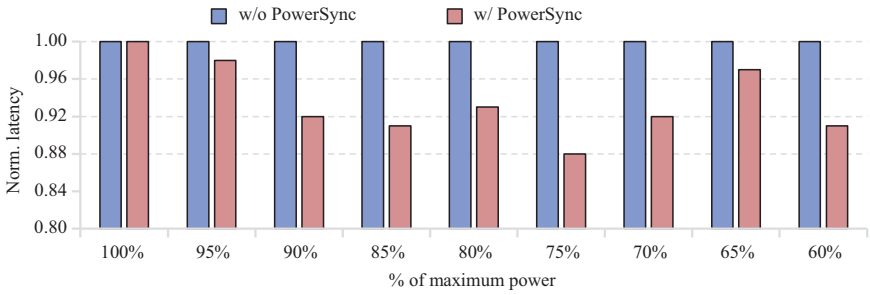


Figure 21 (Color online) Average latency under different power budget.

D_t could reduce the energy efficiency of serverless computing platforms. A lower differentiation threshold implies that functions need more queue time to synchronize the power which would increase the latency. As shown in Figure 19, the latency of functions with 0-threshold only increases 1% compared with 0.5-threshold while decreasing EDP by 30%.

The degradation of EDP with increasing differentiation threshold is caused by higher BFV and life asynchronization when function invocation. In Figure 20, we estimate the ratio of BFV and life asynchronization. With the increasing threshold, the phase switch and the total duration of co-locations are more asynchronous. The BFV ratio changes from 0 to 0.06 when the threshold increases from 0 to 0.5 while the life asynchronization ratio changes from 0 to 0.12.

7.5 Performance of PowerSync under power budget

PowerSync is well poised to improve the performance of serverless functions under power budget. In Figure 21, we estimate the average latency of functions in the ASP pool under the fluctuating invocation

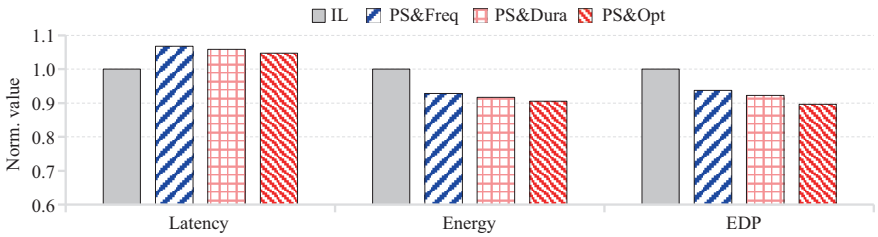


Figure 22 (Color online) Performance comparison on our prototype.

pattern. The maximum power is 72 W per node which allows all cores to process functions at the highest frequency (2.2 GHz). The scheme without PowerSync allocates power for each core with the average value. Under different power budgets, PowerSync achieves better performance. Especially, when the power budget is 75% of the maximum value, PowerSync can reduce function latency by 12%.

8 Discussion and future work

8.1 System prototype analysis

We implement the core part of PowerSync on a scaled-down system. We run experiments based on three-invoker Openwhisk deployment. The workload is ASP and we use `turbostat` to test the energy of the CPU package. The processor power management is important which is studied in prior studies [12, 22]. Additionally, the power consumption of other components is independent of the running frequency. Thus, we only consider to study processors' energy efficiency. The latency, energy, and EDP are shown in Figure 22. The results are normalized to IL. Schemes with PowerSync achieve better energy efficiency than IL. PS&Opt can save 10% energy while increasing only 4% latency compared to IL. Besides, PS&Opt can reduce reduce 11% EDP compared to IL, 3% EDP compared to PS&Freq and 1% EDP compared to PS&Dura.

We also analyze the overhead of PowerSync. To estimate the best-suited frequency and duration of functions, PowerSync adds only about 10 μs to the end-to-end latency. In each invoker, PowerSync needs one core for In-Node Manager to manage and monitor invokers. When checking functions' phase status, In-Node Manager needs to obtain the log information of functions which would add ∼5 ms to functions. In addition, the latency of function migration and frequency change is about ∼15 ms (user-space adjustment) and ∼12 ms on the system. Ideally, the delay of frequency change ranges between ∼10 ns [22]–20 μs [20] and the delay of thread motion is about ∼0.25 μs [24].

8.2 Multi-stage serverless applications

Generally, a serverless function receives its input from the request and returns the message with its result. This implementation of communication with function is feasible when function input and output are both lightweight. However, with the development of serverless computing, there emerge some applications like video processing and DNA visualization whose input or output are heavyweight. Therefore, they should download its input from Cloud Storage such as AWS S3, and upload their results. We analyze 3 representative serverless applications in Figure 23 which are composed of three stages: the download stage, the execution stage, and the upload stage. Figure 23 shows that the three stages of serverless applications present different performance-frequency behavior. The upload stage and especially the download stage have lower sensitivity to CPU frequency than the execution state. It is promising to make use of this property with the idea of PowerSync to synchronize the power requirement of different stages in the scenario of multi-stage serverless applications.

8.3 Multi-processing scenario

In PowerSync, the function will be allocated to a single core, which seems to miss the case of multi-processing. The reason we handle the function in a single-thread way is that serverless functions generally 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. Therefore, it is reasonable for our research to focus on single-thread functions only.

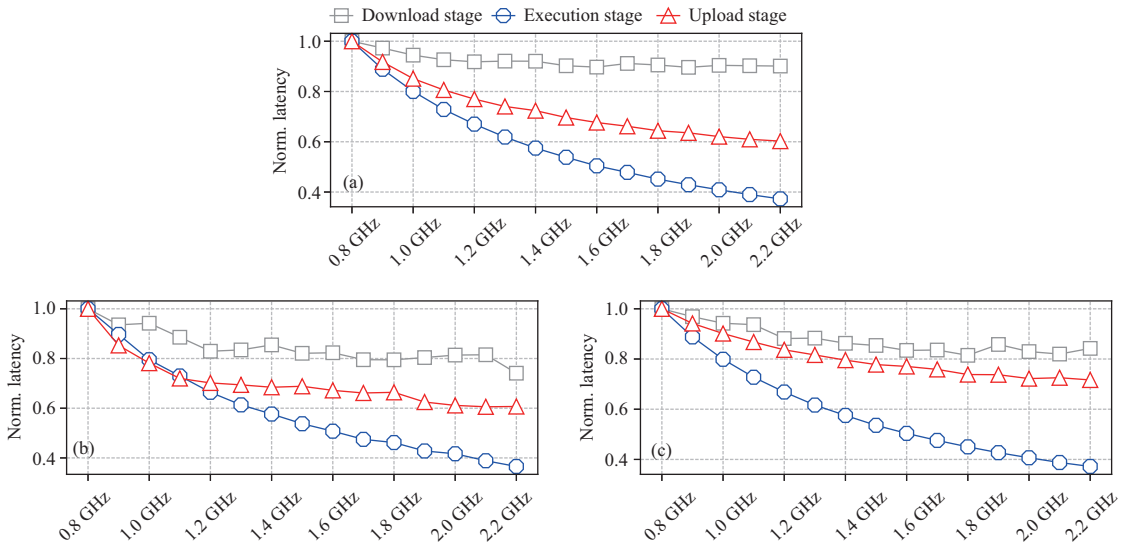


Figure 23 (Color online) Performance-frequency behavior of functions' multi stages. (a) Compression; (b) DNA visualization; (c) video processing.

8.4 Scalability of PowerSync

PowerSync, which includes both the function unification module and the function dispatch module can be regarded as an add-on component in the central scheduler of serverless functions. The function unification module aims to pack up functions in PSS with both the individual and temporal variability considered. The function dispatch aims to allocate the functions set to an idle core. On the controller side, to estimate the best-suited frequency and duration of functions, PowerSync adds only about 10 μs to the end-to-end latency. On the invoker side, PowerSync only makes a few modifications that are to allow an in-node manager to determine which processor functions should be allocated and manage frequency tuning in comparison to previous serverless systems like OpenWhisk. Therefore, the scalability of the serverless system will not be reduced after implementing PowerSync to achieve power efficiency.

8.5 Function duration estimation

We adopt a linear regression model to predict functions' duration due to its low overheads and acceptable performance in our benchmark. However, when it comes to some nonlinear functions, the linear regression model might not be as effective. Towards these functions, some complex machine learning models should be adopted. Since the focus of our research is to propose the concept of Power Synchronization to make serverless computing more efficient, we will not delve into this problem any further and leave it as an open question for other researchers to solve.

Based on the observations from the characterization of FaaS workload of Azure function we learn that 18.6% of functions generate 99.6% of invocations [3]. If we want to implement PowerSync in the real production environment where functions are far more than our benchmark, we can add a filter mechanism to record functions' invoked frequency and filter out those infrequently invoked functions, which avoids unnecessary training. For the frequently invoked functions, it is worthwhile to train a model, since they act as the main contribution to upcoming invocations and the model size is only kB level.

9 Conclusion

While serverless computing is gaining popularity and bringing many benefits, it poses new challenges to achieving high power efficiency of functions. In this paper, we consider power as a key driving issue for function management. We identify unique variability of functions which is largely overlooked in prior works. We propose PowerSync, a novel power management framework that can exploit the optimal operational setting of various serverless functions. We show that the design of PowerSync can improve the energy efficiency of functions by up to 16% without performance loss compared to the state-of-the-art

per-application power management strategy. We expect that our design can provide insights into the design of highly efficient and sustainable cloud-native infrastructures.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 62122053) and Shanghai S&T Committee Rising-Star Program (Grant No. 21QA1404400).

References

1 Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud programming simplified: a berkeley view on serverless computing. 2019. ArXiv:1902.03383
2 Shahrad M, Balkind J, Wentzlaff D, et al. Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019. 1063–1075
3 Shahrad M, Fonseca R, Goiri Í, et al. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In: Proceedings of USENIX Annual Technical Conference, 2020. 205–218
4 Wang L, Li M, Zhang Y, et al. Peeking behind the curtains of serverless platforms. In: Proceedings of USENIX Annual Technical Conference, 2018. 133–146
5 Ioana B, Paul C, Kerry C, et al. Serverless computing: current trends and open problems. In: Research Advances in Cloud Computing. Berlin: Springer, 2017. 1–20
6 Tariq A, Pahl A, Nimmagadda S, et al. Sequoia: enabling quality-of-service in serverless computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020. 311–327
7 Agache A, Brooker M, Iordache A, et al. Firecracker: lightweight virtualization for serverless applications. In: Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, 2020. 419–434
8 Suresh A, Gandhi A. ServerMore: opportunistic execution of serverless functions in the cloud. In: Proceedings of the ACM Symposium on Cloud Computing, 2021. 570–584
9 Apostolopoulos P A, Tsiropoulou E E, Papavassiliou S. Risk-aware social cloud computing based on serverless computing model. In: Proceedings of IEEE Global Communications Conference (GLOBECOM), 2019. 1–6
10 Kanev S, Darago J P, Hazelwood K, et al. Profiling a warehouse-scale computer. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015. 158–169
11 Margaritov A, Gupta S, Gonzalez-Alberquill R, et al. Stretch: balancing QoS and throughput for colocated server workloads on SMT cores. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2019. 15–27
12 Zhang L, Li C, Wang X K, et al. FIRST: exploiting the multi-dimensional attributes of functions for power-aware serverless computing. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2023. 864–874
13 Li Z J, Liu Y S, Guo L S, et al. FaaSFlow: enable efficient workflow execution for function-as-a-service. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022. 782–796
14 Jia Z P, Emmett W. 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. 152–166
15 McCracken S. How to architect for sustainability in a cloud native environment. 2022. https://www.contino.io/insights/cloud-native-sustainability
16 Judge P. Ovhcloud, orange, and inria join french research project for greener clouds. 2022. https://www.datacenterdynamics.com/en/news/ovhcloud-orange-and-inria-join-french-research-project-for-greener-clouds/
17 Fuerst A, Sharma P. FaasCache: keeping serverless computing alive with greedy-dual caching. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. 386–400
18 Du D, Yu T, Xia Y, et al. Catalyzer: sub-millisecond startup for serverless computing with initialization-less booting. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020. 467–481
19 Sharma P. Challenges and opportunities in sustainable serverless computing. In: Proceedings of Workshop on Sustainable Computer Systems Design and Implementation, 2022
20 Hou X F, Li C, Liu J C, et al. ANT-Man: towards agile power management in the microservice era. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, 2020. 1–14
21 Yang H, Chen Q, Riza M, et al. Powerchief: intelligent power allocation for multi-stage applications to improve responsiveness on power constrained CMP. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017
22 Hsu C H, Zhang Y Q, Laurenzano M, et al. Adrenaline: pinpointing and reining in tail queries with quick voltage boosting. In: Proceedings of the 21st International Symposium on High Performance Computer Architecture, 2015. 271–282
23 Guliani A, Swift M M. Per-application power delivery. In: Proceedings of the Fourteenth EuroSys Conference, 2019. 1–16
24 Rangan K K, Wei G Y, Brooks D. Thread motion. SIGARCH Comput Archit News, 2009, 37: 302–313
25 Gendler A, Knoll E, Sazeides Y. I-DVFS: instantaneous frequency switch during dynamic voltage and frequency scaling. IEEE Micro, 2021, 41: 76–84

26 Chen S, Delimitrou C, Martínez JF. Parties: QoS-aware resource partitioning for multiple interactive services. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, 2019. 107–120

27 Mars J, Tang L J, Hundt R, et al. 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. 248–259

28 Delimitrou C, Kozyrakis C. Quasar: resource-efficient and QoS-aware cluster management. SIGPLAN Not, 2014, 49: 127–144

29 Meisner D, Gold B T, Wenisch T F. PowerNap: eliminating server idle power. SIGARCH Comput Archit News, 2009, 37: 205–216

30 Haque M E, He Y, Elnikety S, et al. Exploiting heterogeneity for tail latency and energy efficiency. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017. 625–638

31 Wong D. Peak efficiency aware scheduling for highly energy proportional servers. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture, 2016. 481–492

32 Lo D, Cheng L Q, Govindaraju R, et al. Towards energy proportionality for large-scale latency-critical workloads. In: Proceedings of the 41st International Symposium on Computer Architecture, 2014. 301–312

33 Kannan R S, Subramanian L, Raju A, et al. Grandslam: guaranteeing SLAS for jobs in microservices execution frameworks. In: Proceedings of the 14th EuroSys Conference, 2019. 1–16

34 Suresh A, Gandhi A. Fnsched: an efficient scheduler for serverless functions. In: Proceedings of the 5th International Workshop on Serverless Computing, 2019. 19–24

35 Mohan A, Sane H, Doshi K, et al. Agile cold starts for scalable serverless. In: Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing, 2019

36 Zhang L, Feng W, Li C, et al. Tapping into NFV environment for opportunistic serverless edge function deployment. IEEE Trans Comput, 2022, 71: 2698–2704

37 Yu T, Liu Q, Du D, et al. Characterizing serverless platforms with serverlessbench. In: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020. 30–44

38 Meisner D, Wenisch T F. DreamWeaver: architectural support for deep sleep. SIGPLAN Not, 2012, 47: 313–324

39 Brown L, Renninger T. Cpupower. 2021. https://linux.die.net/man/1/cpupower

40 Kim J, Lee K. Functionbench: a suite of workloads for serverless cloud function service. In: Proceedings of the 12th International Conference on Cloud Computing, 2019. 502–504