FIRST: Exploiting the Multi-Dimensional Attributes of Functions for Power-Aware Serverless Computing

Lu Zhang[1], Chao Li[1], Xinkai Wang[1], Weiqi Feng[1], Zheng Yu[1], Quan Chen[1], Jingwen Leng[1], Minyi Guo[1],
Pu Yang[2], Shang Yue[2]
[1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University* [2]*Tencent*
Emails: {luzhang, unbreakablewxk}@sjtu.edu.cn, {lichao, chen-quan, guo-my}@cs.sjtu.edu.cn , {dowpuyang}@tencent.com

***Abstract*—Emerging cloud-native development models raise new challenges for managing server performance and power at microsecond scale. Compared with traditional cloud workloads, serverless functions exhibit unprecedented heterogeneity, variability, and dynamicity. Designing cloud-native power management schemes for serverless functions requires significant engineering effort. Current solutions remain sub-optimal since their orchestration process is often one-sided, lacking a systematic view. A key obstacle to truly efficient function deployment is the fundamental wide abstraction gap between the upper-layer request scheduling and the low-level hardware execution.**

In this work, we show that the optimal operating point (OOP) for energy efficiency cannot be attained without synthesizing the multi-dimensional attributes of functions. We present FIRST, a novel mechanism that enables servers to better orchestrate serverless functions. The key feature of FIRST is that it leverages a lightweight Internal Representation and meta-Scheduling (IRS) layer for collecting the maximum potential revenue from the servers. Specifically, FIRST follows a pipeline-style workflow. Its frontend components aim to analyze functions from different angles and expose their key features to the system. Meanwhile, its backend components are able to make informed function assignment decisions to avoid OOP divergence. We further demonstrate the way to create extensions based on FIRST to enable versatile cloud-native power management. In total, our design constitutes a flexible management layer that supports power-aware function deployment. We show that FIRST could allow 94% functions to be processed under the OOP, which brings up to 24% energy efficiency improvements.

***Index Terms*—FaaS, power management, multicore**

I. Introduction

We have entered a new era in which serverless computing (or Function as a Service, FaaS) is redefining the cloud [14], [28], [32]. A serverless function is a bundle of code that can be invoked through the Internet. To achieve higher server utilization and make the serverless model profitable, cloud vendors often colocate functions [1], [32]. To make the serverless model more cost-effective, the underlying cloud servers should be carefully managed for greater energy efficiency [24], [33], [38]. Further, green consciousness is quickly making its way into the cloud. IT companies now have strong incentives to limit system power to improve sustainability.

Till now, very limited prior work exists on the power-efficient deployment of a variety of serverless functions. The lightweight nature of serverless computing allows multiple functions that are drastically different to be running on a single processor core [23], [40]. In this case, providing the most desirable configuration for every single function requires μs-scale power adjustment to fit users' fleeting requests. With current performance/power scaling speed, it is still an impossible mission [11] under massive diversified functions.

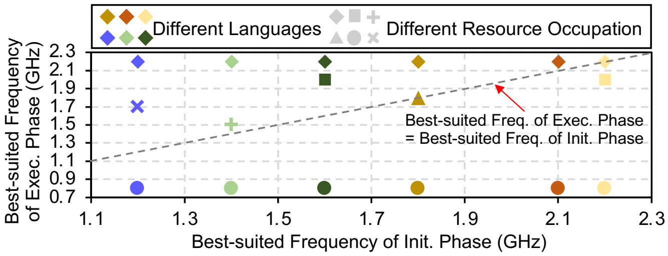
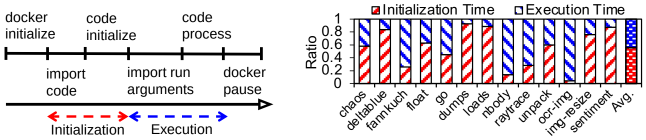


Fig. 1. Variety of functions' best-suited frequency

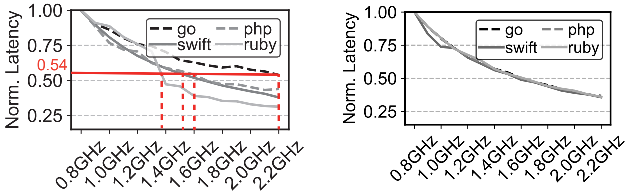
Importantly, our characterization (detailed in II-A) reveals that the power behavior of serverless functions is affected by multiple attributes. 1) *Function Lifecycle*: functions quickly goes through several major phases in its lifecycle [4], [29]. Each phase has different behavior which requires special care when managing the power. 2) *Runtime Environment*: Serverless platforms allow developers to write their FaaS code in different languages [28], [32]. A function's sensitivity to performance scaling can be language-specific. 3) *Resource Occupation*: functions demand different types of resources. Each function has a different best-suited frequency. Blindly scheduling functions may disrupt the established operating states and cause inefficient frequency adjustments. In Figure 1, our evaluation of over a dozen realistic functions demonstrates a highly diversified workload portfolio. Each function demands a different frequency which changes with its lifecycle phase.

In this work we argue that FaaS platforms need to consider multiple key attributes and make informed power management decisions – a process we refer to as *function attribute synthesis*. Through attribute synthesis, we aim to identify the optimal operating point (OOP) of the FaaS platform. Traditional request scheduling and micro-architectural dynamic scheduling both abstract away the abundant attributes of functions. There is a wide *abstraction gap* between the upper-layer software application and the low-level hardware execution. As a result, existing power management policies are oblivious to function attributes and they unavoidably lead to a situation in which co-located functions have drastically diverged OOP. Thus, the



(a) The phases of functions (b) Duration breakdown of functions

Fig. 2. Different phases of serverless functions



(a) Initialization phase (b) Execution phase

Fig. 3. Language type affects latency differently

state-of-the-art designs may achieve high server utilization (function density) but never the best energy efficiency, especially in the power budget environment.

This paper takes the first step to empower the FaaS platform to support power-aware function scheduling on processor cores. We propose to reduce the wide abstraction gap with a lightweight management layer. Our novelty is two-fold. First, we create *function internal representations* enriched with multi-dimensional attributes from the code runtime, OS, and hardware architecture. Second, we introduce *function meta-scheduling*, which aims to fine-tune the allocation process at the server level through an efficient pipeline-style workflow.

We present *Function Internal Representation and meta-Scheduling Toolset (**FIRST**)* for power-aware function deployment. It includes a mini set of operators that play different roles. Specifically, FIRST employs two front-end operators for function analysis and two back-end operators for workload control. Its pipeline-style workflow allows one to perform function orchestration in power-efficient ways. This paper makes the following contributions:

- **Analysis:** We analyze the performance-power implications of serverless functions and demonstrate the rationale for maintaining a converged optimal operating point.
- **Design:** We introduce *FIRST*, a novel mechanism for fine-tuning the function placement process with a versatile, pipeline-like working flow. We further enhance it to support more flexible function power management.
- **Evaluation:** We build a proof-of-concept testbench of FIRST and show that it could improve energy efficiency by more than 24% with minor performance overhead.

The rest of this paper is organized as follows: Section II further motives our work. Section III proposes FIRST. Section IV describes experimental methodologies. Section V presents evaluation results. Section VI analyzes FIRST on real machines. Finally, Section VII discusses related work and Section VIII concludes the paper.

II. Abstraction Gap: Implications

We start by analyzing multiple attributes unexploited in function power management due to the abstraction gap. We then show that it is infeasible to reach a desirable operating point without coordinated orchestration.

A. Multi-Dimensional Attributes

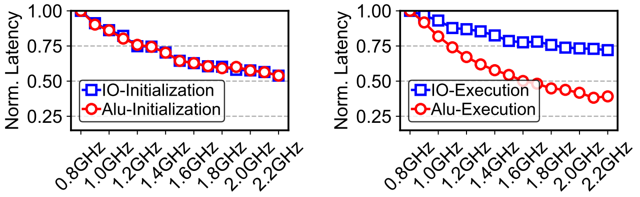
We analyze the power behavior of representative serverless functions (detailed in Table II) from different perspectives.

1) Function Phase Matters: Although a serverless function can be deployed in various ways [1], [8], [14], [32], [37], its invocation often consists of multiple phases as shown in Figure 2(a). Among all the phases, the initialization and execution are the most important. Figure 2(b) shows the latency breakdown of a set of functions. We can see that the initialization time is generally comparable to the execution time, accounting for over half of the total latency across our evaluated benchmarks. The two phases have different processing flows and characteristics. During initialization, containers import codes with libraries and initialize them using language-specific interpreters. In contrast, the execution of functions mainly processes user code, which emphasizes different types of resource occupation. As detailed below (Figures 3 and 4), **one needs to treat the initialization and execution phases *differently* when managing the energy of serverless functions**.

2) Language Runtime Matters: Developers generally send functions written in a variety of high-level programming languages [7], [8], [32] to the serverless platform. To understand the influence of the runtime environment on power management, we implement `ALU` (a compute-intensive function) using different languages: Go, PHP, Swift, and Ruby. We record the time of function initialization and execution under different frequencies. We observe different performance trends in response to processor frequency adjustment. For example, in Figure 3(a), the latency of Ruby function drops significantly when the frequency changes from 1.0GHz to 2.2GHz. The latency of Go function does not change that much as the frequency increases. For a given QoS (e.g., 0.54x latency in the figure), the four `ALU` functions manifest different requirements on the minimum frequency. If we look at the execution time, however, the performance trends are very similar across all the evaluated languages, as shown in Figure 3(b). It is evident that **a function's performance-power behaviors are language-specific in the initialization phase while influenced less by language type in the execution phase.**

3) Resource Type Matters: A serverless platform usually hosts a variety of functions that could be invoked at any time. The resource type (e.g., CPU-bound / IO-bound) [12], [16] may affect functions' power-performance behavior as well. To see how the use of resources can impact system power efficiency, we invoke two different functions: `ALU` (CPU-bound) and `FileIO` (IO-bound). We measure the time spent on the initialization and execution phase, respectively.

The resource type can affect functions' power-performance behavior in a way directly opposite to the language runtime.



(a) Initialization phase

(b) Execution phase

Fig. 4. Resource type affects latency differently

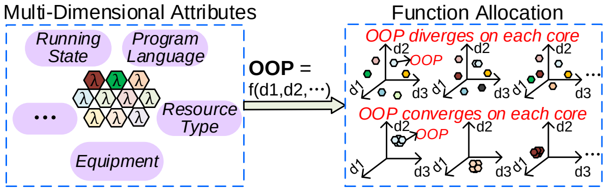


Fig. 5. Optimal operating point (OOP) divergence

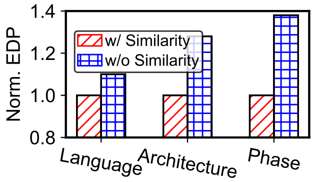


Fig. 6. Normalized EDP of different function deployment scenarios

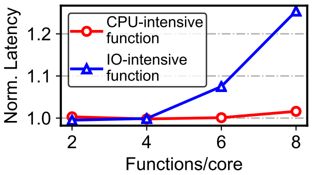


Fig. 7. Impact of co-locating functions with the same resource type

Figure 4(a) plots the normalized initialization time of two functions under different processor frequencies. We observe that it is the initialization time that keeps the same tendency as frequency increases for both function types. When it comes to execution time, as shown in Figure 4(b), the two functions differ greatly. Although both functions slope downward from left to right, the latency change of `FileIO` is much lower than `ALU`. The above results show that **resource type has negligible influence during function initialization, but it dominates when a function is executing**.

Summary: Serverless computing presents a highly heterogeneous and dynamic environment. The power behavior of functions is affected by multi-dimensional attributes including resource types, function phases, language runtime, etc. Efficiently reining cloud-native applications can be a daunting task without grasping the underlying multi-dimensional attributes.

B. Optimal Operating Point Divergence

As mentioned, functions' optimal operating points (OOPs), which provides the energy efficiency, are different with different attributes. We observe that a CPU core can be assigned with a group of functions that have drastically different OOPs, which we call *OOP divergence* of functions. This phenomenon makes it difficult or impossible for a core to find an optimal V/F setting. CPU time-multiplexing does not solve this issue since existing hardware cannot quickly adjust its power states facing massive diversified applications [21].

In Figure 5 we compare OOP divergence and convergence. The ideal function co-location strategy needs to synthesize the multi-dimensional attributes to group functions whose OOPs are the same. This will lead to OOP convergence, which prevents the system from making compromised frequency scaling decisions on a processor core.

To quantitatively measure the OOP divergence of the system, we define a simple metric, *Function Chaos Factor* (FCF):

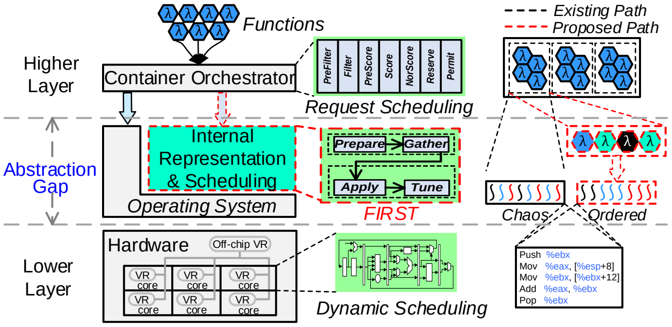
$$FCF = \frac{1}{N}\sum_{i}^{N}\alpha_i\sqrt{\frac{1}{M}\sum_{j}^{M}(x_{ij}-\bar{x}_i)^2} \tag{1}$$

In the above equation, N is the dimension of information that we extract and α_i is the weight of dimension i contributed to FCF. M is the number of functions running and x_{ij} is the value of function j on the dimension i. The metric can be used to identify and quantify the OOP divergence of the system. It is basically a measure of the diversity/disorder of functions based on multiple indices ($0 \leq$ FCF ≤ 1, 0 indicates the orderly function execution while 1 indicates the irregular function execution).

To illustrate how diversity/disorder affects the efficiency of a system, we conduct a case study as shown in Figure 6. We use energy-delay product (EDP) to present energy efficiency, which offers equal weight to both energy and performance. Each function has its best-suited frequency in each dimension: language, architecture, and phase. Take the language dimension as an example, we run heterogeneous functions in two ways. 1) With similarity: we run two Go functions on one core, and two Ruby functions on another. We configure each core with a single best-suited frequency. 2) Without similarity: each core runs a set of functions written in different languages and different functions may have different OOPs. As shown in Figure 6, due to OOP divergence, the overall processor EDP increases by 10%-38%. It is evident that orderly grouped functions have the best efficiency.

Note that resource contention can be a minor issue with mild co-location on a core. Figure 7 shows the impact of function co-location. Our baseline scheme co-locates functions of different resource type (50% CPU-intensive and 50% IO-intensive) on a core. We show the performance of co-locating functions of the same resource type, which is normalized to the performance of that resource type in our baseline scheme. For example, in the figure, co-locating 8 IO-intensive functions result in 25% latency increase compared to the average latency of IO-intensive functions in our baseline (4 IO-intensive + 4 CPU-intensive). For CPU-intensive functions, co-location introduces negligible performance difference even if we increase the co-location density to 8 functions per core. Differently, IO-intensive functions are more sensitive in this regard. To avoid severe contention, in this work we limit co-location density to 4 functions per core.

Summary: Without a deep understanding of the massive diversified functions, it is difficult to make the best power allocation decision. It is better to avoid OOP divergence



```
Push  %ebx
Mov   %eax, [%esp+8]
Mov   %ebx, [%ebx+12]
Add   %eax, %ebx
Pop   %ebx
```

Fig. 8. An overview of FIRST

and ensure orderly function assignment. Smart function redistribution based on function attribute synthesis demonstrates remarkable efficiency and limited resource contention.

III. Power-Aware Function Processing

We present **F**unction **I**nternal **R**epresentation and meta-**S**cheduling **T**oolset (FIRST), a novel processor power management scheme for serverless functions.

A. Overview of FIRST

In Figure 8 we provide an overview of FIRST. The proposed FIRST design has three salient features:

- *Function Internal Representation*. FIRST constructs a new abstraction layer in-between the upper function orchestration layer and the low-level hardware execution layer. In the new layer, function internal representation is created as an extended ID with enriched information of the original serverless function. It encapsulates key configurations and/or parameters of functions, which makes attribute synthesis possible. Function internal representation is the foundation of FIRST and it allows one to tap into the multi-dimensional attributes of functions that are largely overlooked by conventional designs.
- *Meta-Scheduling*: On top of the internal representation, FIRST provides a pipeline-style workflow that can manipulate function distribution. It is composed of a mini set of four operators: `Prepare`, `Gather`, `Map`, `Tune`. They can be classified as either front-end analytic modules or back-end control modules. We call the above process ”meta-scheduling” since it goes beyond traditional scheduling and it seeks to fine-tune the orchestration process of functions. It fills a critical void between the macroscopic request scheduling (for utilization) and the microscopic instruction scheduling (for performance).
- *OOP Convergence*: Our goal is to maintain the greatest efficiency for the greatest number of functions. We choose a *utilitarian* approach to function meta-scheduling. As mentioned before, the OOP can be determined by analyzing the attributes of a function and functions may have different OOPs. The core can best serve its guest functions only if they have converged OOP. Maintaining OOP convergence improves efficiency and stability.

B. The IRS Workflow

FIRST is composed of a mini set of four operators. They can be classified as either front-end analytic modules or back-end control modules, as shown in Figure 9. The front-end operators (`Prepare` and `Gather`) are responsible for information gathering and analysis. The back-end operators (`Map` and `Tune`) are responsible for controlling the execution of the thread running on the servers.

1) ***`Prepare`****:* The `Prepare` operator extracts the key information of functions and transforms the data into a tri-tuple. As shown in Figure 9(a), it spans the major stack of the computing system. We mainly assess functions from three dimensions – the hardware, the OS, and the source code.

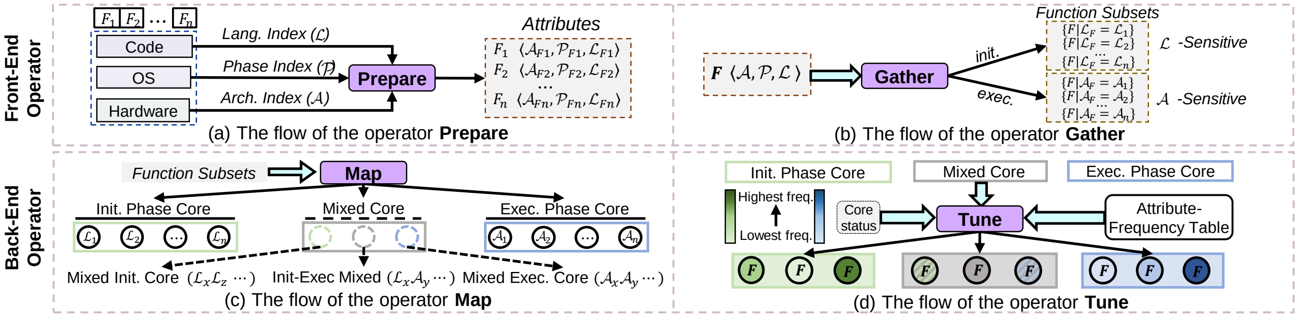
At the code level, `Prepare` extracts a language index ($\mathcal{L}$) by analyzing container image information when functions are invoked. This information keeps unchanged throughout the functions' lifespan. Meanwhile, it periodically detects the current status ($\mathcal{P}$) of the function by interacting with the OS. We insert tags into function runtime libraries to capture important phases and `Prepare` creates a phase index $\mathcal{P}$ by analyzing the container log data. Unlike $\mathcal{L}$, $\mathcal{P}$ is not a constant. At the hardware level, `Prepare` reads low-level performance counters that are available. It determines an architecture index ($\mathcal{A}$) which reflects the resource type that a function depends on. Finally, `Prepare` is able to generate a internal representation $\langle\mathcal{A}, \mathcal{P}, \mathcal{L}\rangle$ for the input function.

2) ***`Gather`****:* The `Gather` operator is responsible for identifying the best resource sharing schemes based on the internal representation offered by `Prepare`. By carefully co-locating functions, we can prevent OOP divergence on a processor core.

The `Gather` operator is performed in a unique two-step manner. This is because the power behavior of a function changes in its lifecycle (detailed in Section II). We firstly gather functions by $\mathcal{P}$, which results in two groups: 1) functions in the initialization phase, and 2) functions in the execution phase. In the second step, we further break each group into several subsets. For functions in the initialization phase ($\mathcal{L}$-sensitive), we classify them based on the code runtime. For functions in the execution phase ($\mathcal{A}$-sensitive), we classify them based on resource consumption behavior.

3) ***`Map`****:* The `Map` operator maps function subsets to cores. The basic principle behind this is that newly added functions should have the minimum impact on the current state of the hardware. We want to keep the processor running in a relatively stable manner. Figure 9(c) illustrates the mechanism behind `Map`. The operator basically manages the function-core affinity hierarchically. Given the function groups created by `Gather`, `Map` divides all the available cores into three categories – *Init. Phase Core*, *Exec. Phase Core* and *Mixed Core*. The *Init. Phase Core* hosts functions that are initializing; we bind functions of the same $\mathcal{L}$ to them. Similarly, functions running on the *Exec. Phase Core* are all in their execution phase; we bind functions of similar $\mathcal{A}$ to these cores.

It is possible that some cores may have to host functions with little similarity. We use *Mixed Core* to refer to them. There are three types of *Mixed Core* – *Mixed Init. Core*, *Mixed*


Front-End Operator
F_1 F_2 … F_n
Code
OS
Hardware
Lang. Index ($\mathcal{L}$)
Phase Index ($\mathcal{P}$)
Arch. Index ($\mathcal{A}$)
Prepare
Attributes
F_1 $\langle \mathcal{A}_{F1}, \mathcal{P}_{F1}, \mathcal{L}_{F1} \rangle$
F_2 $\langle \mathcal{A}_{F2}, \mathcal{P}_{F2}, \mathcal{L}_{F2} \rangle$
…
F_n $\langle \mathcal{A}_{Fn}, \mathcal{P}_{Fn}, \mathcal{L}_{Fn} \rangle$
F $\langle \mathcal{A}, \mathcal{P}, \mathcal{L} \rangle$
Gather
init.
exec.
Function Subsets
$\{F|\mathcal{L}_F = \mathcal{L}_1\}$
$\{F|\mathcal{L}_F = \mathcal{L}_2\}$
…
$\{F|\mathcal{L}_F = \mathcal{L}_n\}$
$\mathcal{L}$ -Sensitive
$\{F|\mathcal{A}_F = \mathcal{A}_1\}$
$\{F|\mathcal{A}_F = \mathcal{A}_2\}$
…
$\{F|\mathcal{A}_F = \mathcal{A}_n\}$
$\mathcal{A}$ -Sensitive
Back-End Operator
Function Subsets
Map
Init. Phase Core
Mixed Core
Exec. Phase Core
$\mathcal{L}_1$ $\mathcal{L}_2$ … $\mathcal{L}_n$
$\mathcal{A}_1$ $\mathcal{A}_2$ … $\mathcal{A}_n$
Mixed Init. Core ($\mathcal{L}_x\mathcal{L}_z \cdots$)
Init-Exec Mixed ($\mathcal{L}_x\mathcal{A}_y \cdots$)
Mixed Exec. Core ($\mathcal{A}_x\mathcal{A}_y \cdots$)
Init. Phase Core
Mixed Core
Exec. Phase Core
Highest freq.
Lowest freq.
Core status
Tune
Attribute-Frequency Table
F F F
F F F
F F F

(a) The flow of the operator **Prepare**

(b) The flow of the operator **Gather**

(c) The flow of the operator **Map**

(d) The flow of the operator **Tune**

Fig. 9. Key resource management operators defined by FIRST. It carefully maps function sets to processor cores

Exec. Core and *Init-Exec. Core*. Here, *Mixed Init. Cores* host functions with different $\mathcal{L}$ index but all in the initialization phase. *Mixed Exec. Cores* host functions with different $\mathcal{A}$ index but all in the execution phase. In the worst case, we co-locate functions of different phases (i.e., *Init-Exec. Core*) which often exhibits the least efficiency.

4) ***`Tune`****:* Finally, the `Tune` operator determines appropriate power levels for each CPU, as shown in Figure 9(d). For the *Init. Phase Core* and *Exec. Phase Core*, we directly set the OOP which ensures the highest efficiency.

For the *Mixed Core*, `Tune` determines the frequency of each core according to an Attribute-Frequency Table, which calculates an efficiency score for each attribute-frequency pair. The score is based on system profiling data. As shown in Figure 10, a function with specific attributes is scored at each frequency level. As an example, the figure shows four functions with an optimal frequency setting of {1,3,5,4}, respectively. The simplest way is to use the minimum, maximal or average value of the optimal frequency of the functions on the *Mixed Core*. On the right side of Figure 10, we can see the overall efficiency score of the three ways are 3.1, 3.1, 2.9 respectively. In this example, the optimal frequency setting should be 2 which brings the efficiency score to 3.4. For each *Mixed Core*, we adopt the frequency that provides the highest efficiency score.

C. Implementation and Optimization

Figure 11 depicts the overall system architecture. As functions arrive at a server node, they will be examined and fine-tuned by the IRS layer. Specifically, each server manages a local meta-scheduling pipeline for the incoming functions. It captures $\langle \mathcal{A}, \mathcal{P}, \mathcal{L} \rangle$ information through the operator `Prepare`. Afterwards, the operator `Gather` takes $\langle \mathcal{A}, \mathcal{P}, \mathcal{L} \rangle$ as the input to form a number of sorted function subsets. By default, each subset will be assigned to the same core with the `Map` operator which oversees the degree of disorder of the multi-core. `Map` uses a Core Label Table to associate function IDs in the function subset with different cores. Lastly, the `Tune` operator assigns functions to the specified core using `cgroups` and adjusts core frequency through *DVFS*.

During runtime, FIRST uses two metrics to track and assess the system status and optimization effectiveness: 1) core chaos factor (CCF), and 2) core occupation rate (COR).

$$CCF = \frac{C_{mixed}}{C_{active}}, \quad COR = \frac{C_{active}}{C_{total}} \qquad (2)$$

In the above equation, C_{mixed} and C_{active} are the number of *Mixed Cores* and active cores, respectively; C_{total} is the number of cores. CCF measures the disorder degree of function assignment and COR reflects general system utilization. Both metrics are easy to calculate. The core status module shown in Figure 9(d) contains CCF and COR information.

Efficient meta-scheduling can be tricky in a highly dynamic FaaS environment. For example, consolidating functions saves power, but it can also lead to more *Mixed Cores* which has low performance-per-watt due to OOP divergence of functions. Therefore, it is often necessary to make subtle yet non-trivial enhancements to the meta-scheduling pipeline.

1) Front-End Enhancement: We define additional modes for `Prepare`/`Gather` to enable more flexible processing.

`Prepare+`: With the plus sign, we mean to provide a new Tidal-Lane Mode for the prepare operation. In its default settings, FIRST allows one to analyze a fixed amount of functions through `Prepare`. When the incoming traffic grows or the current running functions reduce quickly, it is preferable to adjust the batch size for attribute synthesis. `Prepare+` supports such reconfiguration. It also identifies latency-sensitive functions based on either user inputs or history information. As shown in Figure 12(a), the newly invoked functions will not be assigned immediately if they have relaxed deadlines.

`Gather+`: We provide a new Fast-Lane Mode for the gather operation. For functions that demand responsiveness, we can launch them as quickly as possible by skipping a few time-consuming operations. In this mode, the system provides special treatment to functions that are marked as latency-sensitive. As shown in Figure 12(b), once there is spare space on the core, the selected function will be immediately scheduled. Note that, in this mode, the number of *Mixed Cores* will increase due to the insertion of a function of different attributes. To minimize this side-effect, we need to select the most suitable function for running in this mode.

2) Core State Analysis: In Figure 13, we compare the Tidal-Lane mode and the Fast-Lane mode by further analyzing the core occupation rate (COR) and core chaos factor (CCF). We consider a processor overwhelmed by massive functions

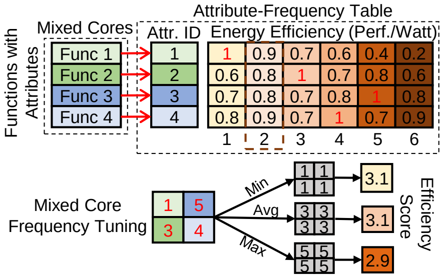


Fig. 10. Frequency tuning of *Mixed Cores*

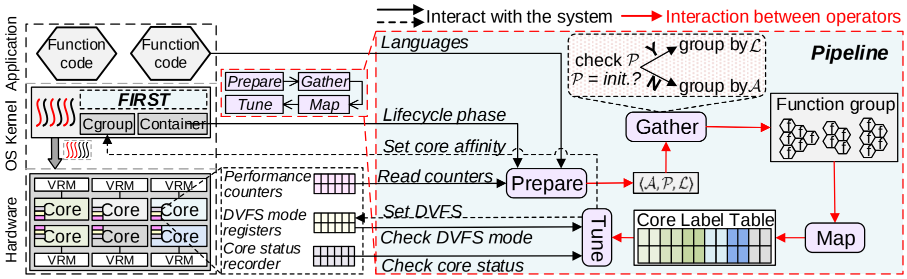
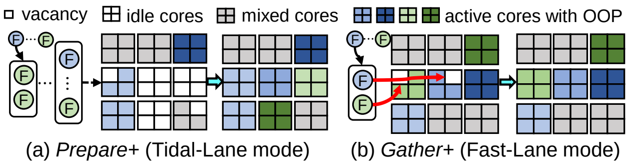
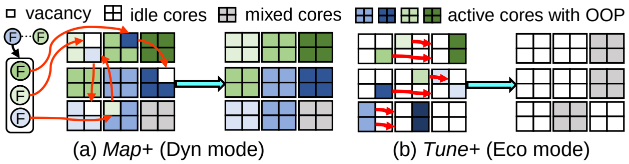


Fig. 11. Overview of the FIRST architecture implemented for a multi-core serverless computing platform



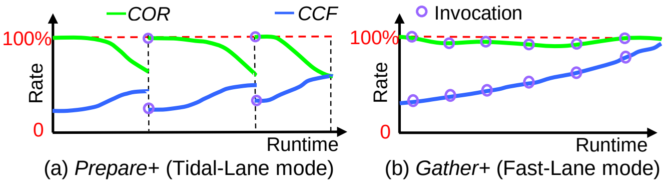
(a) *Prepare+* (Tidal-Lane mode) (b) *Gather+* (Fast-Lane mode)

Fig. 12. Comparison of enhanced front-end operators



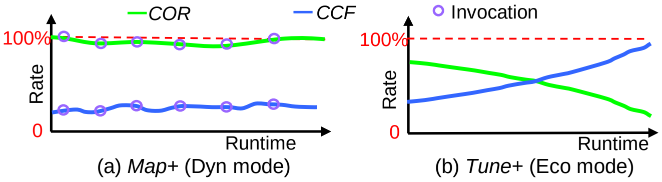
(a) *Map+* (Dyn mode) (b) *Tune+* (Eco mode)

Fig. 14. Comparison of enhanced back-end operators



(a) *Prepare+* (Tidal-Lane mode) (b) *Gather+* (Fast-Lane mode)

Fig. 13. The core occupation rate and core chaos factor under enhanced front-end operators: (a) Tidal-Lane mode; (b) Fast-Lane mode



(a) *Map+* (Dyn mode) (b) *Tune+* (Eco mode)

Fig. 15. The core occupation rate and core chaos factor under enhanced back-end operators: (a) Dyn mode; (b) Eco mode

(e.g., non-empty function queue). With the Tidal-Lane Mode, it becomes easier to maintain a small number of *Mixed Cores*. Its COR declines as the number of idle cores grow during runtime. If the system invokes new functions in bulk, all of the idle cores will be occupied by new function sets which bring COR back to 100%. In addition, it reduces *Mixed Cores* at the beginning of each invocation, thereby bringing down CCF periodically. Since the Fast-Lane mode invokes new functions whenever possible, its COR is very close to 100%. It also exhibits a monotonously increasing CCF curve. This is because the status of each core changes as the phase of functions moves forward. If any of the functions changes its phase, the associated core may become a *Mixed Core*.

3) Back-End Enhancement: Our back-end extension can to provide a better trade-off between OOP and power saving.

`Map+`: With `Map+`, we aim to dynamically re-map function during runtime (i.e. Dyn Mode). When functions finish, they create a temporary vacancy or even make a core idle. With `Map+`, one can dynamically optimize core status. As shown in Figure 14(a), during each control period, we identify the state-changed functions and move them to the appropriate cores. Our goal is to improve the OOP rate, which is defined as the percentage of time a processor core is running under OOP over a certain period. After migration, one may need to adjust the core frequency based on the Attribute-Frequency Table.

`Tune+`: With `Tune+`, we intend to make better performance-power trade-offs (i.e., Eco Mode). If the invoked functions are few, there will be plenty of space on the server.

TABLE I
SUMMARY OF FIRST'S OPERATOR EXTENSION

Type	Operator	Description	Parameters/Hints
Front End	Prepare	Profile of languages&script	*-TL: Tidal-Lane mode*
		Profile of func. lifecycle	
		Profile of arch. attribute	
	Gather	Gather func. based on OOP	*-FL: Fast-Lane mode*
Back End	Map	Map func. to cores	*-Dyn: Dyn mode*
	Tune	Adjust the freq. of cores	
		Dynamically migrate func.	*-Eco: Eco mode*
		Adjust freq. of mixed cores	

In this case, consolidating functions and putting idle cores to sleep is desirable. The `Map+` only mildly moves function around to minimize the number of *Mixed Core*. Differently, `Tune+` reduces the idle power consumption by creating more *Mixed Core* through function consolidation, as shown in Figure 14(b). We will discuss this in the evaluation.

4) Core State Analysis: With enhanced `Map`, one may achieve lower CCF, as Figure 15 shows. Due to function motion, functions will be re-mapped to the appropriate cores with similarities during runtime. This process lowers the core chaos factor. In Figure 15(a), `Map+` will disperse function to idle cores if necessary, which contributes to OOP convergence. Differently, `Tune+` consolidates functions for maximizing idle cores. Thus, as shown in Figure 15(b), the COR will be monotonously decreased. However, to consolidate functions, the system has to process functions with more *Mixed Cores* which increases CCF monotonously.

TABLE II
EVALUATED FUNCTIONS

Function	Description	Runtime
markdown	Renders the markdown text to HTML	python
img-resize	Resize images to icons	nodejs
sentiment	Sentiment analysis of text	python
ocr-img	Find text in images using OCR	nodejs
autocomplete	Autocomplete the string from a corpus	nodejs
FileIO	IO-intensive function	go
ALU	CPU-intensive function	go/ruby/swift/php

TABLE III
THE EVALUATED FUNCTION POOLS

Function Pool	Abbr.	Functions
Homogeneous Pool $(FCF = 0)$	HO	Functions with $var(OOP)_{i\&e} = 0$
Less Homogeneous Pool $(0 < FCF < 1)$	HOI	Functions with $var(OOP)_i = 0, var(OOP)_e > 0$
	HOE	Functions with $var(OOP)_i > 0, var(OOP)_e = 0$
Heterogeneous Pool $(FCF = 1)$	HE	All functions combined $var(OOP)_i > 0, var(OOP)_e > 0$

D. Summary of Interface Extension

In general, FIRST is not a cloud-scale abstraction layer; instead, it focuses on function re-distribution (i.e., meta-scheduling) at the server level. FIRST uses four kernel operators to aid the efficient management of serverless functions on each server node. These operators, if used and combined synthetically, can achieve different resource management goals.

To support a graceful transition to IRS-based function management, FIRST provides an independent interface for each operator, as shown in Table I. In this case, each operator can be used and extended separately. In keeping with common practice, FIRST does not require significant modifications to the underlying OS or hardware. With different parameters (enhancement), these versatile operators allow one to manipulate function assignments in different ways. we expect that each operator can be further customized for different purposes.

IV. Experimental Methodologies

We use trace-driven evaluation to thoroughly analyze the very large design space of FIRST. We collect detailed function execution data from real machines (a 20-core server, Intel Xeon Silver 4114) with Ubuntu 16.04.5 LTS installed. The processor supports per-core DVFS with frequencies from 0.8GHz to 2.2GHz at the interval of 0.1GHz. The default driver is ACPI with the ”ondemand” governor. We record dynamic power consumption using *turbostat*. We use Openwhisk [8] as our serverless platform. The workload configuration (e.g., input size) is the same as prior work [29].

The workflow of our trace-driven evaluation is shown in Figure 16. We set up experiments with various serverless functions [28] shown in Table II. Specifically, we evaluate our design using different function pools as shown in Table III. *HO* includes homogeneous functions whose multi-dimensional attributes are all the same. Both *HOI* and *HOE* contains less homogeneous functions which have limited similarity. Functions in *HOI* all use the same languages (OOP convergence in the initialization phase) and functions in *HOE* have the same resource type (OOP convergence in the execution phase). Lastly, *HE* just mixes all functions in the Table II.

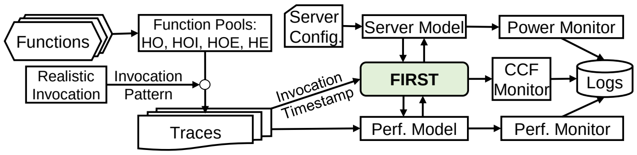


Fig. 16. The workflow of trace-driven evaluation

TABLE IV
INVOCATION PATTERNS ADOPTED FROM REAL CLUSTERS.

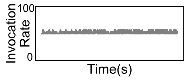
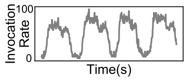
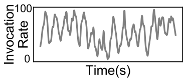
		
(a) Fairly Flat [2])	(b) Periodical [2])	(c) Fluctuating [29])

TABLE V
THE EVALUATED SCHEMES

Mechanism	Description
PerfFst	Performance-first scheduling scheme for ideal performance
Per-APP	Fine-grained control considering function as a whole application
IRS-TL	FIRST in Tidal-Lane mode (enhanced `Prepare+`)
IRS-FL	FIRST in Fast-Lane mode (enhanced `Gather+`)
IRS-Dyn	FIRST with enhanced front-end operators and `Map+`

We invoke each request set using three invocation patterns as shown in Table IV. The average invocation rate is 50 functions/second. Especially, the fluctuating invocation pattern follows the distribution-based invocation pattern of the Azure serverless trace [29]. The generated traces includes the IDs, invocation timestamps, and best-suited frequency of each function. We feed the generated traces into our simulation framework which uses a realistic server power model and function execution data.

Table V summarizes our evaluated power management schemes. We consider two important baselines which present the state-of-the-art function management and power management: *PerfFst* and *Per-APP*. *PerfFst* represents the current performance-first scheduling scheme which invokes the function as soon as it arrives with the highest frequency and colocates functions without resource contention [8]. It has the shortest latency but may lead to OOP divergence. *Per-APP* represents the state-of-the-art application-level, fine-grained power management [10], [11], [36]. Rather than compare IRS with a naïve Per-App mechanism, we enhanced Per-App by supporting OOP (but did not further distinguish between initialization and execution phases). It can dynamically move functions and select the desired frequency like *IRS-Dyn*.

V. Evaluation Results

A. Effectiveness of FIRST

We mainly use the energy-delay product (EDP), which offers equal weight to both energy and performance, to evaluate our design. Figure 17 shows the average EDP using various function pools and invocation patterns. All results are normalized to *PerfFst*. On average, *IRS-TL*, *IRS-FL* and *IRS-Dyn* show 18%, 14% and 24% lower EDP compared with

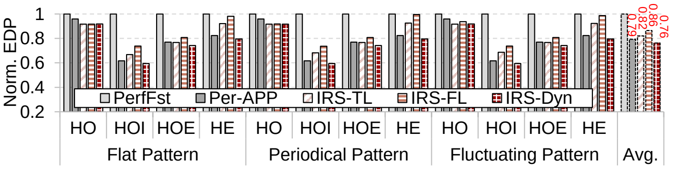


Fig. 17. A comparison of the EDP of different schemes

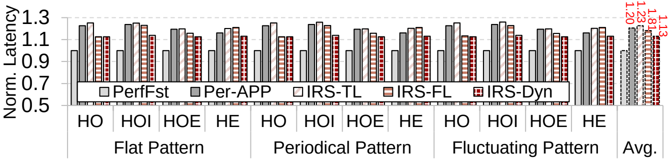


Fig. 18. A comparison of the latency of different schemes

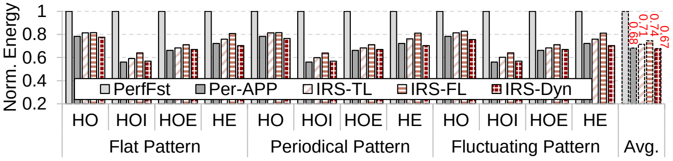


Fig. 19. A comparison of the energy of different schemes

PerfFst, respectively. *Per-APP* is better than *IRS-TL* and *IRS-FL* since it can dynamically select the most suitable frequency. Overall, *IRS-Dyn* outperforms all the other schemes. The EDP of *IRS-Dyn* is better than *IRS-FL* since *IRS-FL* pursues high performance for some functions while disrupting OOP.

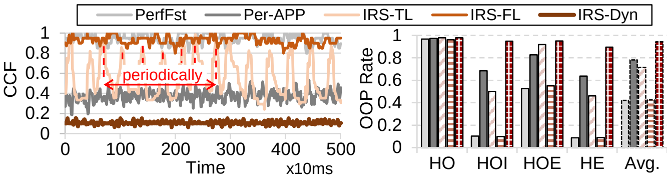
In addition to EDP, we also evaluate the latency and energy as shown in Figure 18 and 19. All results are normalized to *PerfFst*, which aims at high performance. We observe that the latency of other schemes are all larger than *PerfFst*. *IRS-Dyn* achieves the best performance compared to *Per-APP*, *IRS-TL* and *IRS-FL*. Compared with *PerfFst*, *IRS-Dyn* only has 13% performance loss while achieving 33% energy savings on average. *IRS-Dyn* outperforms *Per-APP* by 6%, with 3% less energy. This could lead to attractive data center ROI (return on investment) improvement. Notably, IRS would achieve better performance with more diversified functions in practice.

B. Overall CCF and OOP Rate

Figures 20(a) and 20(b) show our results with heterogeneous request set under invocation pattern C. As can be seen in Figure 20(a), *IRS-Dyn* can maintain a near-zero CCF due to its orderly function migration. In contrast, *IRS-FL* schedules function on-demand and it cannot orderly deploy functions. Its CCF is similar to *PerfFst* whose CCF fluctuates significantly. Since *IRS-TL* invokes functions and assigns them in bulk, its CCF fluctuates periodically. The results meet our expectations as discussed in Section III-C. In general, the OOP rate of *IRS-Dyn* is about 94%, as shown in 20(b). It implies that functions can be processed with OOP convergence and it is the reason that *IRS-Dyn* can achieve the best efficiency than others.

C. A Glance at the Frequency Distribution

It is critical to ensure that a function adopts its OOP. We present the statistics of frequency distribution under different schemes in Figure 21. The rightmost bar (*Oracle*) shows the optimal operating frequency of each workload. As we can see, *PerfFst* uses fixed frequency since they are oblivious to function attributes. The frequency distribution of *IRS-Dyn* is most similar to *Oracle* in all cases as it manages functions using FIRST to avoid OOP divergence.



(a) CCF

(b) OOP Rate

Fig. 20. The CCF and OOP rate in the worst case

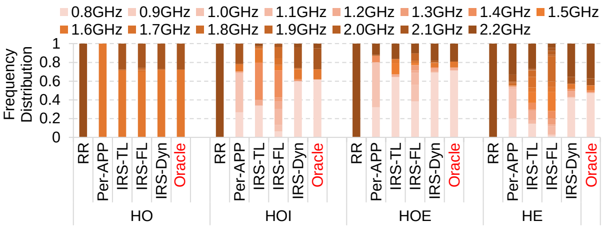


Fig. 21. The resulted core frequency distribution

D. Importance of Fine-Tuning Mixed Cores

Improperly setting the frequency of the *Mixed Cores* may lead to degraded performance. To understand this, we simply set the frequency of *Mixed Cores* with the min/max/avg value of functions' best-suited frequency. We compare the results with the most efficient frequency settings calculated using our heuristic algorithm (detailed in Figure 10).

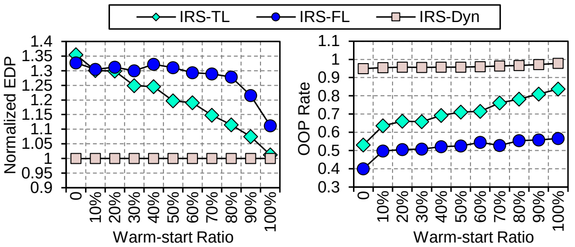
Table VI shows the EDP of each mode under different frequency tuning policies. It shows that Eff provides the best efficiency for all the evaluated modes. Our customized frequency tuning method Eff. for *IRS-TL*, *IRS-FL* and *IRS-Dyn* can manifest 25%, 42%,8% EDP reduction compared with Min, respectively. The EDP reduction of *IRS-Dyn* is far less than *IRS-TL* and *IRS-FL* since *IRS-Dyn* maintains high OOP rate and there are much fewer mixed cores.

E. Impact of Warm-start on FIRST

Functions with a warm start can skip the initialization phase which is common in serverless computing platforms. In Figure 22, we investigate the optimization effectiveness of different power management schemes under various warm start ratios (0

TABLE VI
COMPARISON OF DIFFERENT FREQUENCY TUNING POLICIES

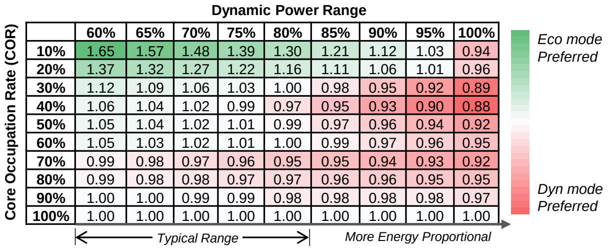
	IRS-TL	**IRS-FL**	**IRS-Dyn**
Min. freq.	1	1	1
Max. freq.	0.76	0.6	0.93
Avg. freq.	0.86	0.77	0.99
Eff. allocation	**0.75**	**0.58**	**0.92**



(a) Normalized EDP

(b) OOP Rate

Fig. 22. Impact of different warm-start ratios


Dynamic Power Range
Core Occupation Rate (COR)
Eco mode Preferred
Dyn mode Preferred
Typical Range
More Energy Proportional

	60%	65%	70%	75%	80%	85%	90%	95%	100%
10%	1.65	1.57	1.48	1.39	1.30	1.21	1.12	1.03	0.94
20%	1.37	1.32	1.27	1.22	1.16	1.11	1.06	1.01	0.96
30%	1.12	1.09	1.06	1.03	1.00	0.98	0.95	0.92	0.89
40%	1.06	1.04	1.02	0.99	0.97	0.95	0.93	0.90	0.88
50%	1.05	1.04	1.02	1.01	0.99	0.97	0.96	0.94	0.92
60%	1.05	1.03	1.02	1.01	1.00	0.99	0.97	0.96	0.95
70%	0.99	0.98	0.97	0.96	0.95	0.95	0.94	0.93	0.92
80%	0.99	0.98	0.98	0.97	0.97	0.96	0.96	0.95	0.95
90%	1.00	1.00	0.99	0.99	0.98	0.98	0.98	0.98	0.97
100%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Fig. 23. Normalized efficiency of the Eco mode

means all functions experience cold starts and 100% means all functions experience warm starts). We use the most heterogeneous workload and highly fluctuating invocation pattern. In the figure, all the results are normalized to *IRS-Dyn*. More pre-warmed functions indicate that the variety of functions reduces. As the warm-start ratio increases, *IRS-TL* and *IRS-FL* both become more efficient. The differences between *IRS-Dyn* and other schemes are narrowed.

F. The Necessity of the Eco Mode

FIRST's Eco mode allows one to save more energy by reducing the overall OOP rate. In Figure 23 we evaluate two key parameters (COR and dynamic power range) that may affect the effectiveness of the Eco Mode. All the results are normalized to *IRS-Dyn*. As can be seen, Tune+ shows better efficiency with lower core occupation rate and smaller dynamic power range (top left corner). For a typical dynamic power range (i.e., 60% - 80%), Eco appears to be more competitive. *IRS-Dyn* is desirable when the COR ranges between 30% and 70%. In extreme cases, *IRS-Dyn* can outperform Eco by 12% if the server is fully energy-proportional.

G. Performance comparison under power budget

Ensuring OOP convergence of functions can help to improve performance of serverless functions under power budget. In Figure 24, we estimate the average latency of functions in HE pool under the fluctuating invocation pattern. The maximum power is 72W per node which allows all cores to process function at the highest frequency (2.2GHz). Under different power budgets, *IRS-Dyn* achieves better performance compared with *PerfFst* and *Per-APP*.

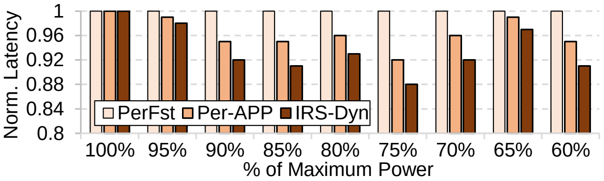


Fig. 24. Average latency under different power budget

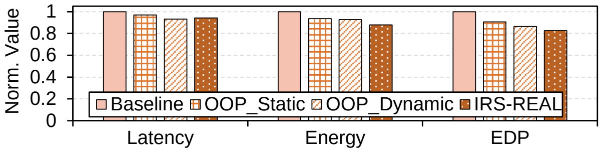


Fig. 25. Comparison of measured performance

VI. Discussion

Finally, we evaluate FIRST on real machines that have non-ideal control overhead. We also discuss how power-related architectural features that affect the efficacy of FIRST.

A. System Prototype Analysis

We test FIRST on our server. We use the latest Linux power management and function scheduling mechanism as the baseline. We further consider three schemes: *OOP_Static* - smartly changing core frequency based on OOP; *OOP_Dynamic* - consolidating functions based on similarities, leaving frequency tuning to the OS; and *IRS-REAL* - jointly consolidating functions and selecting the optimal frequency.

We measure the latency, energy and EDP of our system prototype as shown in Figure 25. The results of *IRS-REAL* in all the cases are better than *Baseline*. The latency of *IRS-REAL* is higher than *OOP_Dynamic* since *IRS-REAL* needs to adjust the frequency for the entire system. In terms of energy saving, *IRS-REAL* can save 12% energy and reduce 6% latency, compared to *Baseline*. Besides, *IRS-REAL* can reduce 17% EDP compared to *Baseline*, 8% EDP compared to *OOP_Static*, and 3% EDP compared to *OOP_Dynamic*.

We also measure and present the key overhead introduced by FIRST's new abstraction as shown in Table VII. Before functions are invoked, the operator `Prepare` needs about ∼5ms to obtain the attributes of functions. Afterwards, `Prepare` can process all functions in parallel. In addition, FIRST needs to move functions among cores and change the core frequency. We observe that the latency of function migration and frequency change are about ∼ 15ms (user-space adjustment) and ∼ 12ms on our system. Ideally, the delay of frequency change ranges between ∼10ns [13]- 20μs [11] and the delay of thread motion is about ∼0.25μs [25].

TABLE VII
Overhead introduced by FIRST

Operation	Prepare	Frequency adjustment	Function migration
Software Cost	∼ 5ms	∼ 15ms	∼ 12ms
Hardware Cost	/	∼10ns [13] - 20μs [11]	∼ 0.25 μs [25]

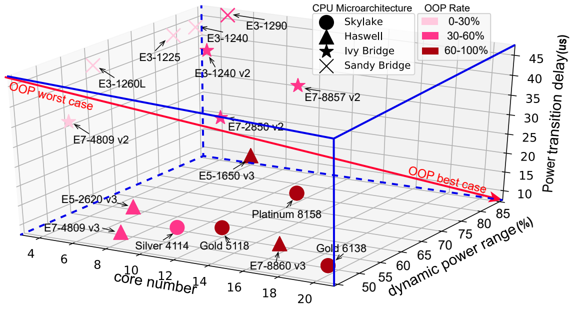


Fig. 26. Analysis of OOP convergence on different CPUs

TABLE VIII
THE IMPACT OF ENERGY PROPORTIONALITY METRICS ON CCF AND COR

Type	Metrics	Impact on CCF	Impact on COR
Spatial	Core Number	Large	Large
Temporal	Power Transition Delay	Large	Minimal
Magnitude	Dynamic Power Range	Large	Minimal

B. OOP-Friendly Architecture

When implementing FIRST, three factors in relation to server energy proportionality also need to be considered carefully. As shown in Table VIII, they affect the CCF and COR of evaluated servers in different ways.

First, the core number has a large impact on CCF and COR. With more cores, it becomes easier to reshuffle functions from the perspective of OOP convergence. However, the COR may decline with more cores due to core under-utilization.

Second, the power transition delay (i.e., DVFS latency) mainly affects the CCF of the system. With smaller power transition delays, we can quickly adjust the power levels for functions, thereby reducing the degree of disorder.

Third, given a larger dynamic power range of the processor, the server often has more power allocation choices. In this case, it is easier to identify the OOP for a group of functions and deploy them with OOP convergence.

In Figure 26 we estimate the likelihood of OOP convergence on modern CPUs of different generations. Some designs such as Skylake and Haswell are more OOP-friendly. We expect them to have better efficiency with FIRST, due to more CPU cores, smaller power transition delay, and larger dynamic range. Besides, more DVFS levels, agile thread management and advanced monitoring can also enhance FIRST.

VII. Related Work

Many prior works [4], [7], [9], [26], [29] aim to reduce the cold start latency. Since the stateless feature makes function interaction costly, researchers also propose to reduce the communication overhead [18], [19]. There are some works focusing on the scheduling of functions for performance [31], [33]. Besides, researchers have also developed frameworks and programming environment, aiming to construct serverless functions easily and efficiently. [30], [39]. All the prior works do not look at the power management side. Importantly, they also ignore the multi-dimensional attributes of functions when making scheduling decisions.

Many prior works [3], [5], [15], [22], [34], [36] aim to reduce power/energy consumption while guaranteeing the QoS. Some works [6], [15], [22] identify the latency slacks inside over-subscribed data centers. Another line of work mainly focuses on efficient power control [17], [35], such as fine-grain voltage boosting techniques [13]. Additionally, prior works such as PEAS [34] focuses on optimizing energy-proportional systems. These works incur OOP divergence in a complex serverless computing scenario, since they are oblivious to the attributes of co-located functions.

To the best of our knowledge, there are very limited work on serverless function power management [20], [27]. Power-aware system design for cloud-native applications is an open problem that requires more effort.

VIII. Conclusion

Emerging cloud-native design raises new challenges about managing server performance and power. We observe that serverless functions exhibit unique power behaviors that are overlooked by existing designs. To improve system efficiency in such a highly complex and dynamic environment, we propose FIRST, a well-crafted, multi-faceted system covering both workload analysis and cross-system control. Extensive evaluation shows that FIRST brings up to 24% energy efficiency. We expect that FIRST would enable balanced performance and efficiency in the cloud-native environment.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (No.62122053 and No.61972247), the Shanghai S&T Committee Rising-Star Program (No.21QA1404400), and a Tencent Research Grant. We thank all the anonymous reviewers for their valuable feedback. Corresponding author is Chao Li.

References

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[2] Alibaba, “Alibaba cluster data,” https://bit.ly/3iAus2R.

[3] A. Biswas, A. Majumdar, S. Das, and K. L. Baishnab, “Ocso-ca: opposition based competitive swarm optimizer in energy efficient iot clustering,” *Frontiers of Computer Science*, 2022.

[4] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “Seuss: skip redundant paths to make serverless fast,” in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[5] 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 *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[6] C.-H. Chou, L. N. Bhuyan, and D. Wong, “μdpm: Dynamic power management for the microsecond era,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[7] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, “Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[8] A. S. Foundation, “Apache openwhisk,” Available: https://openwhisk.apache.org/, accessed: 2019-11-12.

[9] A. Fuerst and P. Sharma, “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 (ASPLOS)*, 2021.

[10] A. Guliani and M. M. Swift, “Per-application power delivery,” in *Proceedings of Fourteenth European Conference on Computer Systems (EuroSys)*, 2019.

[11] X. Hou, C. Li, J. Liu, Y. Hu, and M. Guo, “Ant-man: Enabling agile power management in the microservice era,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[12] 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)*, 2021.

[13] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[14] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, “Cloud programming simplified: A berkeley view on serverless computing,” *arXiv preprint arXiv:1902.03383*, 2019.

[15] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, “Rubik: Fast analytical power management for latency-critical systems,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[16] 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)*, 2019.

[17] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, “System level analysis of fast, per-core dvfs using on-chip switching regulators,” in *2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[18] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, “Understanding ephemeral storage for serverless analytics,” in *2018 USENIX Annual Technical Conference (ATC 18)*, 2018.

[19] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 427–444.

[20] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, “The serverless computing survey: A technical primer for design architecture,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

[21] M. Liu, C. Li, and T. Li, “Understanding the impact of vcpu scheduling on dvfs-based power management in virtualized cloud environment,” in *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, 2014.

[22] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[23] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, “Stretch: Balancing qos and throughput for colocated server workloads on smt cores,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[24] S. McCracken, “How to architect for sustainability in a cloud native environment,” https://www.contino.io/insights/cloud-native-sustainability, 2022.

[25] K. K. Rangan, G.-Y. Wei, and D. Brooks, “Thread motion: fine-grained power management for multi-core systems,” in *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, 2009.

[26] R. B. Roy, T. Patel, and D. Tiwari, “Icebreaker: warming serverless functions better with heterogeneity,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[27] H. Shafiei, A. Khonsari, and P. Mousavi, “Serverless computing: a survey of opportunities, challenges, and applications,” *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[28] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural implications of function-as-a-service computing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[29] M. Shahrad, R. Fonseca, Í. 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 (ATC)*, 2020.

[30] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, “Serverless linear algebra,” in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[31] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, “Sequoia: Enabling quality-of-service in serverless computing,” in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[32] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *2018 USENIX Annual Technical Conference (ATC)*, 2018.

[33] X. Wang, C. Li, L. Zhang, X. Hou, Q. Chen, and M. Guo, “Exploring efficient microservice level parallelism,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[34] D. Wong, “Peak efficiency aware scheduling for highly energy proportional servers,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[35] G. Yan, Y. Li, Y. Han, X. Li, M. Guo, and X. Liang, “Agileregulator: A hybrid voltage regulator scheme redeeming dark silicon for power efficiency in a multicore architecture,” in *IEEE International Symposium on High-Performance Comp Architecture (HPCA)*, 2012.

[36] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, “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.

[37] 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*, 2021.

[38] L. Zhang, Y. Pu, C. Xu, D. Liu, Z. Lin, X. Hou, P. Yang, S. Yue, C. Li, and M. Guo, “Cloud-native server consolidation for energy-efficient faas deployment,” in *Annual IFIP International Conference on Network and Parallel Computing (NPC)*, 2022.

[39] W. Zhang, V. Fang, A. Panda, and S. Shenker, “Kappa: a programming framework for serverless computing,” in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[40] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, “Cpi2: Cpu performance isolation for shared compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.