SHANGHAI JIAO TONG UNIVERSITY

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

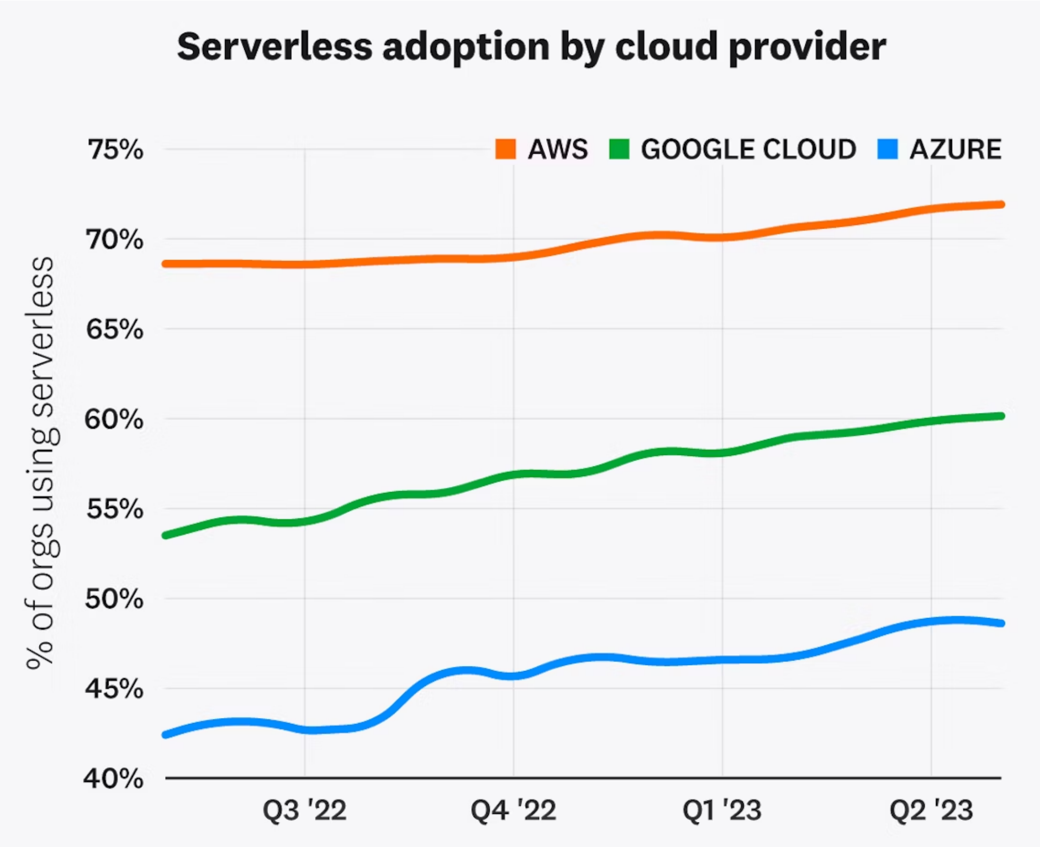
Du Liu, **Jing Wang**, Xinkai Wang, Chao Li*, Lu Zhang, Xiaofeng Hou, Xiaoxiang Shi and Minyi Guo

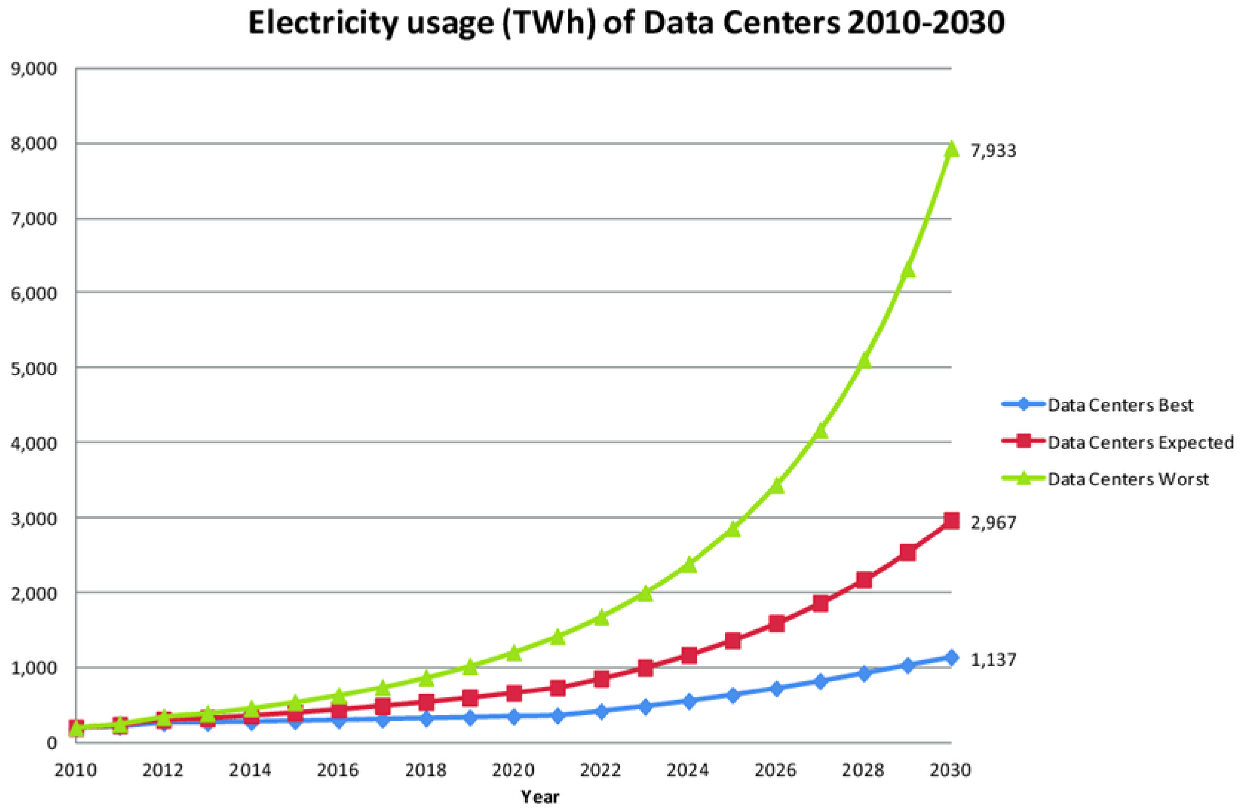
Shanghai Jiao Tong University
Department of Computer Science and Engineering

Outline

- Background
- Observations
- System Design
- Evaluation
- Conclusion

Background: Power of Serverless Computing





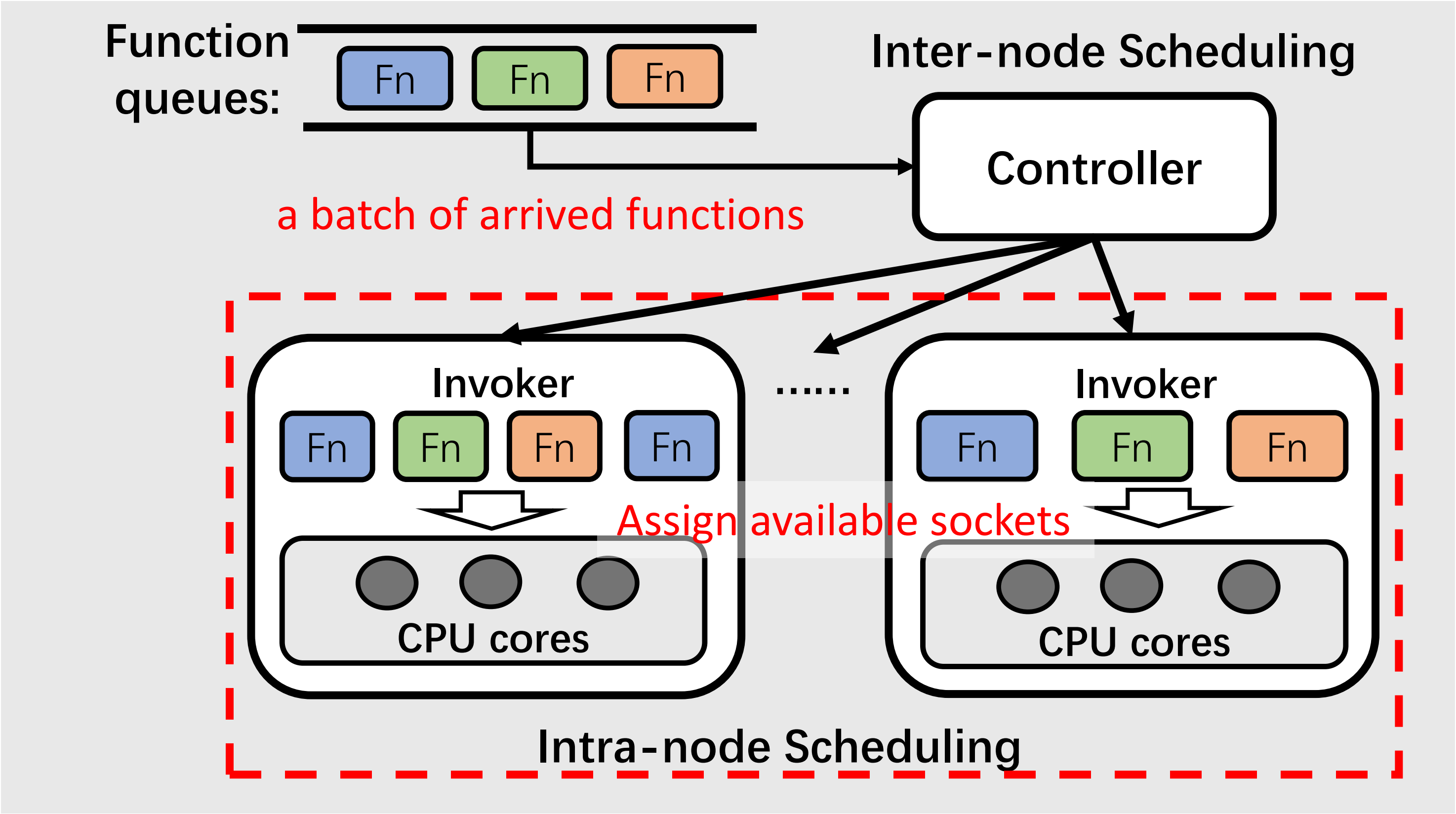
Serverless usage continues to **rise** across major clouds.

Power management of serverless functions is very important to the data center electricity cost.

[1]. https://www.datadoghq.com/state-of-serverless/
[2]. Global electricity demand of data centers 2010-2030.

Background: Serverless Scheduling

Both Inter-node and Intra-node serverless function scheduling optimization methods lack **per-function performance tuning**



Server-level scheduling:

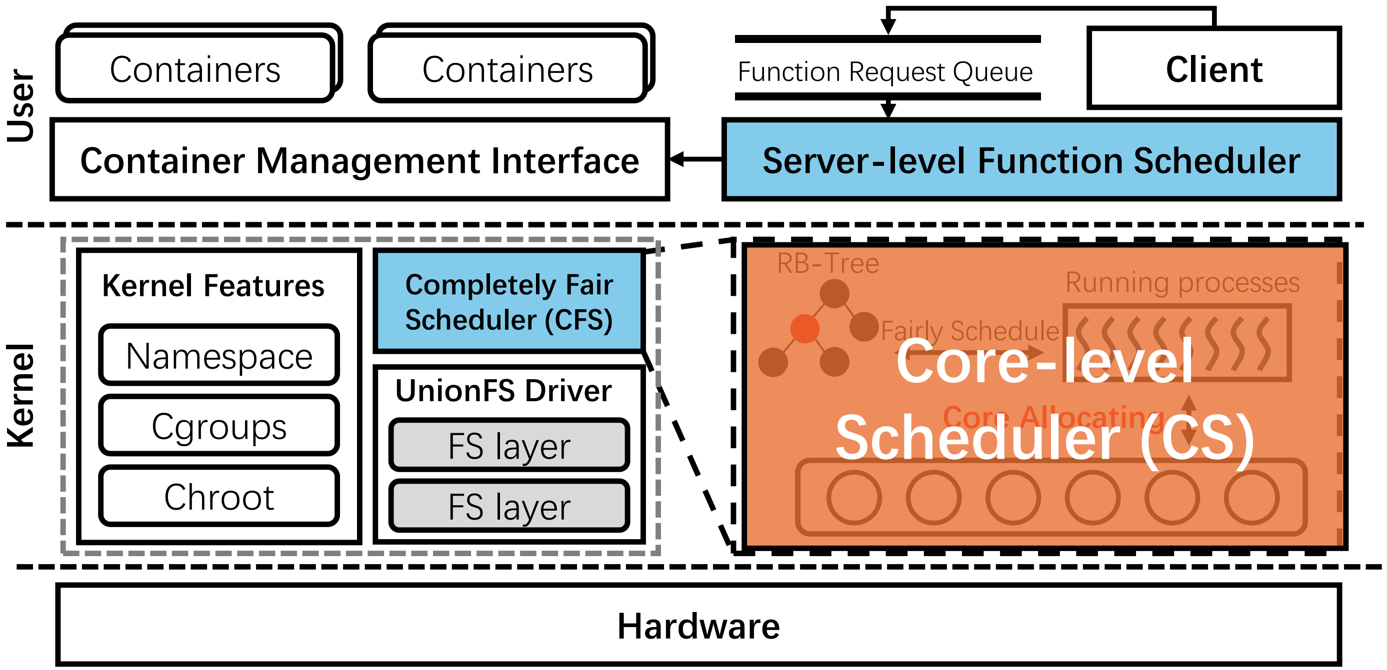
- Shortest Remaining Processing Time (SRPT)
- First-Come, First-Served (FCFS)

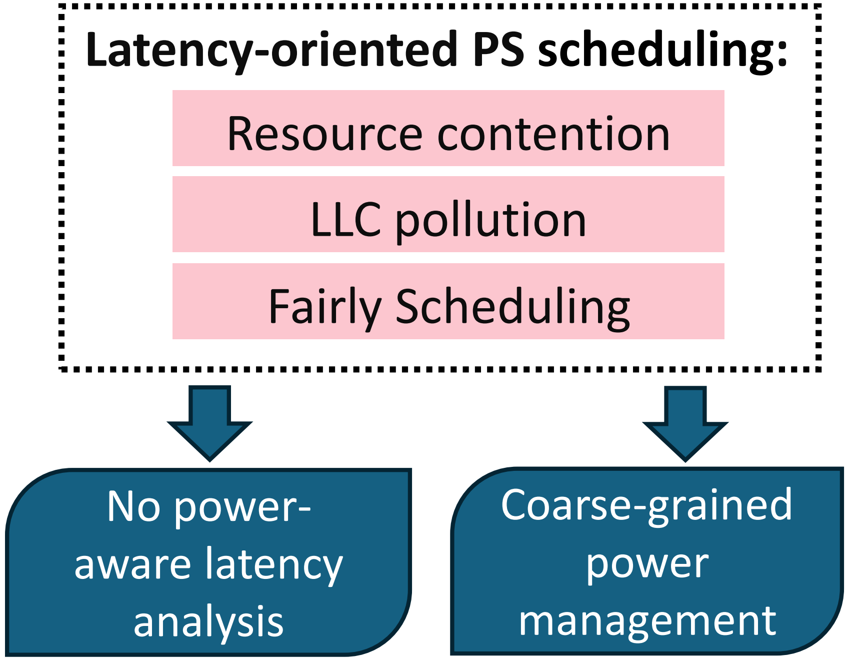
No per-function performance tuning

Server-level Serverless Scheduling → Core-level Serverless Scheduling

Background: Processor-sharing Scheduling Strategies

Existing serverless function scheduling: **Processor Sharing (PS)**





Designing **fine-grained power management methods** with processor (core) sharing is important.

[1] Kaffes, et al. "Hermod: principled and practical scheduling for serverless functions." Proceedings of the 13th Symposium on Cloud Computing. 2022.

Challenges

Challenge 1:
The complexity of co-located serverless functions makes it challenging to seize the core-level efficiency opportunities.

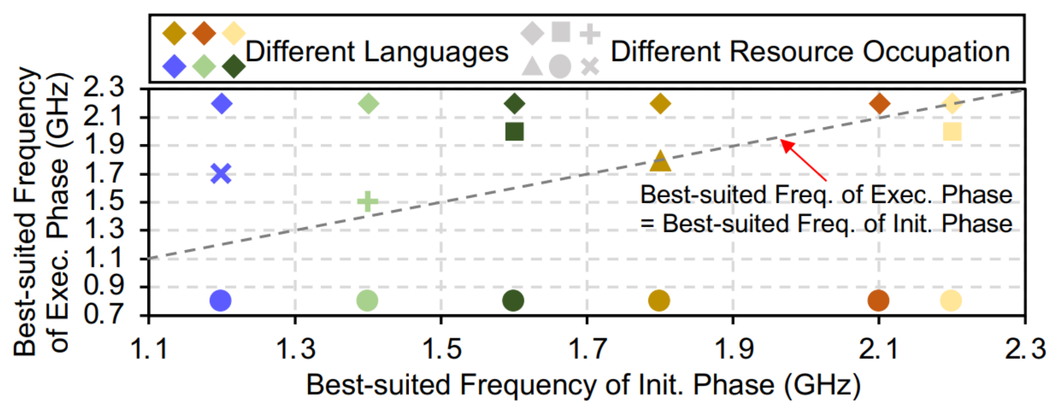
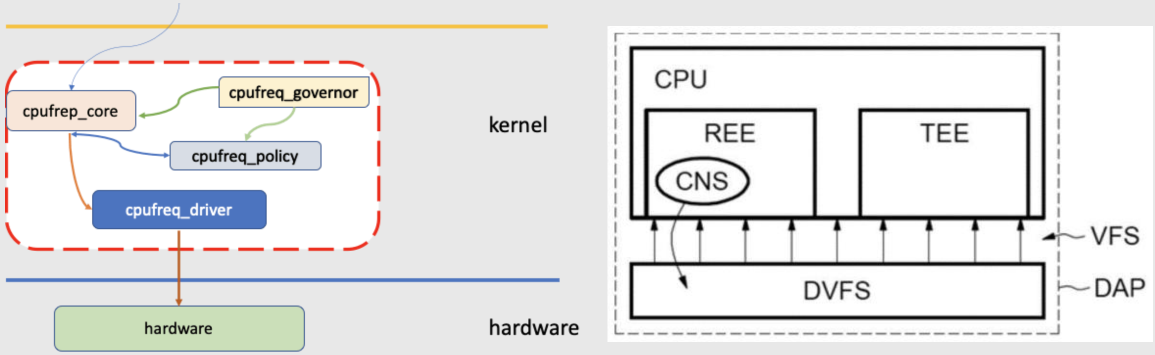


Fig. 1. Variety of functions' best-suited frequency

Indicate functions' power consumption simply by frequency is difficult.

Challenge 2:
Core-level power management cannot be directly implemented in current serverless computing platforms.



DVFS support socket-level frequency tuning but core-level power monitoring has no hardware support yet.

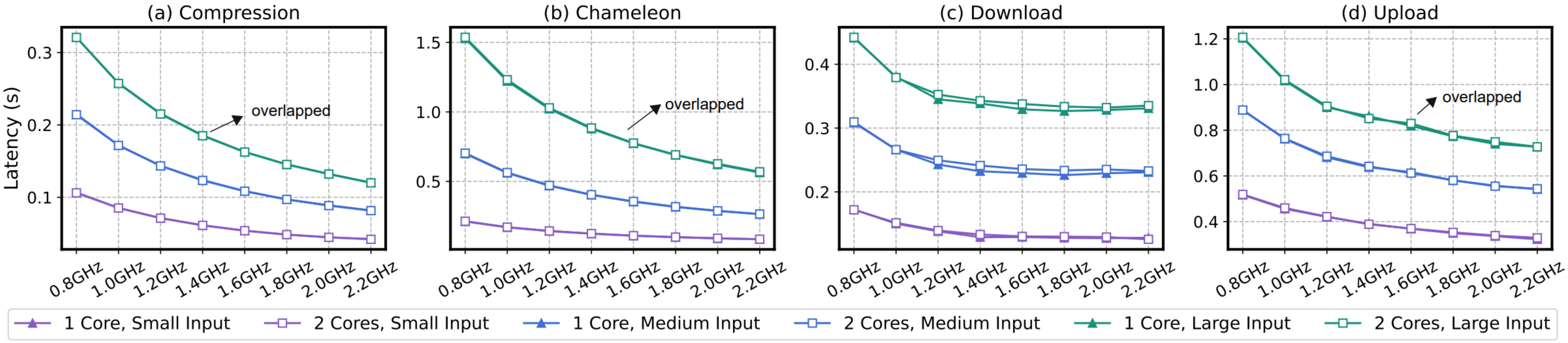
We seek to design a **core-level** serverless scheduling method to achieve **power saving** with **QoS** guaranteed.

Outline

- Background
- Observations
- System Design
- Evaluation
- Conclusion

Observations

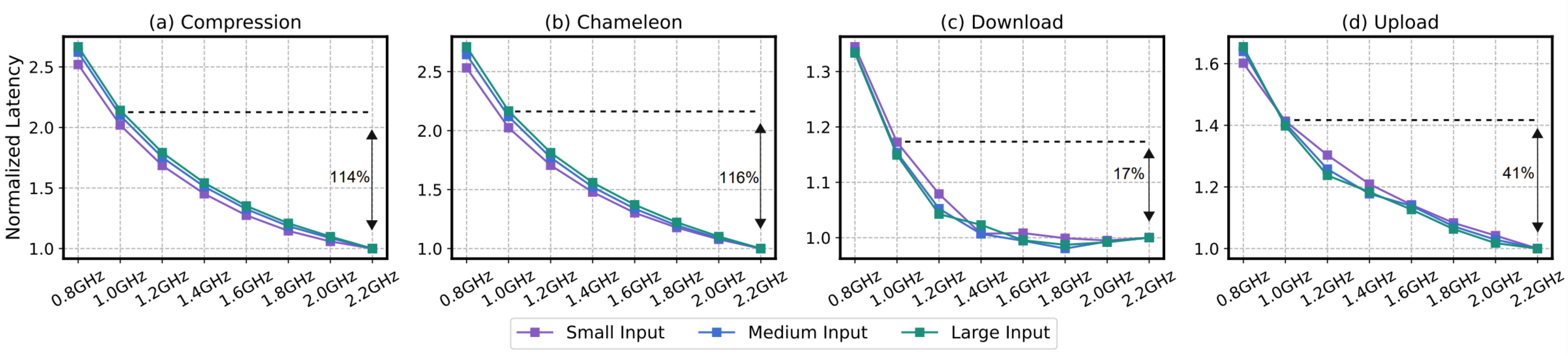
- **Limited Intra-function Parallelism:**
  - Functions hardly suffer performance loss when assigned to only a single CPU core.



One can assign each function on only one CPU core with high resource efficiency.

Observations

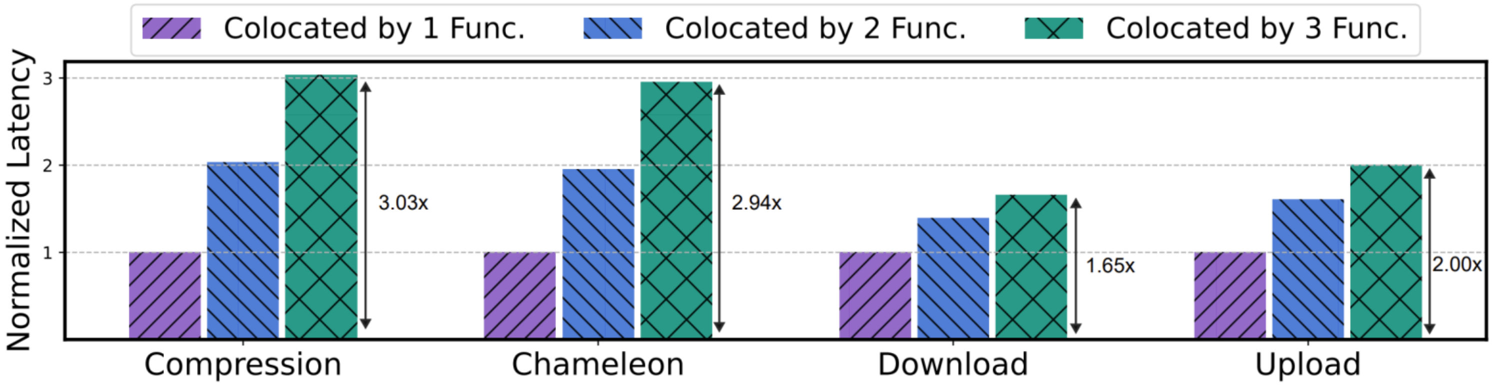
- **Limited Intra-function Parallelism:**
  - Functions hardly suffer performance loss when assigned to only a single CPU core.
- **Consistent Latency-Frequency Mappings Across Functions:**
  - Different functions have specific mappings of normalized latency and allocated frequency, which remain consistent across diverse inputs.



One can use lower CPU frequency if the performance reduction is acceptable.

Observations

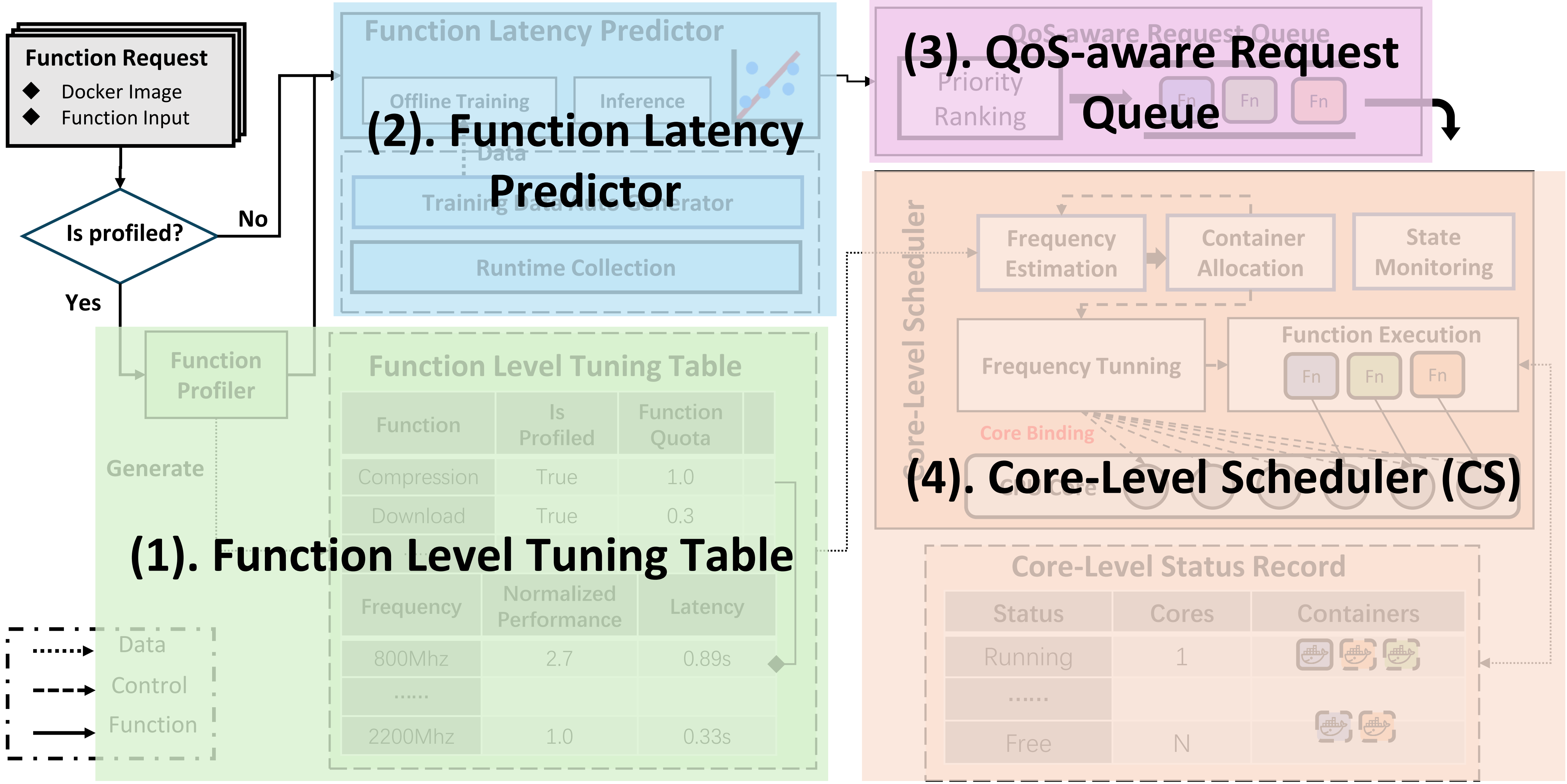
- **Limited Intra-function Parallelism:**
  - Functions hardly suffer performance loss when assigned to only a single CPU core.
- **Consistent Latency-Frequency Mappings Across Functions:**
  - Different functions have specific mappings of normalized latency and allocated frequency, which remain consistent across diverse inputs.
- **Improving CPU Utilization by Co-locating Non-CPU-intensive Functions:**
  - We can co-locate non-CPU-intensive functions to leverage available CPU resources and execute CPU-intensive functions independently by assigning unique CPU cores to maintain QoS.



Outline

- Background
- Observations
- **System Design**
- Evaluation
- Conclusion

System Design: Overview



System Design: Function level Tuning Table

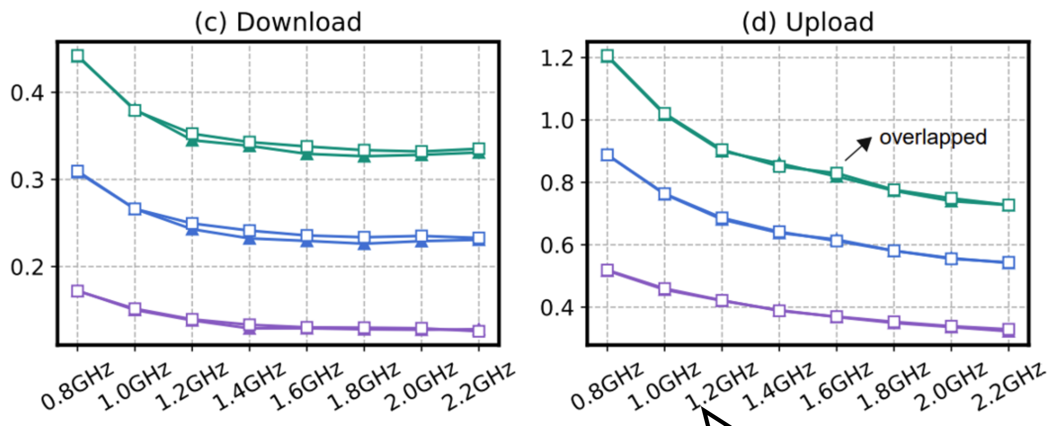
Function Profiler

Generate

Function Level Tuning Table

Function	Is Profiled	Function Quota
Compression	True	1.0
Download	True	0.3
......		

Frequency	Normalized Performance	Latency
800Mhz	2.7	0.89s
......		
2200Mhz	1.0	0.33s



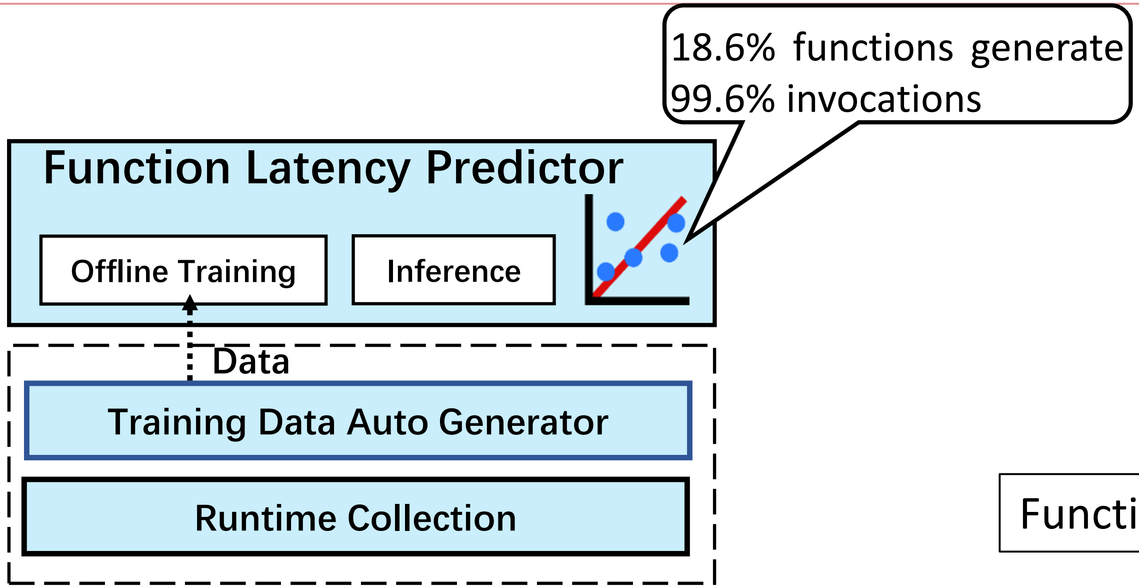
Quotas are calculated by the **CPU core utilization** in docker stat.

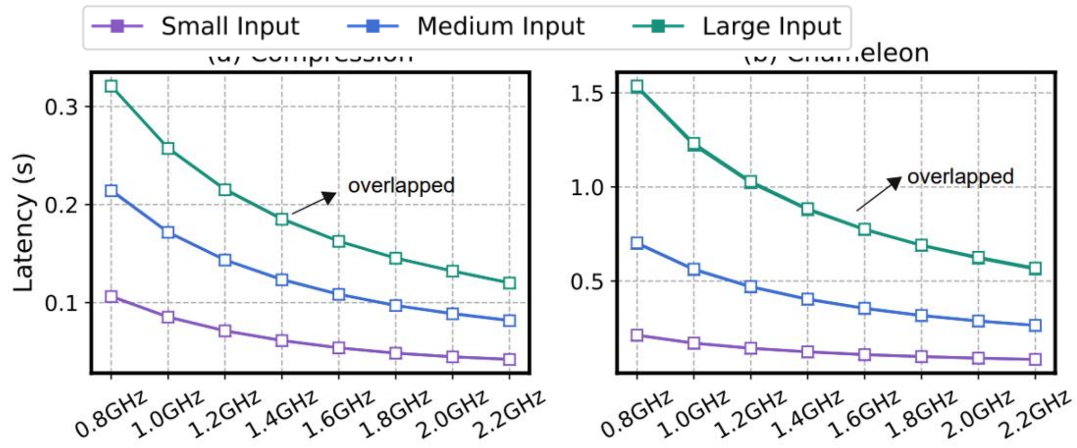
We obtain a **normalized performance curve** for each function at different CPU frequencies.

Key idea:

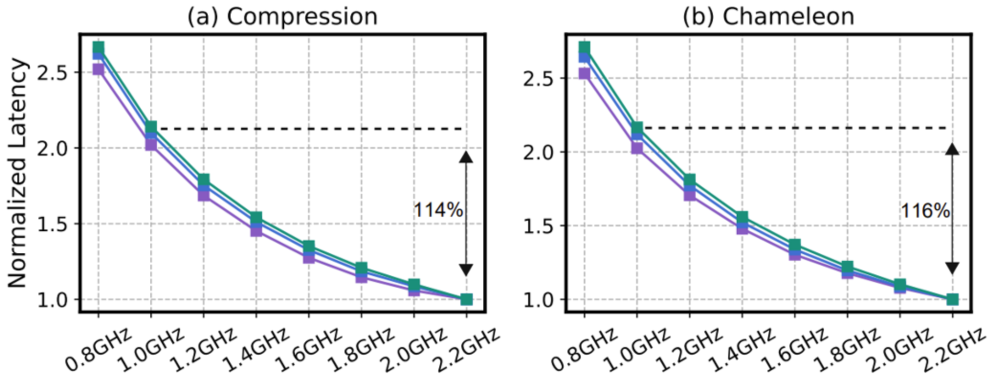
- Functions with **high** quotas can **monopolize** a core.
- Functions with **low** quotas will **co-locate** with other functions per core.

System Design: Function Latency Predictor





Functions on different **input size** has different end-to-end **latency**.

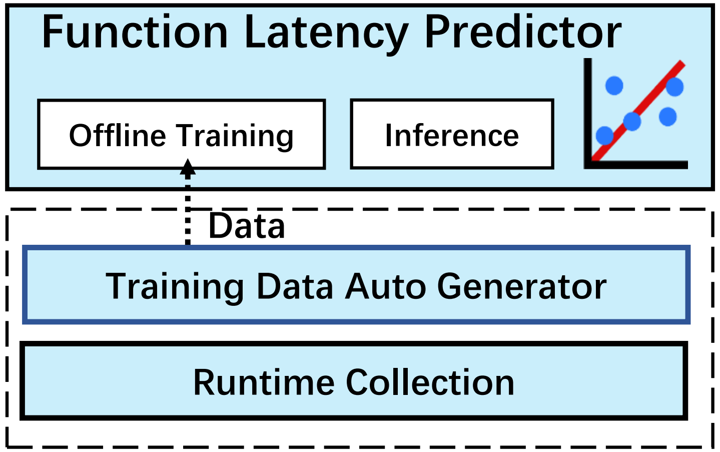


Functions on different core allocation keeps the **same latency trends**, even the input size changed.

Key idea:

- We can estimate function execution time according to a specified set of inputs based on **the performance latency trends**.
- Use **ML-based latency predictor** to accurately estimate the latency giving the CPU frequency.

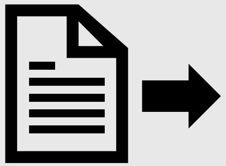
System Design: Function Latency Predictor



$$R2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2} \qquad \text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$$

- Different Types of Function Inputs:
  - **Numeric Input:** Represented by concrete numerical values.
  - **Composite Input:** A collection of multiple attributes.

Numeric Input:



File size

Composite Input:

- Pixel Height
- Pixel Width
- Image Format
- Image Size
- …

High accuracy of latency prediction

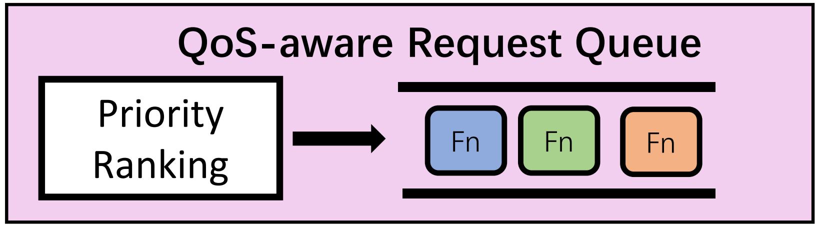
Function	Overhead		Accuracy	
	Training	Inference	R^2	RMSE
Upload	2.0 ms	0.19 ms	0.988	0.028
Download	2.0 ms	0.19 ms	0.988	0.023
Chameleon	3.8 ms	0.19 ms	0.894	0.111
BFS	2.2 ms	0.18 ms	0.998	0.007
Compression	2.9 ms	0.18 ms	0.999	0.001
Dynamic HTML	1.2 ms	[illegible]	0.999	0.012
Linpack	2.1 ms	[illegible]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

Lower training and inference latency

Key design:

- ➢ We adopt linear regression as the prediction model for latency and utilize **R2 score** and **Root Mean Square Error (RMSE)** as metrics to evaluate the models.
- ➢ We train specific models for different types of serverless functions.

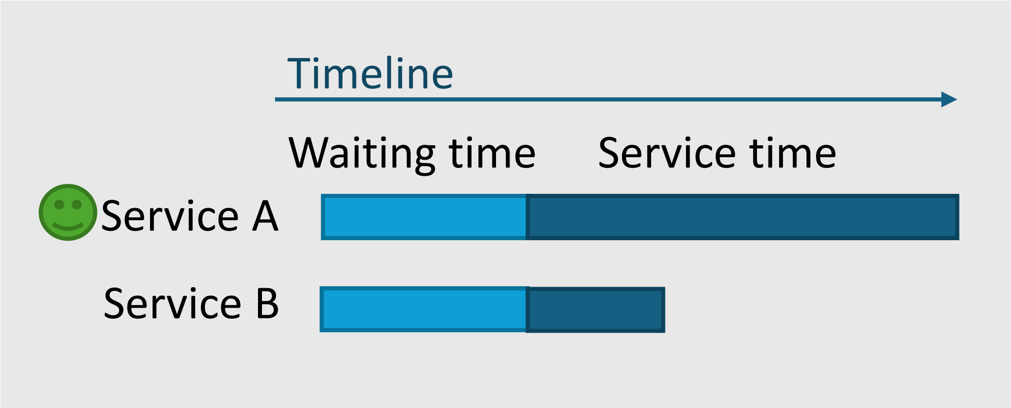
System Design: QoS-aware Request Queue

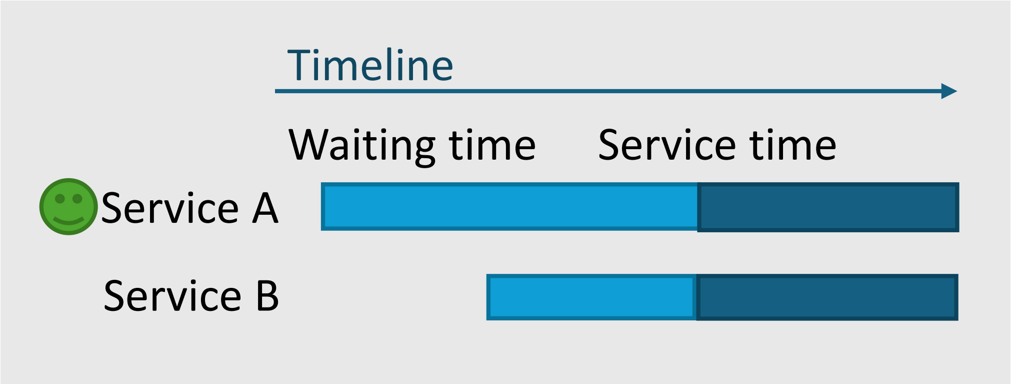


Response Ratio (RR) based priority ranking:

$$RR = \frac{\text{Waiting Time} + \text{Service Time}}{\text{Service Time}}$$

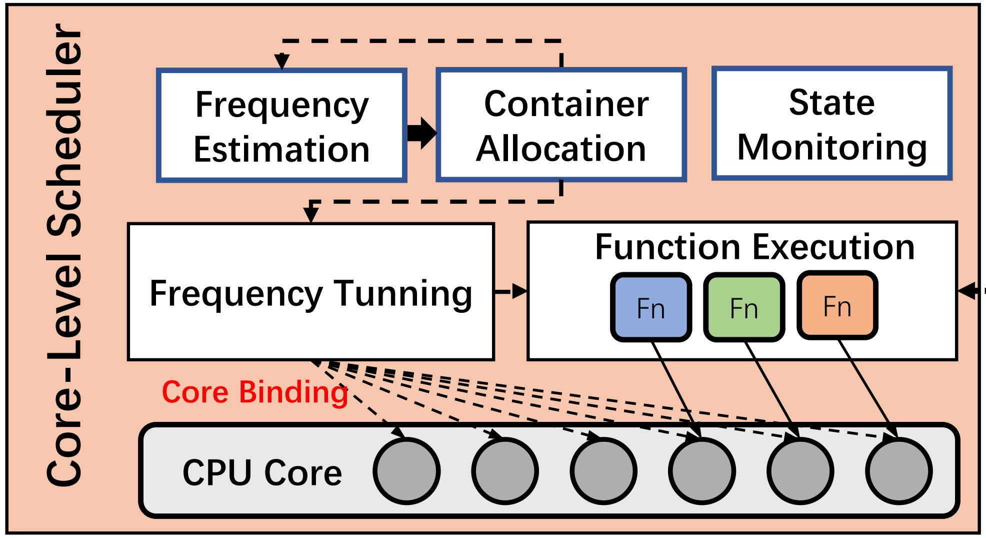
Request with a higher response ratio (RR) will be prioritized.

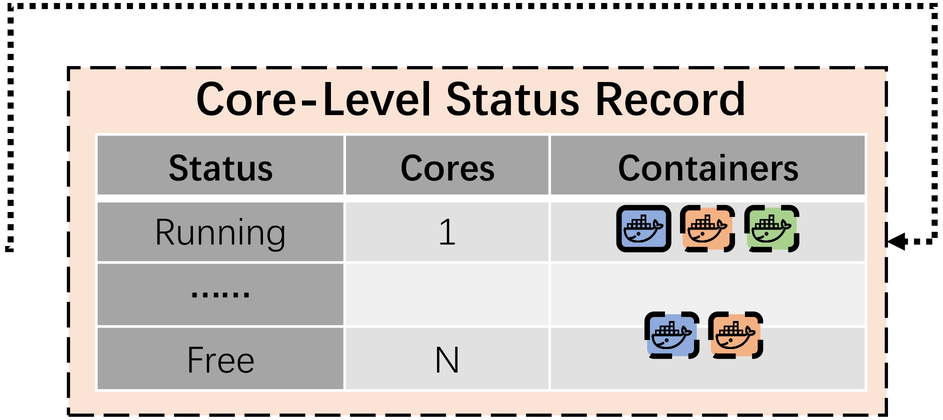


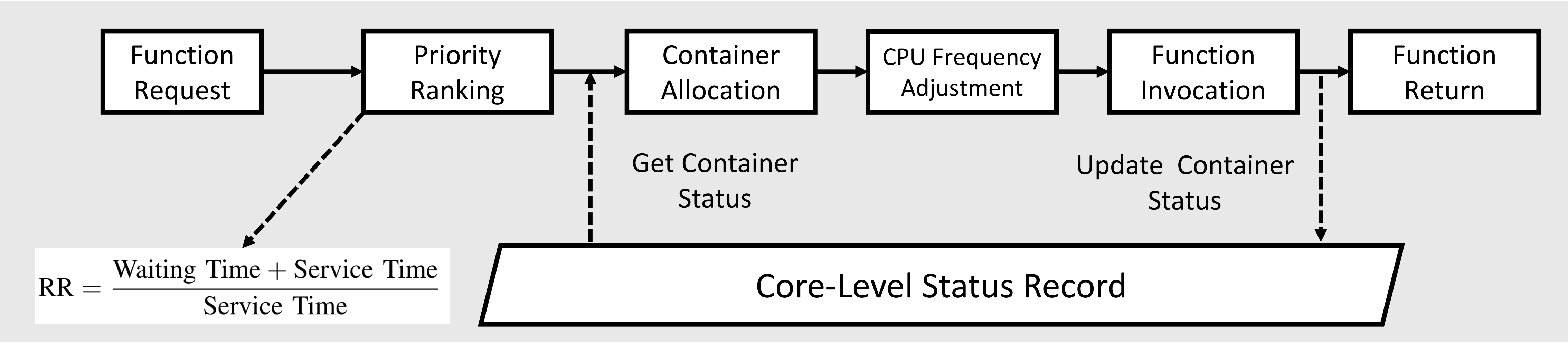


Our design combines the advantage of Shortest Job First and First Come First Serve algorithms to fairly handle both short tasks and long tasks.

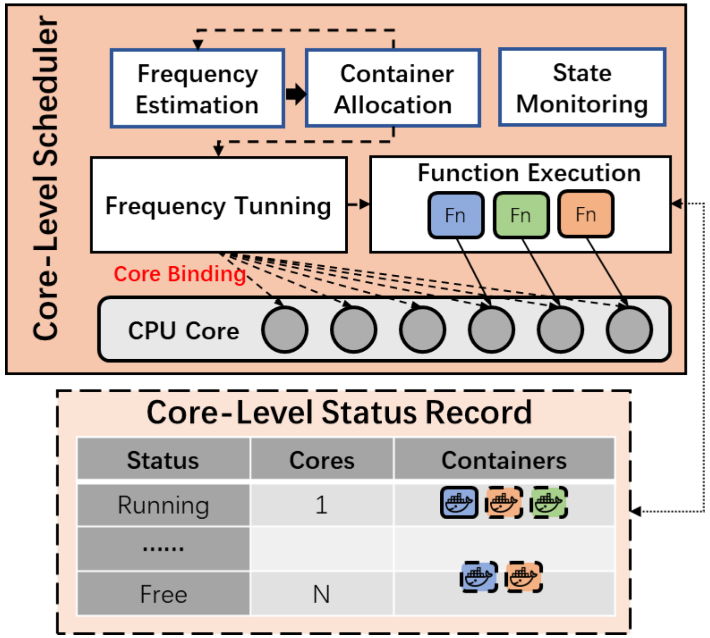
System Design: Core-Level Scheduler

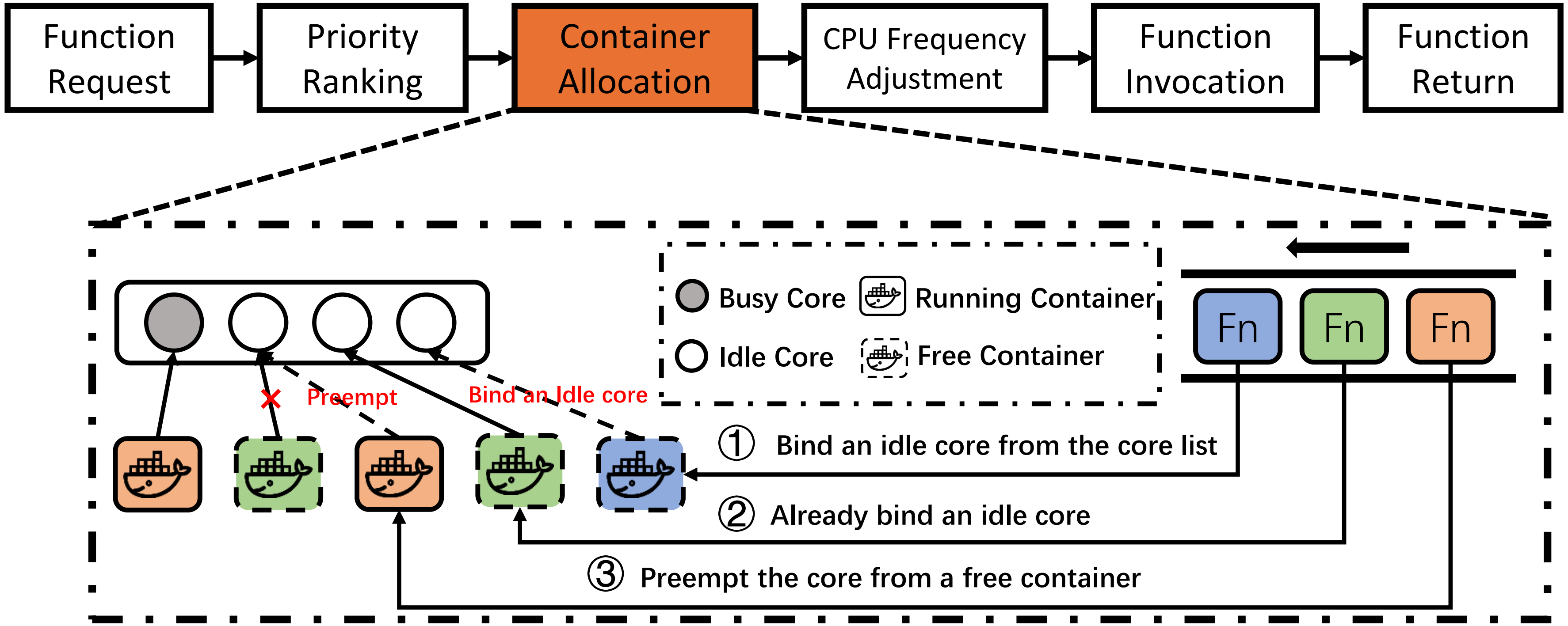






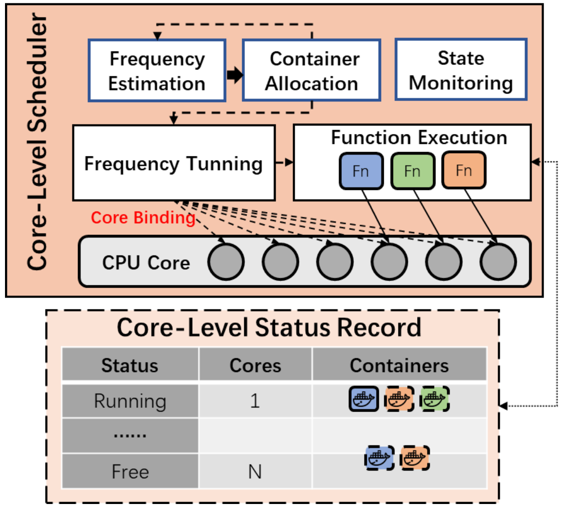
System Design: Core-Level Scheduler

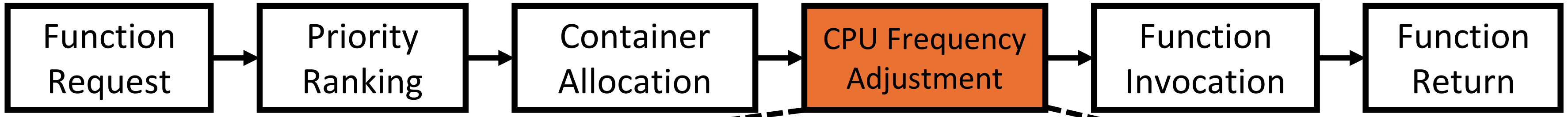




- **Core binding:**
  - Bind functions with **free CPU cores** in a container
  - Assign **other CPU cores** to a full-use container and bind with the functions
  - **Preempt a core** from other functions and assign an idle container on this core

System Design: Core-Level Scheduler





Best Degradation Rate

$$\mathrm{BDR} = \frac{\mathrm{QoS}}{\mathrm{Service\ Time}} - \frac{\mathrm{Waiting\ Time}}{\mathrm{Service\ Time}}$$
$$= \mathrm{QoS\ Ratio} - \mathrm{RR} + 1$$

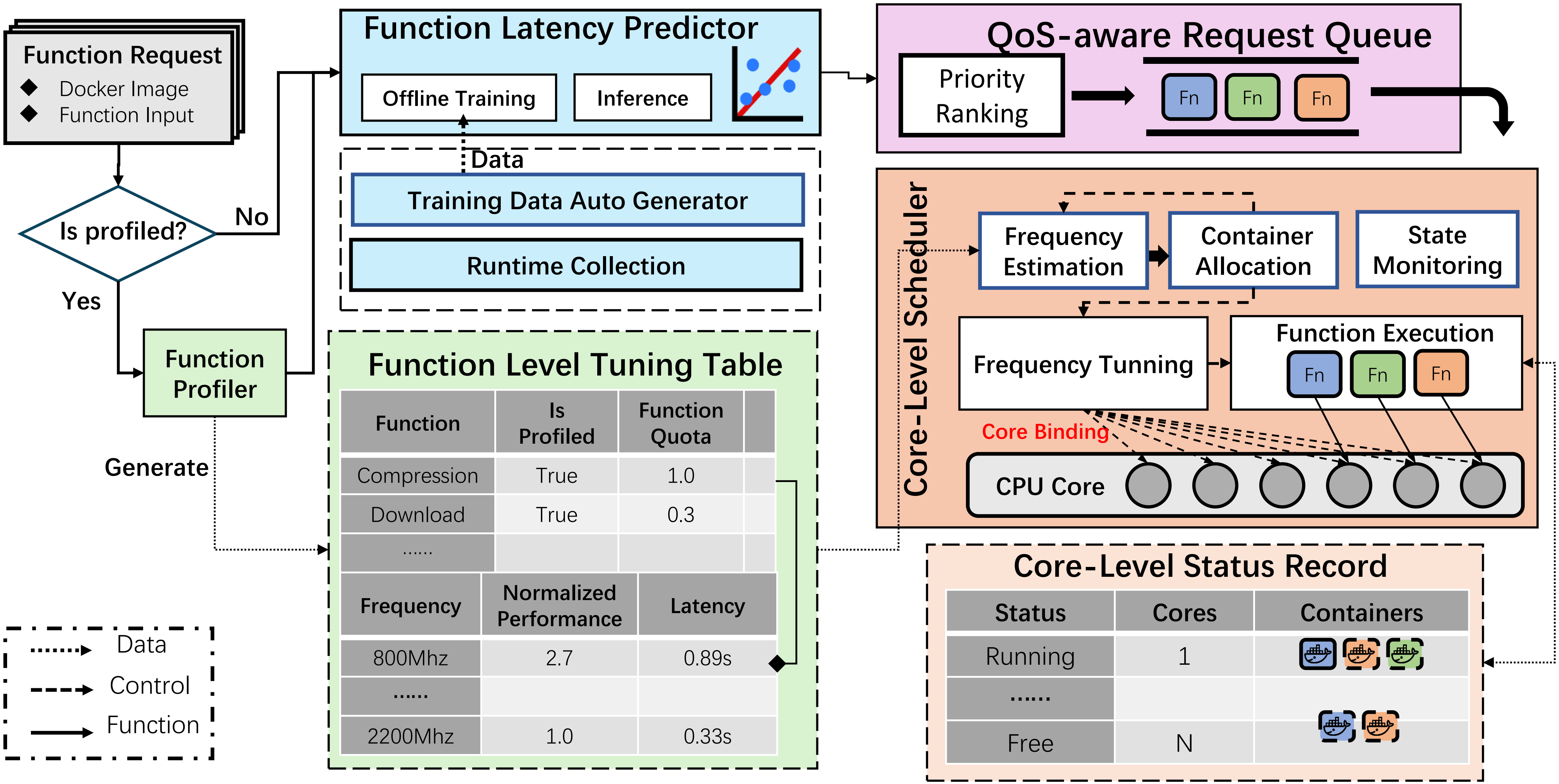
Search Normalized Latency under Different CPU Frequencies in FLTT

→ Adjusting CPU Core to Optimal Frequency

- **Core Frequency configuration:**
  - Find frequency with acceptable performance
  - Select the highest frequency among different functions

- **Server state:**
  - **Idle mode:** The system does not run processes in the highest frequency.
  - **Busy mode:** If there are too many waiting tasks in the queue, all cores work will switch to the highest CPU frequency.

System Design: Overview



Outline

- **Background**
- **Observations**
- **System Design**
- **Evaluation**
- **Conclusion**

Evaluation: Methodology

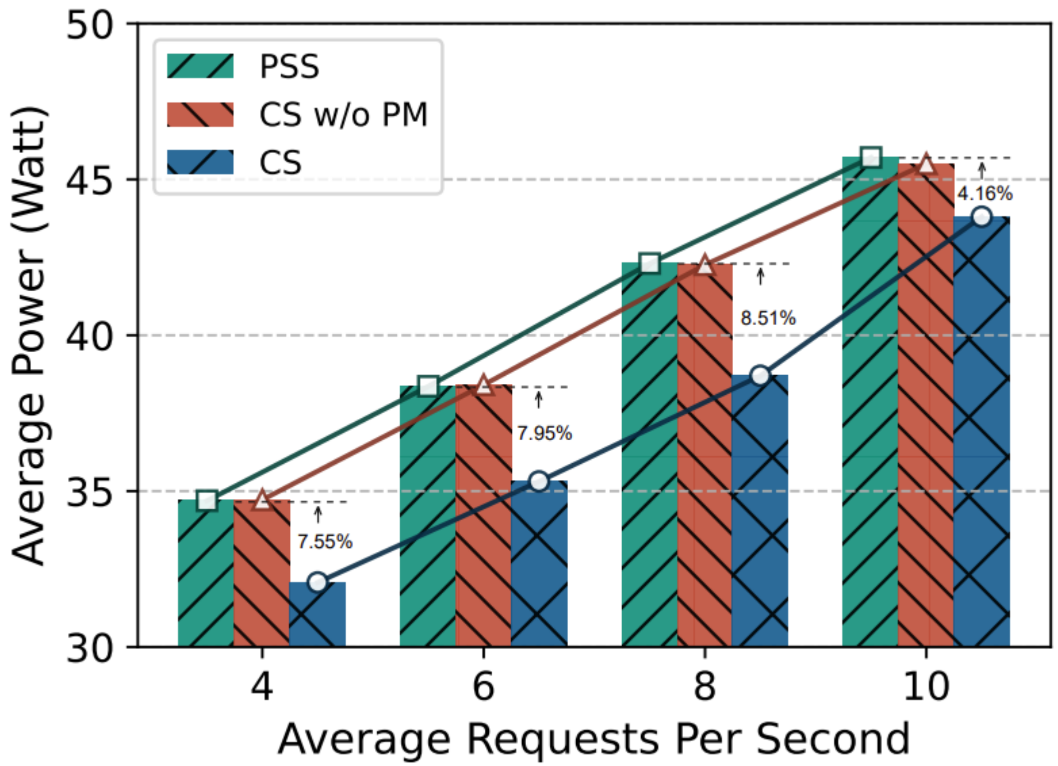
- **Evaluated Functions**
  - selected from FunctionBench [1] and SeBS [2]
- **Load Generator**
  - emulate the fluctuations of the coming requests
  - include peaks and valleys
- **Metric**
  - P95 function latency
- **Experiment Environment**
  - separated CPU sockets for function execution and the scheduling system
  - comparative experiments across three systems

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

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

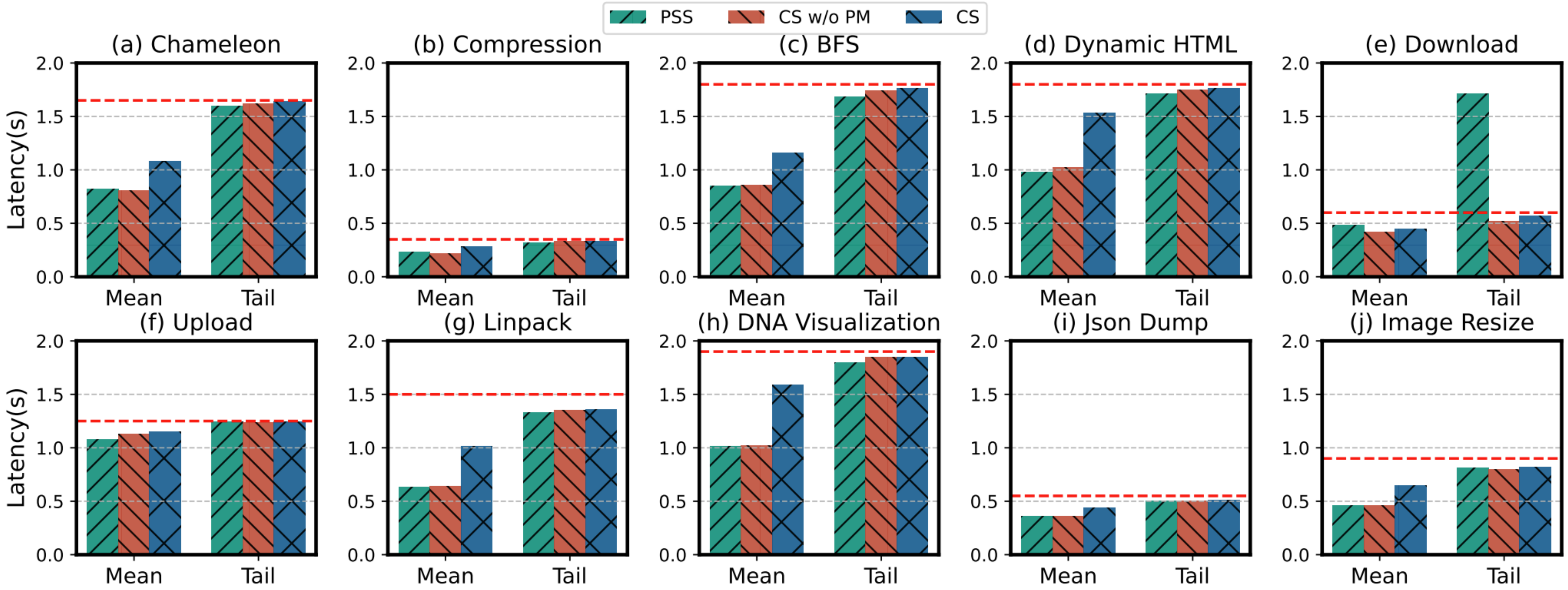
[1] Kim, et al. "Functionbench: A suite of workloads for serverless cloud function service." 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, 2019.
[2] Copik , et al. "Sebs: A serverless benchmark suite for function-as-a-service computing." Proceedings of the 22nd International Middleware Conference. 2021.

Evaluation: Experiment Result



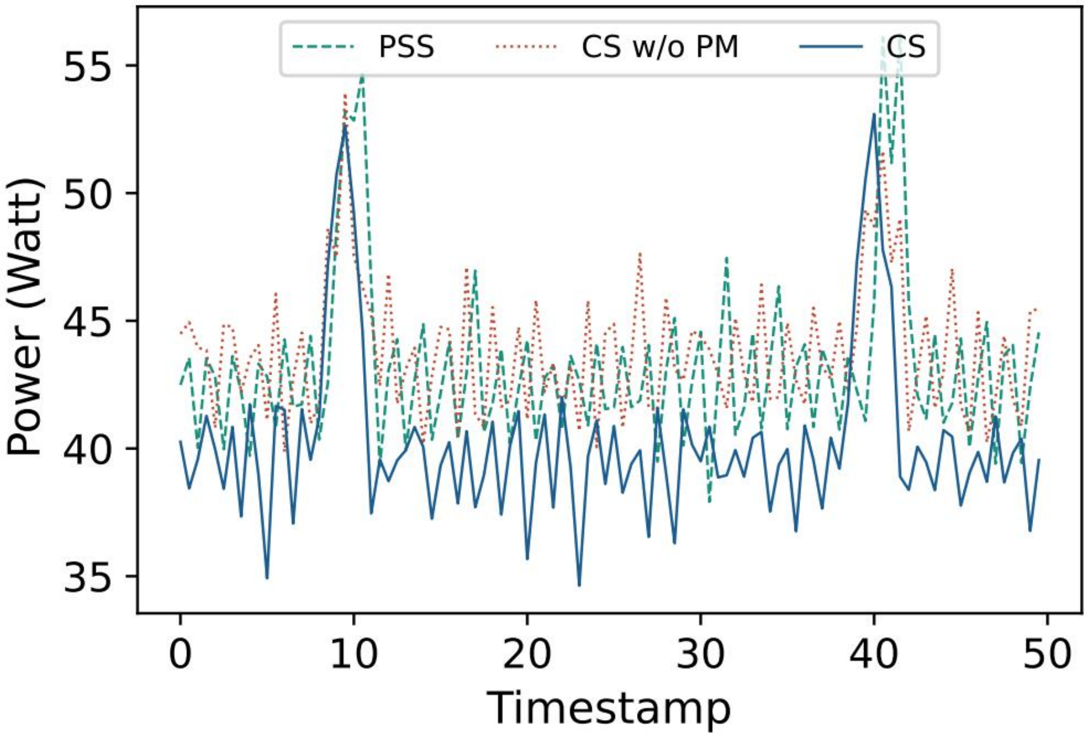
- CS achieves **an average of 8% power** saving when the workload is below the threshold.
- CS can also have **power reduction about 4%** compared with PSS on high resource pressure.

Evaluation: Experiment Result



- Our work can schedule the tasks before the deadlines (while PSS can not in some cases).
- Compared with the CS without power management (CS w/o PM), we show acceptable latency reduction.

Evaluation: Experiment Result



- CS can adapt system dynamically with **faster frequency configuration** and **lower power consumption**.
- CS is sensitive to fluctuations in workload. When the workload decreases, the system transitions into a power-saving mode to **enhance efficiency with QoS guaranteed**.

Conclusion

(1). Function Level Tuning Table

- Detailed latency and power analysis of serverless functions.

(2). Function Latency Predictor

- Accurate ML-based latency prediction methods for efficient core binding.

(3). QoS-aware Request Queue

- Core-level scheduling mechanism with low-overhead core configuration.

(4). Core-Level Scheduler (CS)

- Significantly saving power cost under QoS guarantee.

上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

Thank You & Questions

Contact me at: jing618.sjtu.edu.cn