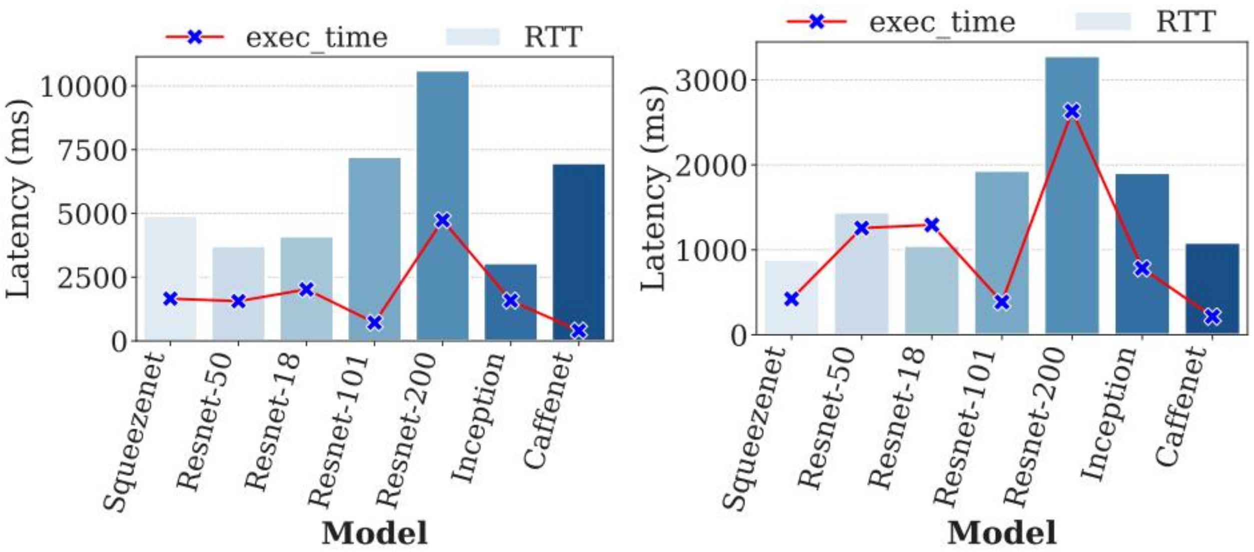
Flame: A Centralized Cache Controller for Serverless Computing

Yanan Yang (Tianjin University)
Laiping Zhao (TianJin University)
Yiming Li (Tianjin University)
Shihao Wu (Tianjin University)
Yuechan Hao (Huawei Cloud)
Yuchi Ma (Huawei Cloud)
Keqiu Li (Tianjin University)

Presented at: ASPLOS 2024, USA.
29 April, 2024

Cold start Problem in Serverless Computing

- Stateless functions suffers from a problem of cold start
- Function' startup time can be **orders of magnitude higher** than the execution time



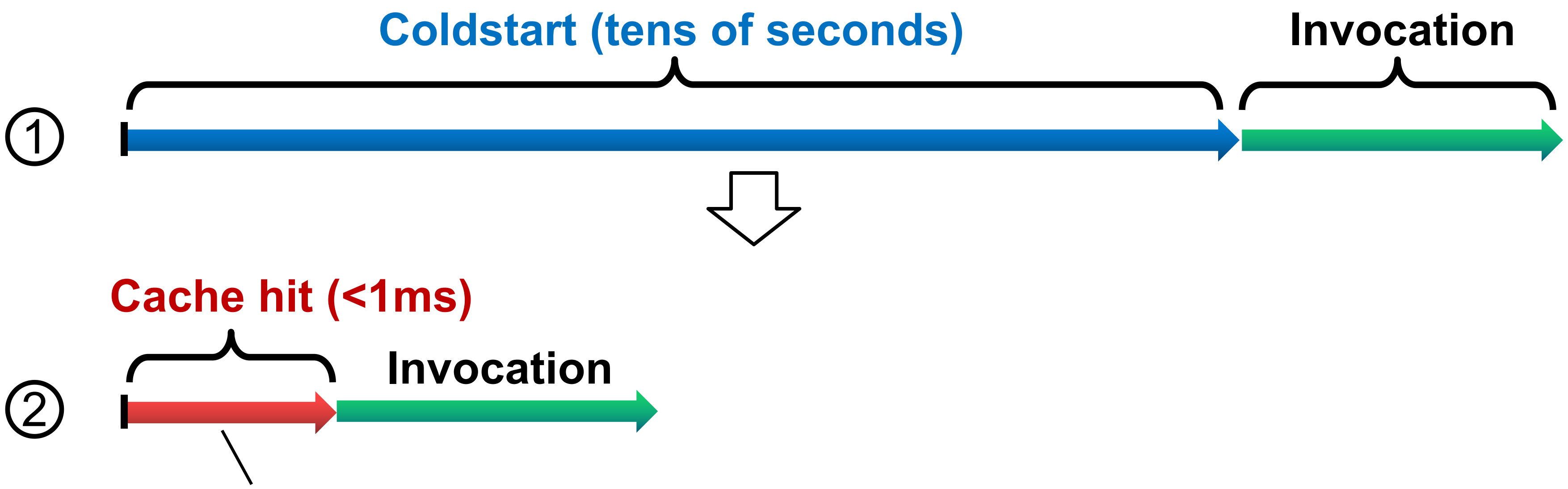
(a) Cold Start Latency. **(b)** Warm Start Latency.

*Fifer [Middleware 2020]

The Function Caching

- Eliminating coldstart from cache-based function prewarming

  E.g., Instance pool, user-side function reserve



unpausing a cached container takes only 0.5 ms *[OSDI 2023]

Serving user request from a cached function instance can avoid long coldstart latency

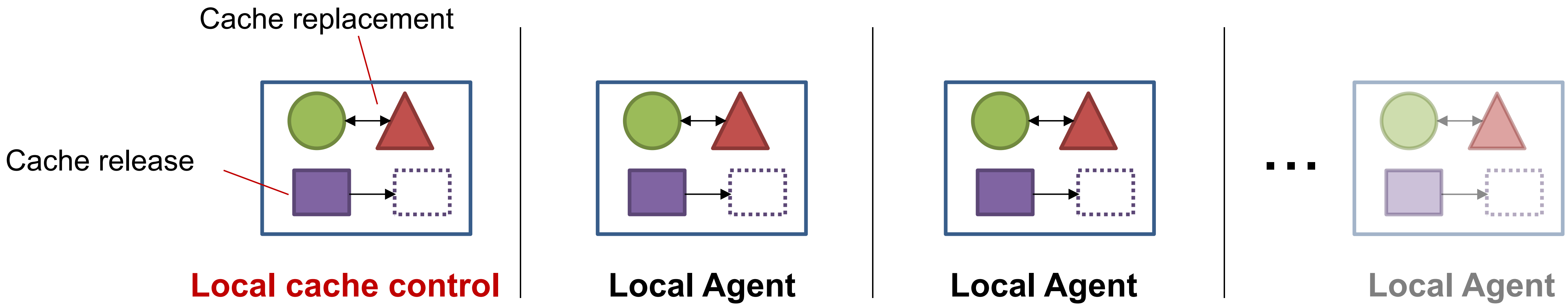
The Non-trival Cache Cost

- Caching functions will consume the limited cloud resources
  - A production system from one of the China's largest cloud providers
  - Serving **2 billions** of requests every month
  - Using **>20%** of memory to cache functions for **<1%** of coldstart ratio

Q: How to improve the function cache efficiency?

Existing Approaches

- Time-to-live-based (TTL) function keepalive *[AWS 2016]
- Dynamic function prewarming *[ATC 2020]
- Priority-based function caching *[ASPLOS 2021]



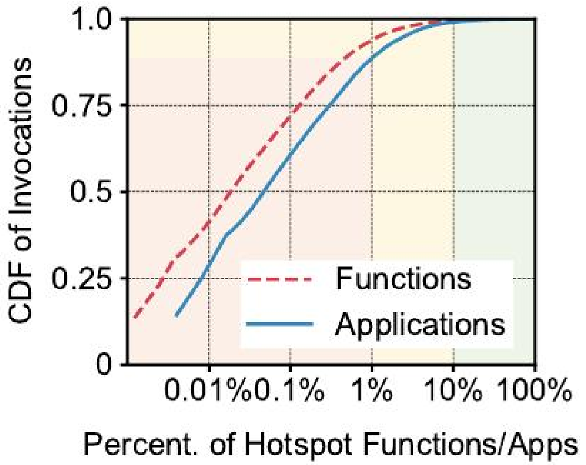
To improve the cache efficiency, serverless platforms commonly launch a local controller in each server, which manages the creation and destruction of cached instances

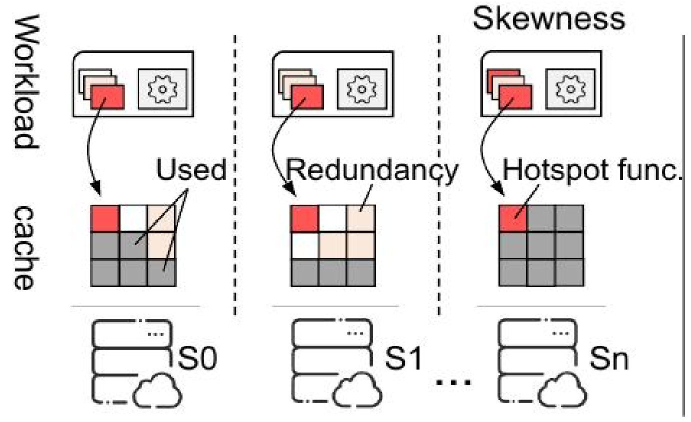
Our Contributions

■ Key observations

- **Cache contention from hotspot functions**
- **Cache redundancy across different servers**

*[ASPLOS 2019]





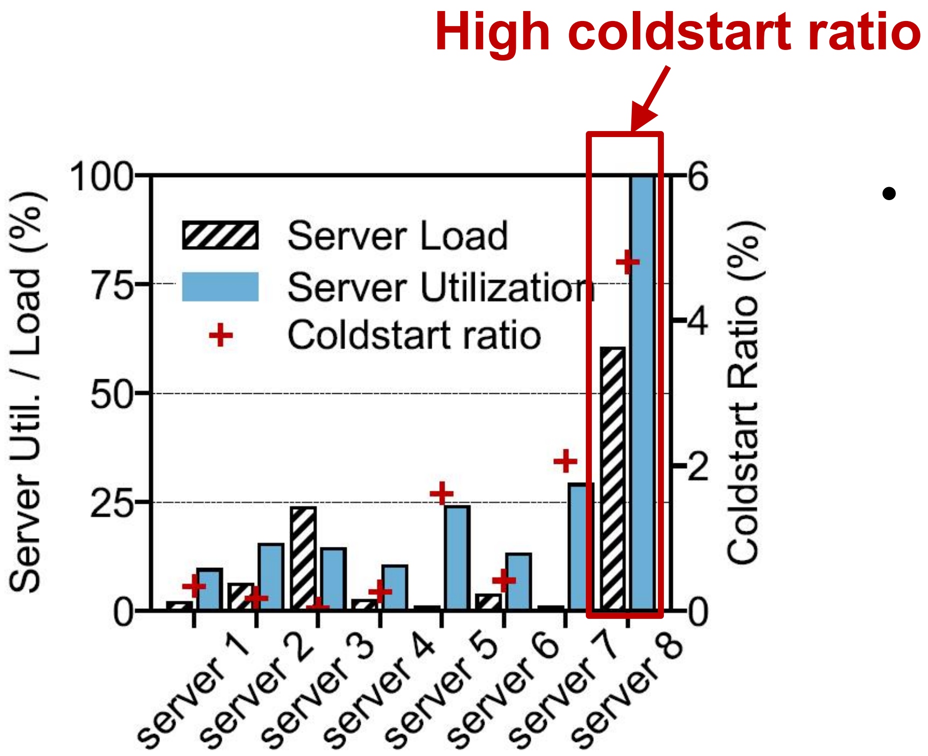
(a) Local Cache Control

workload skewness from hotspot functions

The "local cache control" is far from achieving high cache efficiency due to the workload skewness across servers

Observations

■ Cache contention



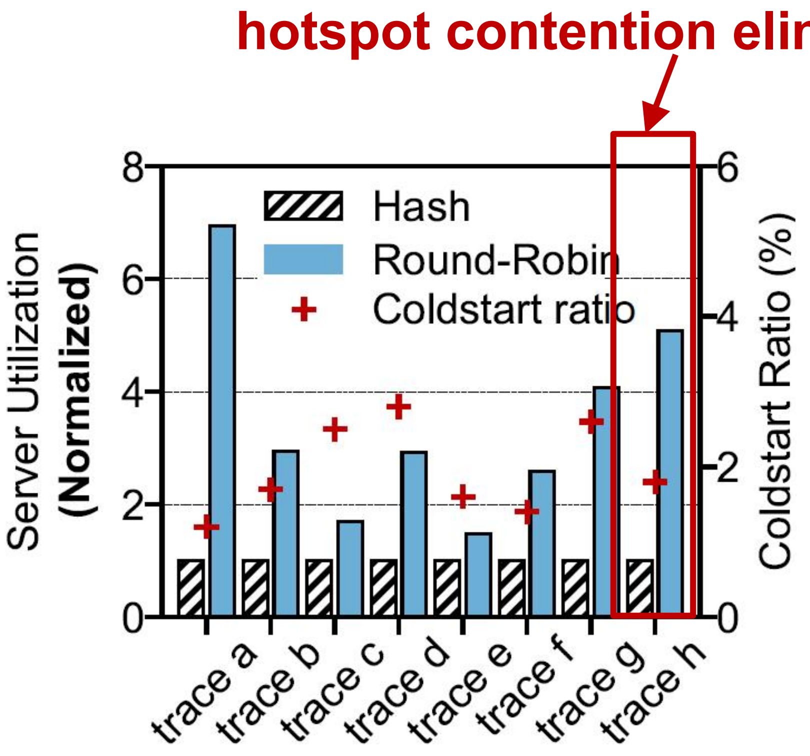
- The **cache contention of hotspot functions** in some servers can lead to **38%** of coldstart ratio fluctuations, degrading both resource efficiency and performance

e.g., 4.7% on server 8 v.s. 0.04% on server 3

Hotspot contention can result in 100x of function performance difference across servers

Observations

■ **Cache contention**

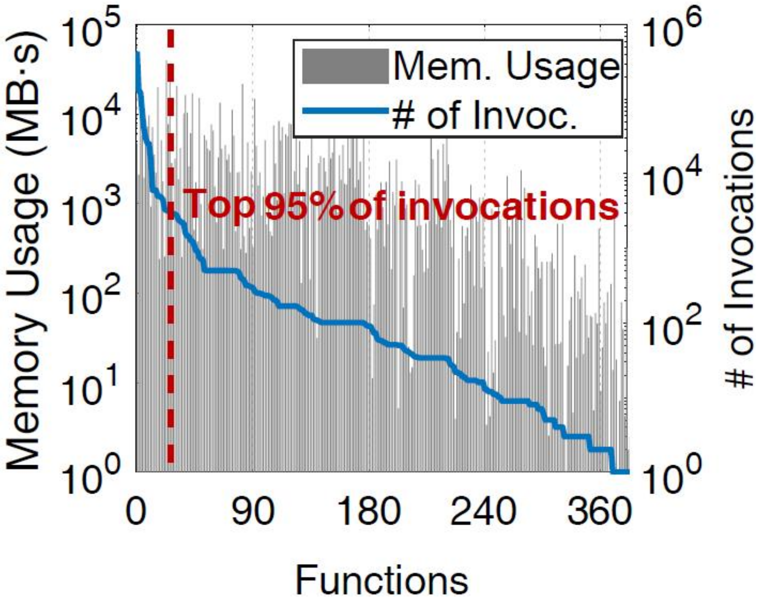


- Distributing workload on multiple servers evenly, the performance bottleneck on local server can be improved
- However, this may violate the locality and reduces cache hit ratio, causing 3× more cache resource usage under the same performance

Simply switching the load dispatching rule is insufficient to achieve high cache efficiency

Observations

■ Cache redundancy



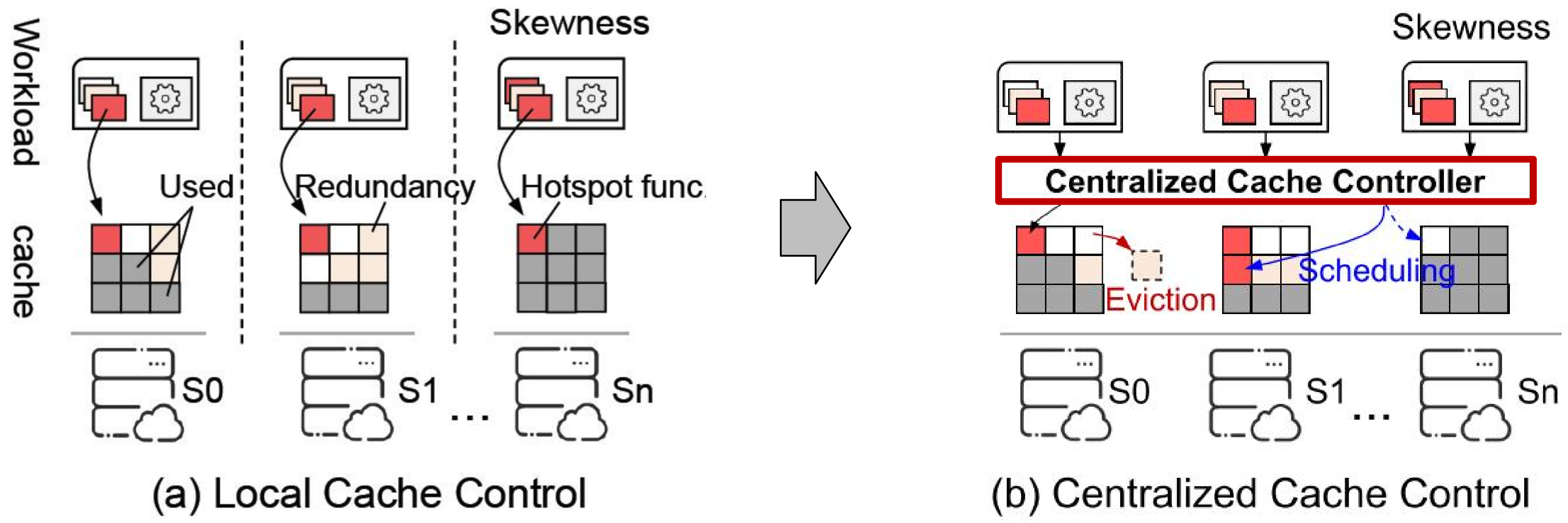
- The top-20 hotspot functions contribute nearly 95% of invocations but take only 20% of memory usage over the total 385 functions

- It also means that more than 75% of the resources are consumed by caching functions that are seldom invoked.

Local cache control can also lead to much cache redundancy in serverless cluster

Summarize

- An observation of the low cache efficiency problem from local cache control
- A centralized cache control system (Flame) to efficiently manage the cached functions via a global view of cluster status



Design of Flame

■ Overview:

- Flame adopts a globally “centralized cache control" for managing caching in a serverless cluster, thus to enable an optimized cache-hit ratio and resource efficiency

■ The 3W questions:

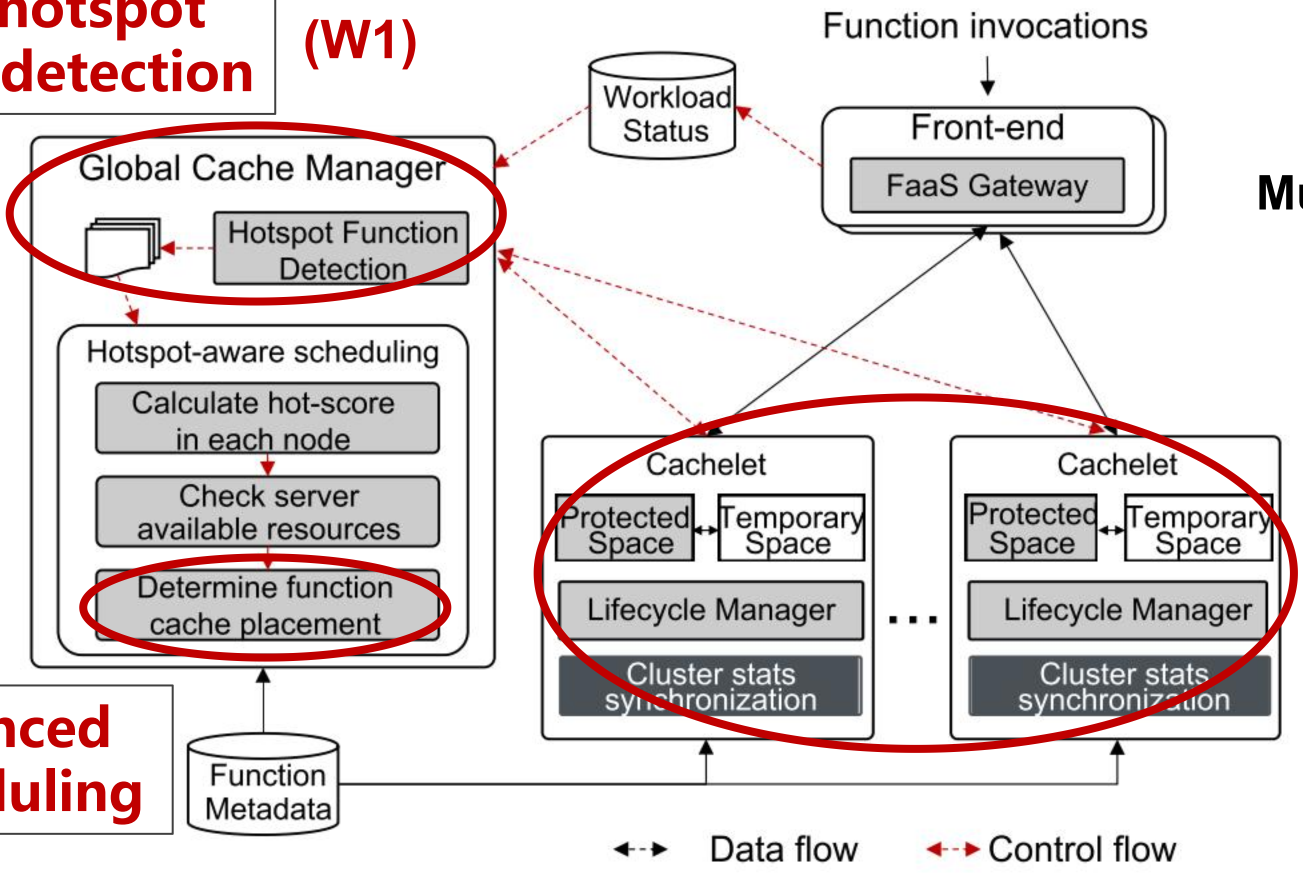
- Which function should be cached? (W1)
- Where the cached functions should be cached? (W2)
- When the cached functions should be released? (W3)

Design of Flame

Global hotspot function detection

(W1)

- Two-layer controller design:
  A global CacheManager
  +
  Mutilple subcontrollers (Cachelets)



Load-balanced cache scheduling

(W2)

Hybrid cache allocation policy

(W3)

An overview of Flame

Design of Flame

■ Global hotspot function detection (W1)

[ASPLOS'19]

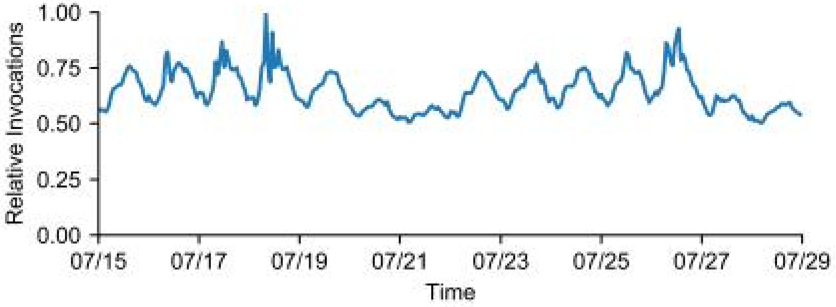


Figure 4: Invocations per hour, normalized to the peak.

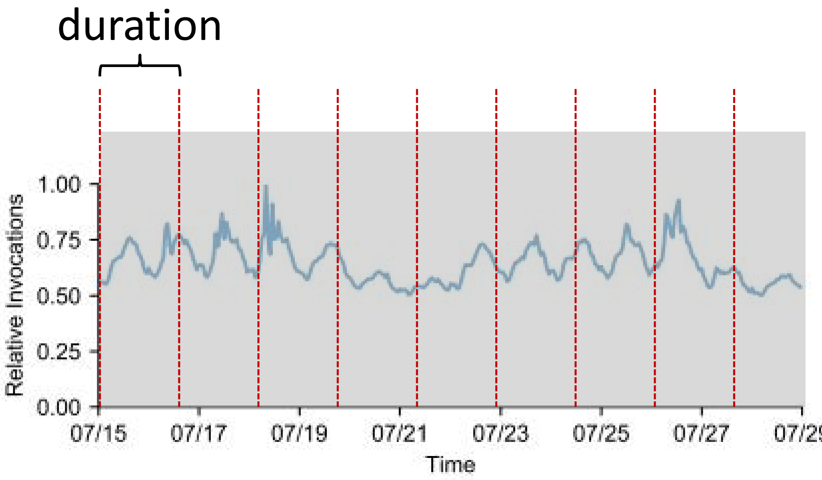


Figure 4: Invocations per hour, normalized to the peak.

$counter$:

The number of total requests in duration

- Exponentially decaying algorithm

Hot-score of a function: $H_i = \sum_{t=1}^{T} 2^{1-t} counter_i[t]$

T: # of historical durations

$counter_i[t]$: The tth invocation counter of function i

- **Top-N Hotspot function list**

$$\{H_i | i = 1,2,\dots,m\}, \mathrm{H}_i \geq H_{i+1}, 1 \leq i < m$$

$$\sum_{i=1}^{\mathrm{N}} H_i \geq \frac{1}{2} \sum_{i=1}^{m} H_i$$

m: # of total functions N: # of hotspot functions

Design of Flame

■ Load-balanced Cache Scheduling (W2)

- Filter the servers with enough resources (idle resources + nonhotspot resources)
- Calculate the hot-score in each server
- Determine the cache placement (minimum hotspot aggregation)

```
Function GetHotScore(S_k, H, M):
    hot_score_k ← 0; resource_k ← 0;
    Initialize F_k, I_k as the cached functions and instances in
      server k;
    for F_{k_i} ∈ F_k do
        if I_{k_i} ≠ [] then
            if isHotspot(F_{k_i}) then
                score_k ← score_k + H_i;
            else
                for I_{k_i} ∈ I_k do
                    if isIdle(I_{k_i}) then
                        resource_k ← resource_k + M_i;
    resource_k ← resource_k + S_k; // Add the available
      resources in the server
    if resource_k > 0 then
        if hot_score_k == 0 then
            hot_score_k ← MAX_VALUE;
        else
            hot_score_k ← hot_score_k / resource_k;
              // Calculate the weighted score
    return ⟨hot_score_k, resource_k⟩;
```

Design of Flame

■ Load-balanced Cache Scheduling (W2)

- Filter the servers with enough resources (idle resources + temporary space)
- Calculate the hot-score in each server
- Determine the cache placement (minimum hotspot aggregation)

```
Function GetHotScore(S_k, H, M):
    hot_score_k ← 0; resource_k ← 0;
    Initialize F_k, I_k as the cached functions and instances in
      server k;
    for F_{k_i} ∈ F_k do
        if I_{k_i} ≠ [] then
            if isHotspot(F_{k_i}) then
                hot_score_k ← hot_score_k + H_i;
            else
                for I_{k_i} ∈ I_k do
                    if isIdle(I_{k_i}) then
                        resource_k ← resource_k + M_i;
    resource_k ← resource_k + S_k; // Update the available
      resources in the server
    if resource_k > 0 then
        if hot_score_k == 0 then
            hot_score_k ← resource_k/0.001;
        else
            hot_score_k ← resource_k/hot_score_k;
            // Calculate the weighted score
    return ⟨hot_score_k, resource_k⟩;
```

- Hot-score aggregation of a server

$$S = \sum_{j=1}^{M} hot\_score(inst_j), inst_j \in hotspot\_list$$

M: # of instances in the server

$inst_j$: The jth instance in the server

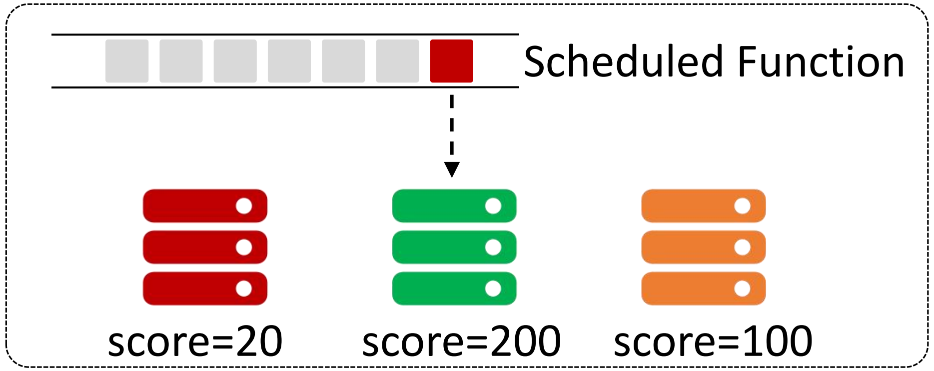
Design of Flame

■ Load-balanced Cache Scheduling (W2)

- Filter the servers with enough resources (idle resources + temporary space)
- Calculate the hot-score in each server
- Determine the cache placement (minimum hotspot aggregation)

```
    resource_k ← resource_k + S_k; // Update the available
     resources in the server
    if resource_k > 0 then
        if hot_score_k == 0 then
            hot_score_k ← resource_k/0.001;
        else
            hot_score_k ← resource_k/hot_score_k;
             // Calculate the weighted score
    return ⟨hot_score_k, resource_k⟩;
```

```
max_score ← 0;
server_index ← −1;
for S_k ∈ S do
    ⟨hot_score_k, resource_k⟩ ← GetHotScore(S_k, H, M);
     // Calculate the aggregated function hot-score and
     the available resource in server k
    if R ≤ resource_k then
        if hot_score_k > max_score then
            max_score ← hot_score_k;
            server_index ← k; // Find a server to
             schedule new instance
return server_index;
```

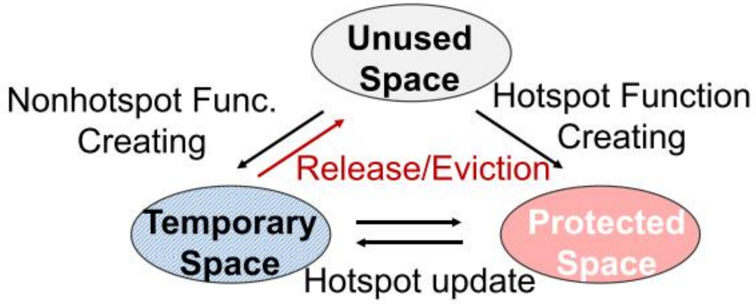


An example of cache scheduling

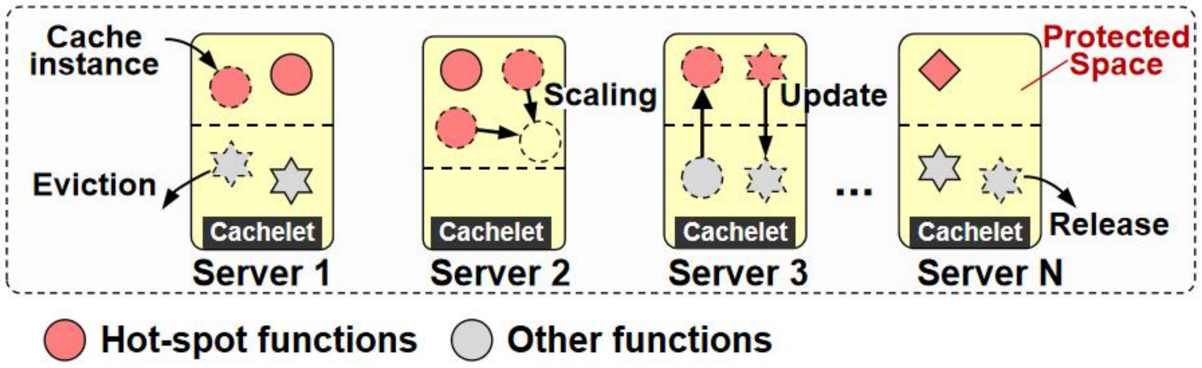
Design of Flame

■ Hybrid cache allocation policy (W3)

- Protected/Temporary Memory Partitions
- **"First-class caching"** for Hotspot function
- **"Best-effort caching"** for non-hotspot funcitons



(b) Lifecycle switch of cached functions



Implementation

- Integration on OpenFaaS
  - Approximately 4,000 lines of Golang
  - Component modifications such as gateway, faas-netes, alert-Manager
  - Adding new components like repository, CacheManager and Cachelet
- Simulator
  - Approximately 12,000 lines of Java
  - For large scale of evaluation
  - Quickly deployed in a local environment

Evaluation

■ Testbed

- 8-server local cluster with Ubuntu 16.04, Docker 18.03
- 128 GB RAM and 16-core Intel Xeon Silver 2.50 GHz CPUs, 10 Gbps network
- A large scale of simulation

■ Workload

- 8 different workload set from Azure’s function trace, 385 functions across 7 days
- Diverse workload characterizations

Table 1: Experimental testbed configuration.

Component	Specification	Component	Specification
CPU device	Intel Xeon Silver-4215	Shared LLC Size	11MB
Number of sockets	2	Memory Capacity	128GB
Processor BaseFreq.	2.50 GHz	Operating System	Ubuntu 16.04
CPU Threads	32 (16 physical cores)	SSD Capacity	960GB
Memory Bandwidth	20 GB/s	Network bandwidth	10 Gbps

Table 2: Workloads for evaluation.

Workload Name	# of Func.	# of Requests	Avg. Reqs/s	(Top-N func.: >90% total invocs.) # of Functions	Mem Usage perc.
Trace A	385	3,111,827	36	14 (3.64%)	47.57%
Trace B	385	11,363,701	132	4 (1.04%)	6.87%
Trace C	385	12,411,923	144	3 (0.78%)	85.09%
Trace D	385	24,774,632	287	2 (0.52%)	52.81%
Trace E	385	3,462,726	40	12 (3.12%)	72.77%
Trace F	385	2,735,329	32	16 (4.16%)	82.89%
Trace G	385	3,665,540	42	13 (3.38%)	2.35%
Trace H	385	2,183,501	25	17 (4.42%)	38.03%

Evaluation

■ Comparison systems

- TTL-based keepalive policy
- FaasCache [ASPLOS 2021]
- CH-RLU [HPDC 2022]
- Icebreaker [ASPLOS 2022]

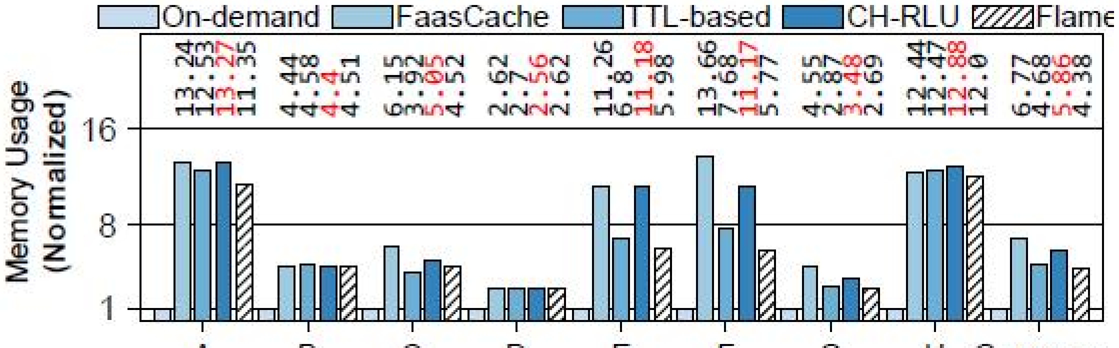
■ Metrics

- Coldstart ratio, latency (i.e., the duration between request arrivals in getaway and invocation completion)
- Overall memory usage for caching and executing requests

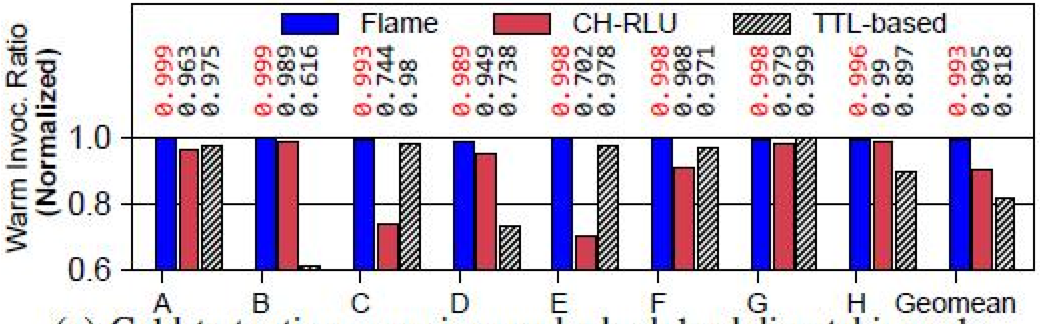
Evaluation

■ Overall Performance

- How Flame perfroms than the compared methods under different workloads?



(a) Cache resource usage comparison under hash load dispatching rule



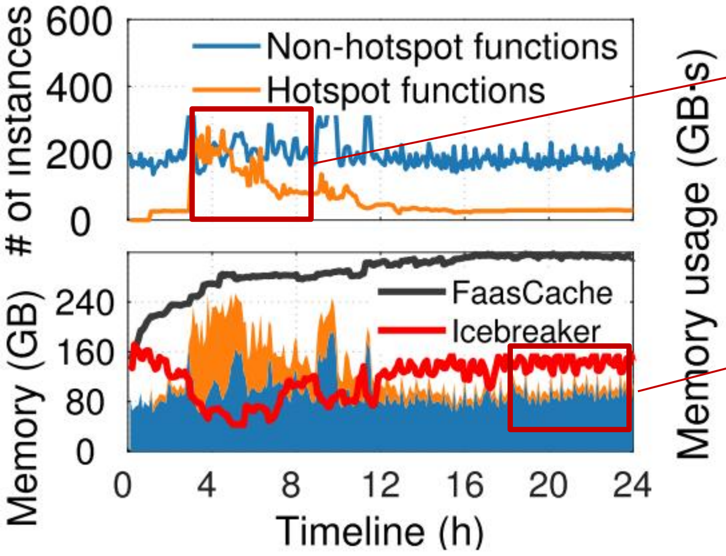
(a) Coldstart ratio comparison under hash load dispatching rule

- Flame can reduce the cache resource usage by 26%-54% on average and improve the coldstart ratio by more than 7× in serverless cluster

Evaluation

■ Cache resource allocation

- How Flame perfroms than the existing methods under different workloads?



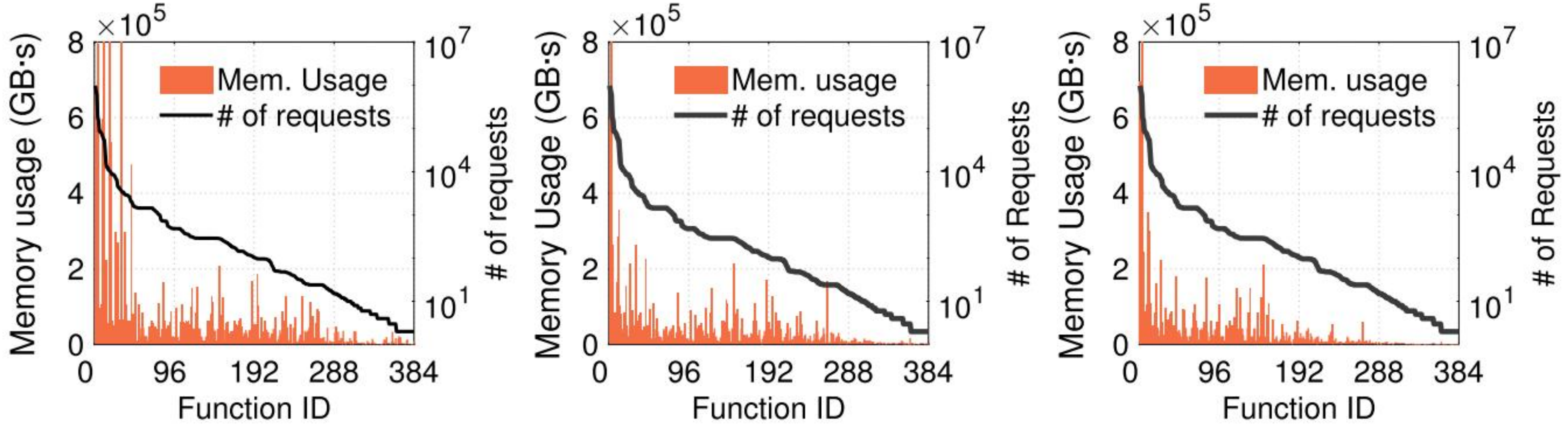
Workload spikes

Less resource consumption

- Flame's hybrid resource allocation strategy mainly focuses on hotspot functions, which consumes less cache resources and can dynamically adjust it with workload changes

Evaluation

■ Cache usage breakdown



(b) FaasCache (c) Flame (d) Icebreaker

25,188 TB·s (FaasCache)

v.s.

11,803 TB·s (Flame)

- Flame can significantly reduce cache usage and redundancy between different functions

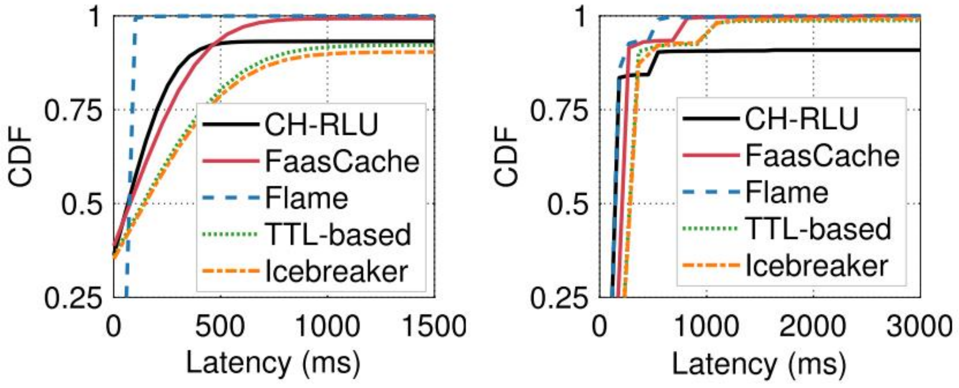
Evaluation

■ Function latency distribution

- How Flame perfroms when compared with existing methods?

Table 3: Coldstart ratio and drop ratio breakdown.

Methods	Coldstart Ratio	Drop Ratio	Overall
Flame	0.86%	0.02%	0.88%
FaasCache	1.83% (↑2.1×)	0.77% (↑38.7×)	2.61% (↑2.9×)
CH-RLU	5.34% (↑6.2×)	1.41% (↑70.5×)	6.75% (↑7.7×)
TTL-based	4.68% (↑5.4×)	1.63% (↑81.4×)	6.31% (↑7.2×)
Icebreaker	6.11% (↑7.1×)	1.34% (↑66.7×)	7.45% (↑8.4×)



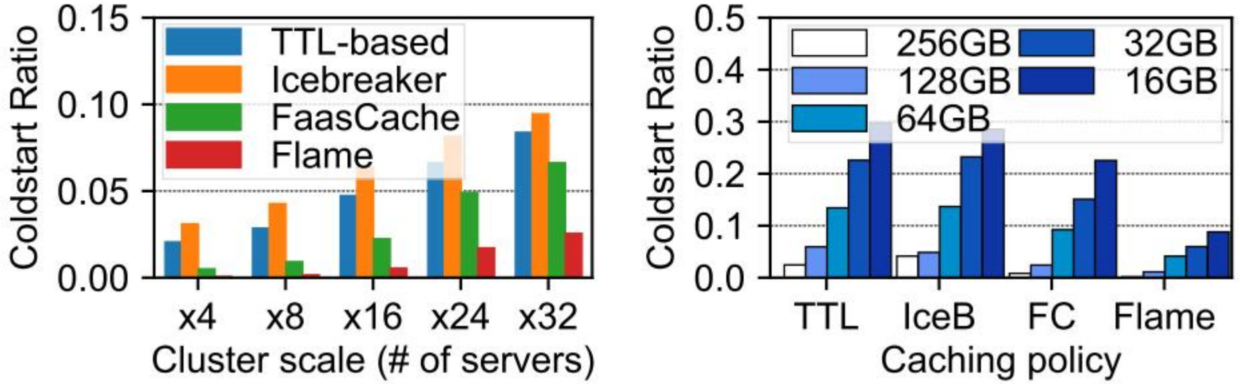
(a) Hashing-based load-balancing (b) Round-robin-based load-balancing

- Flame can reduce the 99th percentile latency more than 10× by mitigating the coldstart overhead.

Evaluation

■ Sensitivity & Scalability

- How Flame perfroms when the cluster scales or memory capacity changes?



(a) Performance evaluation under different cluster scales

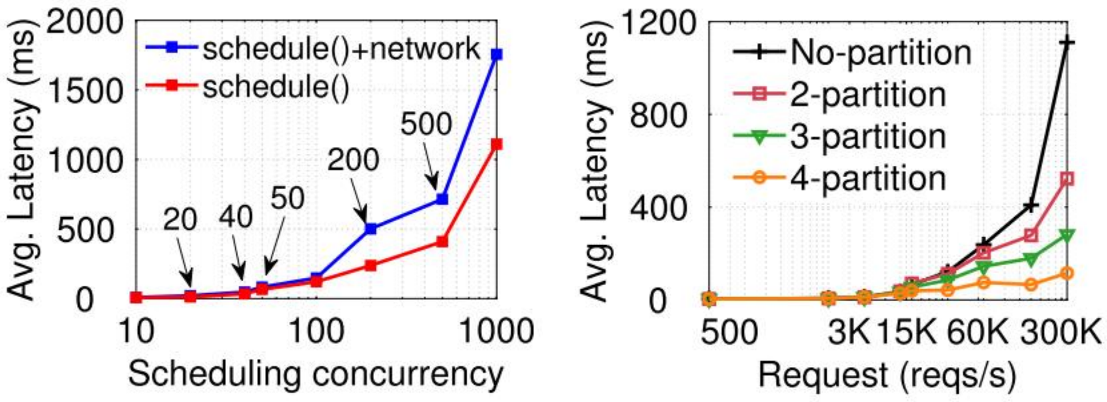
(b) Performance evaluation under different server memory sizes

- Flame can still achieve better performance when changing cluster size or server memory capacity.

Evaluation

■ System overhead

- How is Flame's system overhead?



(a) Cache scheduling time (b) Scheduling time optimization

- Reading the metadata of 385 functions takes <5 ms
- Sync. between the Cachelet and CacheManager takes ~0.5 ms
- CacheManager takes 175 MB of memory
- Cachelet takes <100 MB of memory

- Flame generates negligible overhead in system resource usage and decision latency, and it can be easily extended in large scale of workload scenarios

Conclusion

- Flame aims to improve the cache efficiency in serverless computing
- It addresses the hotspot contention and cache redundancy problems from a centralized cache control design
- Flame can help to save approximately 4,000,000$ every year in our production system

Paper Access

Full Paper: https://dl.acm.org/doi/10.1145/3623278.3624769

Source File: https://github.com/ykiauz/Flame/tree/main/benchmark

Contact

ynyang@tju.edu.cn

Affiliation

College of Intelligence & Computing, Tianjin University

Tianjin, China

Thanks for listening!