Flame: A Centralized Cache Controller for Serverless Computing

Yanan Yang
College of Intelligence & Computing (CIC), Tianjin University, Tianjin Key Lab. of Advanced Networking (TANKLAB)
Tianjin, China
ynyang@tju.edu.cn

Laiping Zhao*
CIC, Tianjin University, TANKLAB
Tianjin, China
laiping@tju.edu.cn

Yiming Li
CIC, Tianjin University, TANKLAB
Tianjin, China
l_ym@tju.edu.cn

Shihao Wu
CIC, Tianjin University, TANKLAB
Tianjin, China
lechou@tju.edu.cn

Yuechan Hao
Huawei Cloud Computing Technologies Co., Ltd.
Shenzhen, China
haoyuechan@huawei.com

Yuchi Ma
Huawei Cloud Computing Technologies Co., Ltd.
Shenzhen, China
mayuchi1@huawei.com

Keqiu Li
CIC, Tianjin University, TANKLAB
Tianjin, China
keqiu@tju.edu.cn

ABSTRACT

Caching function is a promising way to mitigate coldstart overhead in serverless computing. However, as caching also increases the resource cost significantly, how to make caching decisions is still challenging. We find that the prior "local cache control" designs are insufficient to achieve high cache efficiency due to the workload skewness across servers.

In this paper, inspired by the idea of software defined network management, we propose **Flame**, an efficient cache system to manage cached functions with a "centralized cache control" design. By decoupling the cache control plane from local servers and setting up a separate centralized controller, **Flame** is able to make caching decisions considering a global view of cluster status, enabling the optimized cache-hit ratio and resource efficiency. We evaluate **Flame** with real-world workloads and the evaluation results show that it can reduce the cache resource usage by 36% on average while improving the coldstart ratio by nearly 7× than the state-of-the-art method.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

*Corresponding author: laiping@tju.edu.cn

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03…$15.00
https://doi.org/10.1145/3623278.3624769

KEYWORDS

Serverless Computing, Keep-alive, Hotspot Function, Coldstart

ACM Reference Format:
Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. 2023. Flame: A Centralized Cache Controller for Serverless Computing. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, San Diego, USA, 16 pages. https://doi.org/10.1145/3623278.3624769

1 INTRODUCTION

Serverless computing provides a stateless programming model for cloud users, which allows developers to construct their services as only a set of event-triggered functions without considering resource provision and management (e.g., autoscaling). Due to the stateless nature of serverless functions, user requests suffer from a coldstart problem (i.e., launching containers from scratch for functions that are not used frequently). In a coldstart scenario, the startup time of execution environment initialization can be orders of magnitude higher than the function execution time [11], making this one of the central performance problems in serverless computing [8, 21, 27, 29].

Generally, the coldstart overhead can be mitigated through two approaches: *startup acceleration* and *function caching*. *Startup accelerating* methods attempt to reduce the function initialization time by trimming the sandbox loading process, e.g., by snapshotting [3], state reuse [2, 6, 14, 21], or lightweight virtualization [1, 12, 19]. While these mechanisms can reduce the virtual environment creation time significantly (e.g., to < 2 ms in Catalyzer [6]), the user-side overhead remains; e.g., resetting application variables and configuration files can take hundreds of milliseconds [28], and the resulting coldstart time is still unacceptable for latency-critical services. The *function caching* methods seek to invoke functions in a "warmstart" form. By caching finished functions (e.g., via Docker

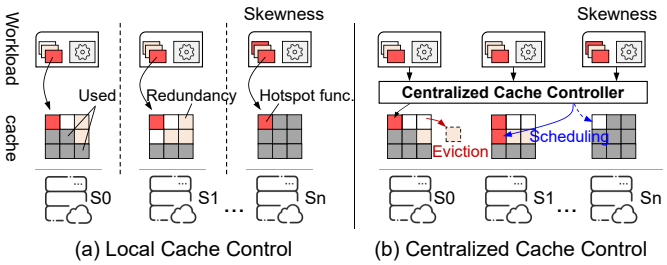


Figure 1: An overview of different cache management methods.

pause [23]) instead of reclaiming them immediately, the subsequent requests can reuse them with almost no coldstart overhead. It takes < 0.5ms to recover a paused container [30]. As serverless mainly attracts short running tasks [26, 36], function cache is a more promising way to eliminate coldstart.

However, caching function instances consumes resources on physical servers, thus increasing the cost of serverless platforms. For example, our platform, one of the China's largest cloud provider, consumes more than 20% of memory resource in serverless cluster for keeping functions alive under a < 1% of coldstart ratio. Worse, it is almost impossible to cache all functions due to limited server resources and varied workloads. To improve the cache efficiency, serverless platforms commonly launch a *local controller* in every physical server, which manages the creation and destruction of cached instances using TTL (Time-to-live)-based keep-alive policies [17, 22] or priority-based caching policies [8, 9].

Unfortunately, we find that the *"local cache control"* design (i.e., making caching decisions based on only a local view of invocations) is far from achieving high cache efficiency due to the skewed invocations across servers (Figure 1(a)). An analysis of Azure's function trace [27] shows that 90% of invocations are sent to only 10% of hotspot functions. Hence, servers deployed with hot functions may not have sufficient resources for caching, whereas servers deployed with cold functions may cache some instances that are seldom called. The lack of a global view of the distributions of hot and cold functions makes it difficult for the controller to make optimal cache decisions. Our analysis shows that > 50% of cached instances are seldom invoked due to the suboptimal cache decisions made by *local controllers*. In contrast, the contention among hotspot functions in some servers can lead to 38% of coldstart ratio fluctuations, degrading both resource efficiency and performance.

In this paper, inspired by the idea of software defined network management [7, 32], we argue for a globally *"centralized cache control"* (Figure 1(b)) for managing caching in a serverless cluster. By decoupling the cache control plane from local servers and setting up a separate centralized controller, caching decisions can be made from a global perspective of cluster status, enabling an optimized cache-hit ratio and resource efficiency.

The *"centralized cache control"* design poses several challenges for a serverless system. First, the cache controller should answer the *"3W"* questions: (1) *Which function instances should be cached?* (2) *Where should the function instances be cached?* (3) *When can the cached instances be released?* The high volatility of the request load and resource consumption of different functions make it difficult to solve these problems. Second, the centralized controller strongly relies on support from monitoring, centralized storage and communication protocols. Integrating the controller into a serverless system with minimum overhead is also a challenge.

To overcome these challenges, we propose **Flame**, the first centralized caching manager for serverless clusters. **Flame** consists of a top controller and multiple agents (named *Cachelet*). In particular, each server is deployed with a *Cachelet*, which collects the local status information, sends it to the top controller and executes its cache decisions. **Flame** introduces a concept of "hotspot" for both functions and servers, which is used to describe the function popularity and server load pressure, respectively. To avoid the intra-server hotspot contentions, the top controller adopts an *exponentially decaying* algorithm to dynamically identify the global hotspot functions and schedules them with the *minimum hotspot aggregation* principle for caching. In each server, **Flame** uses a hybrid caching policy to manage the cached functions. Hotspot functions are cached in the protected memory space for achieving better performance, while the non-hotspot functions are cached in the temporary memory space using a "best-effort" way to utilize server's idle resources, thus to maximize the cache benefit. **Flame** also designs an adaptive reclamation algorithm to detect cache redundancy of hotspot functions, thus improving their cache efficiency.

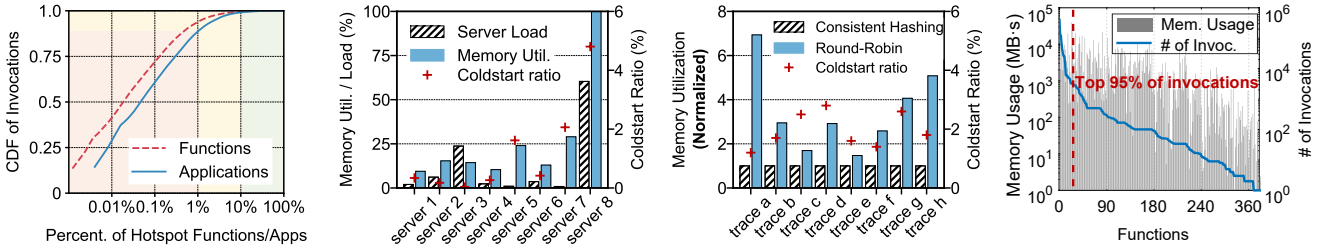
We implement **Flame** on OpenFaaS [22], a popular open-source serverless framework built upon Kubernetes [16] natively. It can achieve more than 99.1% of warm invocation ratio in serverless cluster. We use real-world applications from ServerlessBench [35] and FunctionBench [13] under Azure's production traces to evaluate **Flame**'s cache efficiency. The evaluation result shows that it can reduce the cache resource usage by 36% on average while improving the coldstart ratio by nearly 7× than the existing local cache control methods.

Contributions. We highlight the contributions of this paper as follows:

- **Problem**. An analysis of the low efficiency of the local cache controller due to the skewed invocations.
- **Flame**. A centralized cache controller for managing the caching of serverless functions, which resolves cache redundancy and hotspot contention problem using a cluster-level hotspot-aware caching policy.
- **Evaluation**. A full-system implementation on OpenFaaS and evaluations on real-world applications to demonstrate the high cache efficiency of **Flame**.

2 MOTIVATION

The stateless nature of serverless functions allows the platform to automatically adjust the number of function instances with workload changes. Typically, a serverless cluster contains hundreds of servers, serving tens of millions of requests from thousands of functions. Each function is running inside a container such as Docker [33] or a lightweight virtual machine such as Firecracker [1]. By simply pulling the user code and dependency files, these stateless functions can be easily extended to different servers. For each deployed function, the gateway receives user requests and forwards them to servers with available function instances for processing. When a function instance finishes execution, it will be cached on



(a) Function call distribution (b) Hotspot contention (c) Low resource efficiency (d) Cache redundancy

Figure 2: (a) Function call distribution of the Azure trace. (b) Hotspot contentions in local cache control. (c) Cache memory usage under different workload distributions in local control. (4) Cache redundancy in local cache control.

the local server rather than being released immediately; thus, subsequent requests can reuse it with near-zero coldstart latency.

As hotspot functions are widely common in serverless scenarios, this makes the workload skewness become a common phenomenon in serverless platforms. For example, as shown in Figure 2(a), more than 90% of the requests in Azure's function trace come from approximately 10% of hotspot functions. When multiple hotspot functions are colocated on several local servers, their system pressure can dramatically increase and lead to workload skewness in the cluster. Additionally, load-balancing rules such as consistent hashing, prefer to forward a function's requests to the last accessed servers, and "server lock-in" can also result in many skewed invocations for hotspot functions. In the following, we give several observations to explain why local cache control in existing serverless platforms is insufficient to achieve high cache efficiency due to workload skewness.

2.1 Limitations of Local Cache Control

We implement FaasCache, which is a state-of-the-art local cache control method [8], on an 8-server serverless cluster. Each server in the cluster is configured with 128 GB memory. To evaluate the performance of FaasCache, we deploy several real-world applications and conduct experiments following the workload characteristics in Azure's function traces. Additionally, we employ two different load dispatchers (consistent hashing and round-robin) to observe the caching behaviors in the cluster.

Observation #1: *Hotspot contentions: Local cache control is unable to handle workload skewness in serverless cluster. Hotspot contention can result in 100× of function performance difference across servers.*

Figure 2(b) shows the server load, memory utilization and coldstart ratio in each server under the hashing-based load-balancing rule, where the server load represents the fraction of processed requests in the whole workload trace. We see that server 8 has the highest server load and memory usage ratio since more than 50% of hotspot function invocations are dispatched to it for processing, while server 3 only processes approximately 20% of total requests from hotspot functions. Due to a lack of resource pooling ability in local cache control, the resource contention between colocated hotspot functions on server 8 is more serious in the cluster, which increases its coldstart ratio by 100× higher than the other servers (4.7% on server 8 and 0.04% on server 3).

Eliminating workload skewness can improve system efficiency of the local cache control method. As shown in Figure 2(c), when the load-balancing rule switches from consistent hashing to round-robin, the hotspot functions are more likely to be evenly distributed on multiple servers, and the performance bottleneck on the local server can be eliminated (e.g., server 8). However, compared to the hashing-based load-balancing rule, the round-robin-based rule violates the locality and reduces the hit ratio for the cached function instances, which causes more than 3× greater cache resource usage. Additionally, hotspot functions show high temporal dynamism, and the local cache control method has difficulty in capturing workload behavior changes and leads to oscillations in local cache decisions, leading to a high coldstart ratio in the cluster (1.8% to 3.9% across different servers).

Observation #2: *Cache redundancy: Local cache control causes significant cache redundancy with more than 75% of cache resource wasted in the cluster.*

Local cache control makes cache decisions independently at each server. For each cached function, its instances may be redundantly saved on multiple servers, causing cache resource waste. Meanwhile, lacking a global view of workload behaviors makes local cache control unable to identify cluster-level hotspot functions, thus making incorrect cache decisions, e.g., consuming a large amount of resources to cache cold functions. To analyze cache redundancy in local cache control, we collect the memory usage from 384 real functions during one day's invocations in an experiment. The memory usage for each serverless function consists of the memory for caching its instances and consumption on executing requests.

Figure 2(d) shows the details, where the functions are sorted by the number of invocations in descending order. We see that the top-20 hotspot functions account for nearly 95% of invocations. However, they only take less than 20% of memory usage over all total 384 functions. We also calculate the memory usage for executing requests for the remaining 365 functions and find that they only take approximately 5% of the total memory usage, which means that more than 75% of the resources are consumed by caching non-hotspot functions in the cluster, although they are seldom invoked during the cache time. We also observe that many cached instances of the non-hotspot functions have low resource utilization (from 10% to 30%). If the caching system can identify the over-provisioned function instances and reclaim them with workload changes, the

cache redundancy can be significantly reduced to achieve better resource efficiency.

2.2 Implications

The above observations motivate us to design a centralized cache system to improve cache efficiency in serverless clusters. To overcome the shortcomings of local cache control, a caching system should be able to manage cluster resources from a global cache scheduling design to avoid a single-server performance bottleneck. When scheduling cached functions, hotspot functions should not be colocated together to avoid hotspot contention (**Observation 1**). Additionally, the caching decisions for each cached function should adaptively tune with workload and server status changes, and the cache system should be able to detect cache redundancy, thus improving cache resource efficiency (**Observation 2**). Finally, the system architecture design should be lightweight and extensible to reduce system overhead and decision complexity.

3 FLAME DESIGN

Figure 3 presents our cache system design. **Flame** adopts a two-layer controller design to implement centralized cache management and works as a back-end module in a serverless platform. It consists of a global *CacheManager* and multiple *Cachelets* running on each server. The global *CacheManager* interacts with the underlying *Cachelets* and periodically collects the cluster-level information (e.g., server status and workload behaviors) to make cache decisions. When a user request arrives, the front-end dispatcher will dispatch it to the server with the cached instance, if any. Otherwise, the cache scheduling algorithm in *CacheManager* is triggered under acache-miss. It selects a server with the lowest hotspot contention and launches a new function instance to process user request. Once the function instance finishes execution, the *Cachelet* caches it in memory and manages its lifetime following the *CacheManager*'s instructions.

When **Flame** is deployed in a serverless cluster, its launching and operating takes two steps: *function registration* and *cache management*. (1) *Function registration*: For each deployed function, **Flame** first collects its metadata information and registers the function in a cluster-provided database. The metadata information includes the function ID, image name and memory usage, which is used in **Flame**'s cache allocation and recycling. The function registration is simple and is done only once for each new deployed function. (2) *Cache management*: The cache management of **Flame** works on two planes. The control plane, via the global *CacheManager*, determines whether a function needs to be cached, where it should be scheduled, and how long it should be cached in a back-end server. The cache plane, which includes the cached function instances and functional components such as *Cachelet* and *gateway*, dispatches and executes user requests.

Flame adopts a hotspot-aware caching and scheduling design to address the 3W questions. At the control plane, the *CacheManager* maintains a global hotspot function list and designs a *dynamic hotspot detection* mechanism to identify the cluster-level hotspot functions (§ 4.1). To avoid hotspot contentions, it also introduces a hotspot concept for each server, which is used to describe their load

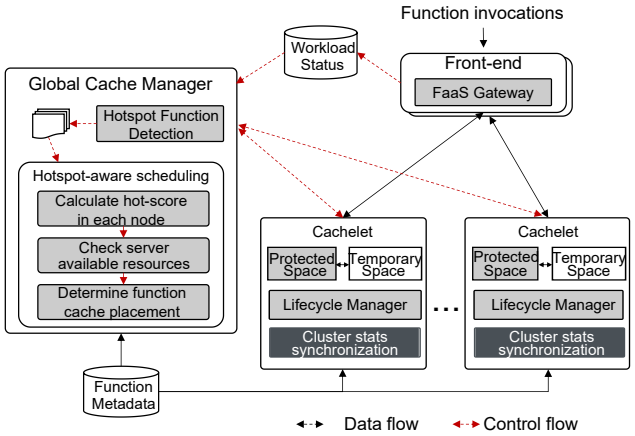


Figure 3: System architecture of Flame.

pressures. When a hotspot function needs to be cached, the *CacheManager* calculates the hot-score on each server and schedules a cached instance based on a *minimum hotspot aggregation* principle, which can balance both the server load and cache hit ratio (§ 4.2).

As the available cache resources in a server are limited and can vary with workload changes, **Flame** adopts a hybrid caching policy to improve cache resource efficiency. At the cache plane, the *Cachelet* synchronizes the global hotspot function list and periodically updates it from the *CacheManager*. When a hotspot function instance is scheduled on a server, it will be cached in a *protected memory space* after invocation, thus receiving a high cache benefit. While the non-hotspot functions are cached in a *temporary memory space* in a best-effort manner (i.e., using a TTL-based keep-alive policy but can be evicted at any time) to utilize a server's idle cache resources. There are two advantages to this hybrid caching strategy. First, it can maximize function performance by reducing coldstart invocations caused by second-class citizen phenomena (i.e., the non-hotpot functions suffer from a high coldstart ratio due to resource preemption from hotspot functions). Second, it can improve system robustness and mitigate function performance fluctuations due to hotspot detection delay and errors.

To reduce cache redundancy, **Flame** also designs an adaptive cache reclamation algorithm to dynamically monitor the cache reuse rate of hotspot functions and reclaim the over-provisioned instances, thus reducing the cache resource waste (§ 4.3).

4 HOTSPOT-AWARE CACHE SCHEDULING

We now introduce the three key modules of **Flame**, *dynamic hotspot detection*, *hotspot-aware instance scheduling* and *cache resource allocation*, and explain how these mechanisms can solve the 3W questions.

4.1 Dynamic Hotspot Detection

The *dynamic hotspot detection* mechanism is used to identify the global hotspot functions in serverless cluster. Inspired by Java Virtual Machine (JVM) [4], the *CacheManager* uses an invocation counter to record the number of invocations for each serverless function. Records collected from each server are aggregated to derive function-level hot-scores. Since serverless computing is widely

used in interactive services, such as web services and machine learning, the workload behaviors of these functions are varying over time. The *CacheManager* dynamically records the number of invocations for each function within a configurable duration (e.g., 1 hour). When a new duration starts, the function invocation counters from the last duration are stored and a new invocation counter is created.

The function's hot-score is calculated with an *exponentially decaying* algorithm [5], which can help **Flame** learn the workload characteristics from historical durations and avoid the hotspot function detection oscillations caused by short-term workload bursts. The hot-score for each function is defined as:

$$H_i = \sum_{j=1}^{T} 2^{1-j} c_i[j]$$

where $c_i(j)$ represents the jth invocation counter of function i and T is the total number of historical invocation counters. Given a collection of function hot-scores $\{H_i | i = 1, 2, ..., m\}$, where m is the number of deployed functions. We sort the function hot-scores in descending order and define the top N functions as the hotspot functions, which meet the constraint $\sum_{i=1}^{N} H_i \geq r \cdot \sum_{i=1}^{m} H_i$, where r is a threshold named region size from 0 to 1. A larger (or smaller) region size means more (or fewer) of the most popular functions are classified as hotspot functions. As the setting of region size can affect the cache efficiency in **Flame**'s cache control, we conduct a lot of experiments to find the optimal region size setting and finally set it to 0.5 in our system design (discussed in § 6.2).

4.2 Hotspot-aware Instance Scheduling

The *hotspot-aware instance scheduling* mechanism is used to determine the instance placement for cached hotspot functions while avoiding hotspot contentions on the local server. The basic idea of the scheduling algorithm is *trying to spread cached hotspot functions across the servers with the minimum hotspot aggregation principle*. To achieve this goal, we also introduce the concept of hotspot for back-end servers to describe their load pressure, which is defined as the aggregated hotspot score of all hotspot functions cached on one server.. When a function scheduling occurs under a coldstart invocation, the *CacheManager* collects the available cache resources in each server and calculates their hot-scores to determine the cache placement. Note that the server status information is collected using a periodic synchronization mechanism (every 5 s) between the *CacheManager* and *Cachelets*, thus reducing the communication delay. **Flame** also supports forced synchronization to address the issue of data inconsistency.

Algorithm 1 shows the details. In the serverless cluster, a function's resource requirements can be represented as a multidimensional vector, including the CPU, memory and I/O requirements. Let R be the resource requirement of a newly launched function instance; the available resources of the physical servers can be represented as a collection $\boldsymbol{S}$. Similarly, the functions' hot-scores and the resource consumption of the deployed functions can be defined as $\boldsymbol{H}$ and $\boldsymbol{M}$, respectively. The scheduling algorithm reads them as inputs and outputs the target server for the newly launched function instance in the cluster.

Algorithm 1: Schedule(R, $\boldsymbol{S}$, $\boldsymbol{H}$, $\boldsymbol{M}$)

```
Input:
  R ▷ The resource requirement of the newly launched function instance;
  S ▷ The available resources of servers;
  H ▷ The hot-score collection of the deployed functions;
  M ▷ The resource consumption of the deployed functions;
Output:
  server_index ▷ The placement of the newly launched instance;
1  max_score ← 0;
2  server_index ← −1;
3  for S_k ∈ S do
4      ⟨hot_score_k, resource_k⟩ ← GetHotScore(S_k, H, M);
         // Calculate the aggregated function hot-score and
         the available resource in server k
5      if R ≤ resource_k then
6          if hot_score_k > max_score then
7              max_score ← hot_score_k;
8              server_index ← k; // Find a server to
                 schedule new instance
9  return server_index;
10 Function GetHotScore(S_k, H, M):
11     hot_score_k ← 0; resource_k ← 0;
12     Initialize F_k, I_k as the cached functions and instances in
         server k;
13     for F_{k_i} ∈ F_k do
14         if I_{k_i} ≠ [ ] then
15             if isHotspot(F_{k_i}) then
16                 hot_score_k ← hot_score_k + H_i;
17             else
18                 for I_{k_i} ∈ I_k do
19                     if isIdle(I_{k_i}) then
20                         resource_k ← resource_k + M_i;
21     resource_k ← resource_k + S_k; // Update the available
         resources in the server
22     if resource_k > 0 then
23         if hot_score_k == 0 then
24             hot_score_k ← resource_k/0.001;
25         else
26             hot_score_k ← resource_k/hot_score_k;
                 // Calculate the weighted score
27     return ⟨hot_score_k, resource_k⟩;
```

In the scheduling process, the algorithm iterates the servers in the cluster and calculates their aggregated hot-scores and available resources by invoking the *GetHotScore()* module (Lines 3-4). If the resource requirement of the newly launched instance can be met, the container will be scheduled in the server with the minimum aggregated hot-score (Lines 5-8). Lines 10-27 show the details of the *GetHotScore()* function. The algorithm first initializes F_k and I_k as the cached function list and instance list in each server, respectively. Note that the subscript represents the index of the server. For each registered hotspot function on server k, its hot-score will be accumulated if it has more than one cached instance on this server (Lines 13-16); otherwise, this hotspot function will be skipped because it has no instances to compete for cache resources with the newly launched instance.

The available server resources consist of unused physical resources and evictable resources in the temporary memory space. For each non-hotspot function in server k, the resource consumption of its idle instances will be accumulated into the available resources in the server since the idle containers can be evicted forcibly when caching new function instances (Lines 17-21). Insufficient server resources or high workload pressures from hotspot

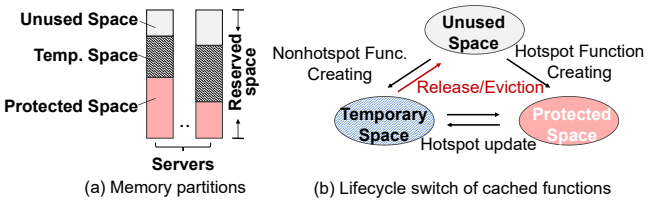


Figure 4: Cache allocation in Cachelet.

functions will lead to intra-server performance bottleneck. To address this problem, we define a server's scheduling priority as the ratio of its available resources and its aggregated hot-score (Lines 22-26). Servers with more free resources and fewer hotspot functions are preferred to schedule new function instances, which can help mitigate hotspot function contentions in a serverless cluster.

Note that the current **Flame** is designed to spread out hotspot functions across multiple servers. When the function workload is low, a separate cluster autoscaler (CA), which is common at cloud provider, can be activated to downsize the cluster through shutting down excessive servers. It is certainly possible to integrate Flame with CA together to make it support workload consolidation when placing cached function instances, thus reducing the cloud provider's cost. Additionally, since the number of cached containers is mainly limited by a server's physical memory capacity, we consider only the memory constraint in current function scheduling algorithm. Our algorithm can also be extended to other resources, such as CPUs and networks.

4.3 Resource Allocation for Caching

On each server, the cache resource allocation for each function is managed by the *Cachelet*. When a new function is scheduled, the *Cachelet* launches the function instance and caches it based on the hybrid caching policy. For each cached function instance, the *Cachelet* also monitors their lifetimes and releases (or evicts) them to reclaim the server's cache resources. We now introduce the the cache resource management in *Cachelet*.

4.3.1 Protected/Temporary Memory Partitions. To implement the hybrid caching policy between hotspot functions and non-hotspot functions, we partition the memory resources of each server into two spaces: a *protected space* and a *temporary space* (Figure 4 (a)). The hotspot functions are cached in the protected space, where instances have no lifetime limitation and cannot be evicted by resource contentions (e.g., creating new function instances). The non-hotspot functions are cached in the temporary space. Once an instance has finished execution, it will be kept in memory and released automatically after a keep-alive time length (discussed in § 6.2). The idle instances in the *temporary space* can be evicted by force when there is no sufficient resources to launch new function instances.

Note that we do not set a threshold to limit the memory capacity in protected space, and the hotspot function may run out the server's cache resources in extreme cases. Fortunately, serverless workloads are commonly variable, and **Flame** can dynamically tune the cache resources usage between hotspot and non-hotspot functions (see § 6.3), and the extreme cases rarely exist.

4.3.2 Lifecycle of Cached Functions. Figure 4(b) shows the lifecycle of the cached function instances on each server. The cache placement switch between the protected space and temporary space can be triggered by the *Cachelet* from several events, e.g., hotspot function list update and non-hotspot function eviction. We list these *Cachelet* cache operations in the following:

Hotspot Function List Update: After the *CacheManager* updates the global hotspot function list, it notifies the *Cachelet* to synchronize and check the cached instances on each server. On each server, if a function no longer belongs to the set of hotspot functions, all of its cached instances are moved from the *protected space* to the *temporary space*. Similarly, if a function is newly added to the list, its available instances cached on the server are moved from the *temporary space* to the *protected space*.

Non-Hotspot Function Release/Eviction: For each non-hotspot function, the *Cachelet* records its keep-alive time setting (e.g., 300 s) and periodically checks whether its instance's idle time exceeds the lifetime limitation. The function's maximum keep-alive time is determined by the *CacheManager*, and the instance idle time is calculated by recording its last active timestamp, which is refreshed after each invocation. If a non-hotspot function instance needs to be released or evicted, the *Cachelet* removes it from the temporary space and invokes the serverless platform to delete it.

Cache Redundancy Reclamation: On each server, hotspot functions are cached in the *protected space* until the hotspot function list is updated. Due to the one-to-one mapping design between requests and instances in today's serverless platforms, a large number of hotspot function instances will be created in a short period of time when workload burst occurs [34]. This can result in considerable cache resource waste since newly launched instances will not be released before the next hotspot function list update. Therefore, it is necessary to shrink the cached instances of hotspot functions after workload bursts.

However, determining the number of cached instances and their settings in each server at any time brings challenges for the global *CacheManager*, especially when it works in a high concurrency scenario. Therefore, we adopt a more practical approach, i.e., offloading the cache reclamation control to the underlying *Cachelet* to reduce the decision complexity. On each server, the *Cachelet* monitors the reuse rate of cached functions and releases over-provisioned instances through an adaptive reclamation algorithm. From a perceptive of centralized system design, this hinders the *CacheManager* from making optimal cache decisions but helps to reduce the decision overhead. Our experimental results show that it brings only 14% of performance loss compared to the optimal cache decisions (discussed in § 6.6).

5 IMPLEMENTATION

Flame is implemented on OpenFaaS [22], an event-driven serverless computing platform on top of Kubernetes, with approximately 4,000 lines of Golang. We introduce the keep-alive policy in OpenFaaS to replace the default capacity-based scaling mechanism and add new components (e.g., a global *CacheManager* and *Cachelet*) for hotspot function detection and caching. This modification mainly refers to *gateway*, *faas-netes* and *alert-manager*, and we reuse many of the existing modules, such as authority certification, security

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	256GB
Processor BaseFreq.	2.50 GHz	Operating System	Ubuntu 16.04
CPU Threads	64 (32 physical cores)	SSD Capacity	960GB
Memory Bandwidth	20 GB/s	Network bandwidth	10 Gbps

checking and NAT streaming to minimize the development cost. Additionally, we develop approximately 7,000 lines of code (Java and Linux Shells) for system simulation and testing.

Simulator. We develop a simulator to explore the preferred system configuration in our caching policy and validating its efficiency on a large-scale testbed. The simulator is written in Java with approximately 12,000 lines of code. It implements different caching policies to help us find the optimal parameter setting in **Flame**.

6 EVALUATION

6.1 Setup & Methodology

Testbed: We evaluate our caching system on an 8-server local cluster. Each server is configured with 256 GB RAM and 32-core Intel Xeon Silver 2.50 GHz CPUs. The server runs the modified OpenFaaS (i.e., **Flame**) and Ubuntu 16.04. Table 1 summarizes the cluster's configuration. All servers are connected via 10 Gbps, full-bisection bandwidth Ethernet. Additionally, we use the simulator to find the optimal parameter setting of **Flame**'s caching policy, which takes the same workloads and server configurations in the real-world testbed. Large-scale experiments are also conducted from simulation.

Workload: We use Azure's function traces [27] to generate production-like workload arrival patterns in our evaluation. The traces include the function names, function memory usage, request arrival timestamps, execution time and startup time from more than 50,000 functions over two weeks. We use smaller traces to match our testbed. To this end, We divide these functions into four groups based on the number of invocations, and randomly select 96 functions from each group to form a sample. As shown in Table 2, we build a total of 8 independent workload traces. Each trace set includes the invocation records from 384 functions across one week. These traces exhibit diverse workload behavior in terms of request arrival rate, hot function distribution, and memory consumption.

Benchmarks: Since the Azure's production traces do not contain the function code, we use real-world serverless applications from the ServerlessBench [35] and FunctionBench [13] benchmark suites to evaluate **Flame**'s cache efficiency. The benchmark pool consists of 22 serverless applications, such as data analytics, machine learning and scientific computing, etc. By configuring different memory sizes and input parameters for these applications, we expand the size of the benchmark pool to a total of 384 functions. These benchmarks also present a wide range of function characteristics, including the application type, execution time and coldstart overhead (see Table 5 in Appendix). We generate requests towards them following the request arrival timestamps in our 8 workload traces. For each function in these traces, we find the nearest match of a benchmark function from our benchmark pool based on its memory allocation and execution time to represent the corresponding function behavior. The function's name, memory size, execution time and startup time in these traces are replaced with the actual values of our benchmarks.

Table 2: Workloads used in Flame's testbed.

Workload Name	# of Func.	# of Requests	Avg. Reqs/s	(Top-N func.: >90% total invocs.) # of Functions	Mem Usage perc.
Trace A	384	3,111,827	36	14 (3.64%)	40.57%
Trace B	384	11,363,701	132	4 (1.04%)	15.87%
Trace C	384	12,411,923	144	3 (0.78%)	69.09%
Trace D	384	24,774,632	287	2 (0.52%)	46.81%
Trace E	384	3,462,726	40	12 (3.12%)	68.77%
Trace F	384	2,735,329	32	16 (4.16%)	84.89%
Trace G	384	3,665,540	42	13 (3.38%)	12.35%
Trace H	384	2,183,501	25	17 (4.42%)	30.03%

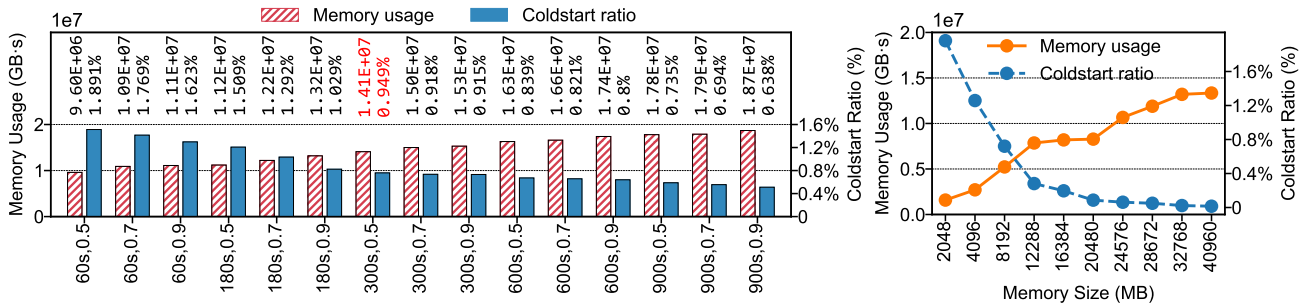
Comparison systems: We compare **Flame** with four existing cache-based methods: the TTL-based keep-alive, FaasCache, CH-RLU and Icebreaker.

- *TTL-based* [29]: Existing serverless platforms, such as AWS Lambda and OpenWhisk, use a TTL-based keep-alive policy to cache functions (e.g., 15 min) after each invocation. In our evaluation, we use the original OpenWhisk as a baseline.
- *FaasCache* [8] (**ASPLOS's 21**): FaasCache takes into account the heterogeneity of functions including factors such as popularity, memory consumption, and coldstart overhead when caching functions in serverless platforms. It employs a priority-based greedy caching policy to determine which functions should be kept alive in each server.
- *CH-RLU* [9] (**HPDC's 22**): CH-RLU aims to reduce the resource contentions in serverless cluster by trading off the cache locality and server load. It proposes an enhanced hashing-based load-balancing policy to schedule requests and cache functions in serverless cluster.
- *Icebreaker* [26] (**ASPLOS's 22**): Icebreaker focuses on both function performance and keep-alive cost in heterogeneous serverless clusters. It determines to cache functions on a high-end machine, or a low-end machine, or not cache it based on its coldstart ratio, resource over-provisioning and server affinity. In our homogeneous testbed, we use Icebreaker's controller to determine whether to cache a function or not.

Metrics: We consider three evaluation metrics: the coldstart ratio (i.e., the proportion of coldstart invocations across all requests in serverless cluster); the function latency (i.e., the duration between request arrivals in the *getaway* and invocation completion); and the overall memory consumption for caching functions (i.e., the memory usage for keeping functions alive and executing requests).

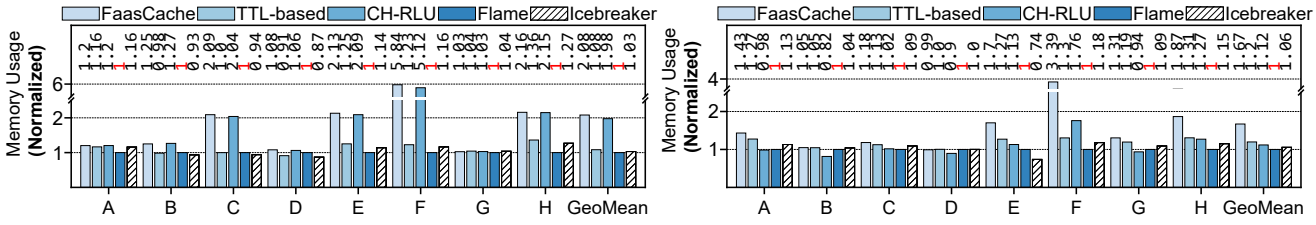
6.2 Parameterizing Flame

Both the region size and keep-alive time setting can affect **Flame**'s cache efficiency. A larger region size brings fewer coldstart invocations for hotspot functions but requires a larger memory size to cache them, which may also preempt the server resources and amplify the second-class citizen effect. A shorter keep-alive time setting enables more elastic caching resource allocation in each server but may also result in a higher cache miss ratio for the non-hotspot functions and increase the scheduling pressure. To find the optimal parameter sittings, we measure the coldstart invocation ratio and cache resource usage from a large number of simulations



(a) Parameter setting (b) Sensitivity analysis

Figure 5: (a) Cache efficiency of Flame under different region size and keep-alive time settings; (b) Effectiveness of Flame's Eden-setting.



(a) Comparison under the hashing-based load-balancing (b) Comparison under the round-robin-based load-balancing

Figure 6: Comparison of the memory usage under the (a) hashing-based and (b) round-robin-based load-balancing rules.

under different $r \times t$ combinations, where t represents the keep-alive time. We do these experiments considering three region size settings (0.5, 0.7 and 0.9) with a discrete keep-alive time settings from 60 s to 900 s.

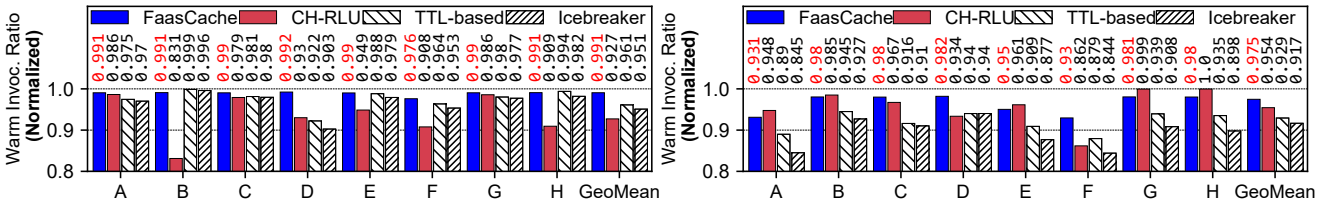
Figure 5(a) presents the results of the study across 15 different $r \times t$ groups under the hashing-based load-balancing rule. Note that the memory cost and coldstart ratio in each group are the geometric means of the results across 8 different workloads. From this characterization, we can see that the geometric mean of the coldstart ratio reaches less than 1% with modest memory consumption under 0.5 of region size and 300 s of keep-alive time setting, which we call the "*Eden-setting*". We find the optimal parameter settings of **Flame** under round-robin load-balancing rule (not shown) from the similar way. Figure 5(b) further presents the cache efficiency of **Flame** with the Eden-setting. We can see that the coldstart ratio decreases quickly as the allocated cache resources increase and finally converges after a knee-point. This demonstrates the effectiveness of **Flame**'s cache management. Thus, we use the Eden-setting (i.e., $r = 0.5$, $t = 300$ s) to evaluate **Flame**'s performance in the following experiments.

6.3 Overall Performance

We evaluate the overall performance of **Flame** using a real 8-server production cluster. The evaluation results of five different cache methods (FaasCache, TTL-based keep-alive, **Flame**, CH-RLU, and Icebreaker) under 8 different workload traces are shown in Figure 6. We mainly focus on comparing the overall memory usage and coldstart ratio of each method, which are normalized to the results in **Flame**.

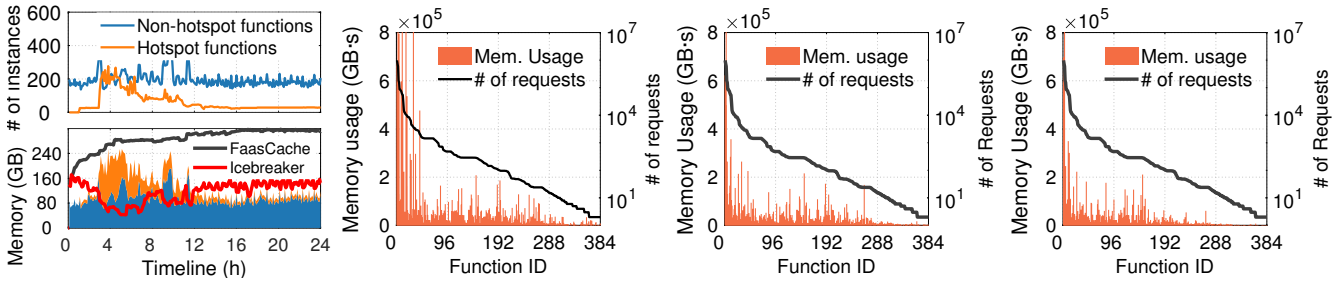
High efficiency: Flame can reduce the cache resource usage by 26%-54% on average and reduce the coldstart ratio by more than 7× in serverless cluster. Figure 6(a) presents the normalized memory consumption of different caching methods under the hashing-based load-balancing rule. The results indicate that **Flame** reduces the overall memory consumption by 36% and 33% on average compared to FaasCache and CH-RLU (80% on maximum in trace *F* where most of the cache resource usage comes from hotspot functions), respectively. Figure 7(a) shows the corresponding warm invocation ratios, which are normalized to the results in **Flame**. Note that although TTL-based keep-alive method consumes only half of the cache resources than FaasCache, it leads to approximately 1.3 × higher coldstart ratio. CH-RLU uses a priority-based caching policy that is similar to that of FaasCache in each server, but its load-aware request distribution policy can violate the cache locality, and thus cause a high coldstart ratio (16.8% at maximum in trace *B*). Icebreaker reduces the overall cache resource usage by decreasing the number of cached functions and achieves a low keep-alive cost. However, the workload fluctuations and function behavior changes make it challenging to make the best decisions, leading to approximately 2.7 × higher coldstart ratio than **Flame**. In comparison, **Flame** nearly achieves the best function performance across 8 workload traces, while the FaasCache, CH-RLU, TTL-based keep-alive, and Icebreaker can only reach 99.1%, 92.7%, 96.1%, and 95.1% of warm invocation ratios than **Flame**, respectively.

The centralized cache controller in **Flame** is designed to be generic and workload agnostic, which makes it achieve a better performance under different workload distributions in serverless cluster. The evaluation results of **Flame** in an 8-server cluster under the round-robin-based load-balancing rule are shown in Figures 6(b) and 7(b). Unlike the hashing-based load-balancing rule, the round-robin-based load-balancing spreads requests across multiple servers even when the invocations come from the same function.



(a) Comparison under the hashing-based load-balancing (b) Comparison under the round-robin-based load-balancing

Figure 7: Comparison of the warm invocation ratio under the (a) hashing-based and (b) round-robin-based load-balancing rules. These results are normalized to Flame, which is set to 1.



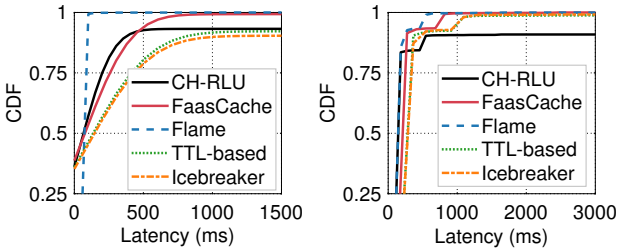
(a) Cache resource allocation (b) FaasCache (c) Flame (d) Icebreaker

Figure 8: (a) The number of cached function instances in Flame's runtime process and the runtime cache resource usage in FaasCache, Flame and Icebreaker; Distribution of the function memory usage in (b) FaasCache, (c) Icebreaker and (d) Flame. The functions are sorted by the number of invocations in descending order.

This can help balance workloads across different servers but violates the cache locality. As a result, all of the five compared cache methods consume more cache resources (approximately 2× memory usage increase on average) to achieve a low coldstart ratio. However, **Flame** overcomes these challenges and achieves better performance. When compared to FaasCache, it reduces the cache resource usage by approximately 40% (71% on maximum) with a lower coldstart ratio. Although **Flame** reduces only 10% of cache resource usage compared to TTL-based keep-alive and Icebreaker, the latter two methods have higher coldstart ratios since the spread workload reduces the function reuse rate in their caching policies. Overall, **Flame** improves the coldstart ratios by nearly 2.9× and 2.7× on average when compared with TTL-based keep-alive and Icebreaker, respectively.

Flexible cache allocation: Flame can efficiently tune the number of cached hotspot functions and their cache resource provision with workload changes. **Flame**'s hybrid caching policy enables it to flexibly allocate the cached resources for both hotspot functions and non-hotspot functions. Its adaptive cache reclamation mechanism in the *Cachelet* can also help to reduce resource over-provisioning, thus improving cache efficiency. Figure 8(a) shows the number of cached instances and their memory usage in **Flame**'s runtime control. We can observe that **Flame** adjusts cache decisions dynamically according to workload changes. During a workload burst, the exponentially decaying algorithm can detect hotspot functions and prevent oscillations in caching decisions. After the workload spike (between 2 h and 4 h), **Flame** automatically reclaims the over-provisioned function instances to reduce cache resource usage. In comparison, FaasCache caches all used functions as long as the cache resources are sufficient, which makes it quickly exhaust the server's memory. Icebreaker uses an FFT-based workload predictor to predict request arrival rate and make cache decisions. However, sudden workload spikes make them difficult to capture the short-term changes in hotspot functions, thus leading to an underestimation of cache resource allocation. Although Icebreaker consumes fewer cache resources than FaasCache, it also results in a high coldstart rate.

Less redundancy: Flame can reduce much of the cache redundancy for the cached functions in serverless cluster. To illustrate the efficiency of cache decisions in **Flame**, we take trace *E* as an example to show the memory consumption breakdown for each function, which includes the memory usage per function for caching instances and executing user requests. Figure 8 shows the details, where the functions are sorted by the number of invocations in descending order. In this example, FaasCache takes 25,188 TB·s of memory to cache all the functions since its local cache control leads to much cache redundancy across the servers. Icebreaker does not distinguish the hotspot functions but reduce the overall caching time for functions, which makes it consumes only 13,463 TB·s of memory, with nearly 50% of cache resource saving per function than FaasCache. However, less resource consumption does not means lower coldstart ratio. **Flame**'s hybrid caching policy improves the memory cost considerably for most of the functions, with only 11,803 TB·s of memory consumption. We could also see that approximately 82% of the memory savings (13,385 TB·s) come from the top-N functions that generate more than 90% of the total requests, which demonstrates the effectiveness of **Flame**'s hotspot-aware cache scheduling and reclamation.

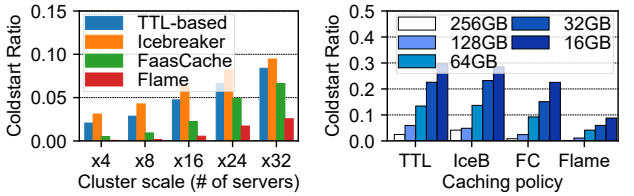


(a) Hashing-based load-balancing (b) Round-robin-based load-balancing

Figure 9: Function latency distributions of five caching methods under the (a) hashing-based and (b) round-robin-based load-balancing rules.

Fewer hotspot contentions: `Flame`'s centralized caching decisions can mitigate hotspot contentions in serverless cluster, and avoid intra-server performance bottlenecks. The workload skewness of hotspot functions can lead to cache resource contentions and result in frequent scheduling failures or request discards. Table 3 shows a breakdown of function coldstart ratio and request drop ratio for the five compared methods. The drop ratio is the fraction of requests that cannot be processed due to insufficient cache resources in the cluster. We see that **`Flame`**'s hotspot-aware cache scheduling greatly reduces cache contentions in serverless clusters, resulting in only 0.86% of coldstart ratios and 0.02% of drop ratios. Among the rest of four methods, FaasCache has the smallest drop ratio (0.77%) since its priority-based caching policy can evict unpopular functions to release cache resources. When compared to the TTL-based keep-alive method (1.63% drop ratio), CH-RLU's load balancing-based cache scheduling policy reduces resource contentions (1.41% drop ratio) but has a higher coldstart ratio (5.34%) due to its lower cache reuse rate. Icebreaker's caching algorithm reduces the number of cached functions in the cluster, which helps reduce the request drop ratio by approximately 20% when compared to the TTL-based keep-alive method. However, it also results in more coldstart invocations (6.11% of coldstart ratio). In summary, **`Flame`**'s centralized caching decisions reduce 38.7×-81.4× of hotspot contentions compared to the existing local cache control methods, which makes it achieve 2.9×-8.4× of performance improvements over them.

Low request latency: `Flame` can reduce the 99th percentile latency more than 10× by mitigating the coldstart overhead. Figure 9 depicts the CDF of request latency for different caching methods. For each method, we aggregate all of the function invocations under the 8 workload traces to compare their latency distributions. We can see that **`Flame`** has the lowest function latency under both the hashing-based and round-robin-based load-balancing rules. CH-RLU has a better function performance than TTL-based keep-alive and Icebreaker under a consolidated workload distribution (i.e., consistent hashing in Figure 9(a)). However, its caching efficiency decreases when using a load-balanced request dispatcher (round-robin in Figure 9(b)). FaasCache's priority-based caching policy always gives lower priority to functions with longer startup times, resulting in longer tail latency than CH-RLU. However, its smaller coldstart ratio enables it to have lower function latency than TTL-based keep-alive and Icebreaker. Icebreaker and TTL-based keep-alive method have similar latency distributions. Both of them adopt a time-length-fixed caching policy, which leads to high coldstart ratios especially when in unpredictable workload patterns. Therefore, they have the worst function performance (a more than 10× increase in *P*99 latency than **`Flame`**) across the compared five methods.



(a) Performance evaluation under different cluster scales (b) Performance evaluation under different server memory sizes

Figure 10: Performance evaluation of Flame by changing (a) server scale and (b) server memory size. TTL: TTL-based; IceB: Icebreaker; FC: FaasCache.

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×)

6.4 Sensitivity Analysis

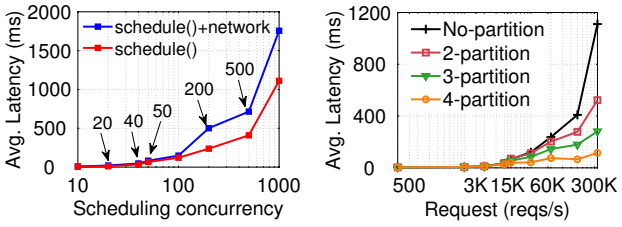
We also evaluate the sensitivity of **`Flame`**'s cache efficiency when the cluster scale of server memory capacity changes.

Better scalability: `Flame` can achieve a high cache efficiency in larger-scale clusters or clusters with less server memory. To evaluate the cache efficiency of **`Flame`** under different cluster scales, we change the number of servers in the testbed while keeping the total memory capacity constant. Figure 10(a) shows a comparison of the coldstart ratio for cluster scales of 4, 8, 16, 24, and 32 servers. We find that the coldstart ratio in the TTL-based keep-alive method, FaasCache, and Icebreaker increase dramatically as the cluster scale increased. This is due to their local cache control causing more hotspot contentions and cache redundancy in small-sized servers. As a result, the overall function performance in these methods becomes even worse. However, **`Flame`** is able to maintain a lower coldstart ratio thanks to its centralized cache management. Figure 10(b) shows the evaluation results when we only scale down the server's memory capacity. We find that FaasCache, TTL-based keep-alive, and Icebreaker all suffer from high coldstart ratios when the server memory is reduced. Although the coldstart ratio of **`Flame`** increases to nearly 8% when the server memory is less than 128 GB, it still achieves the lowest coldstart ratio due to its effective caching policy design.

6.5 Overhead

We further analyze the system overhead of **`Flame`**'s centralized cache controller.

System overhead: `Flame` incurs low system overhead, which enables it to be easily extensible for a large-scale practical deployment. **`Flame`** mainly has three sources of overhead at its



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

Figure 11: Evaluation of Flame's scheduling overhead.

runtime cache control: (1) hotspot function detection (HFD), (2) instance lifetime management (ILM) in each *Cachelet* and (3) cache scheduling (CS) in *CacheManager*. We provide a detailed analysis of each:

- **HFD** is a single module that runs as a daemon process inside *CacheManager*. Its function invocation counter and hot-score calculation are run per function in a parallel manner, so the overhead of these actions does not increase much with the number of deployment functions or the number of servers. Although the detection time of top-N hotspot functions is linearly correlated with the number of functions, it takes only 1.5 ms to deal with 384 functions in our measurements.
- **ILM** is a daemon process launched in each Cachelet. The instance lifetime monitor in ILM also runs in parallel for each local function. When the global hotspot function list is updated, synchronizing it to each *Cachelet* incurs negligible overhead (about 800 us). In each server, the *Cachelet* periodically runs two threads: (i) updating instance's keep-alive setting and (ii) moving instances across three memory spaces. The time interval for thread execution is 5 seconds in our system. When the thread is triggered, the *Cachelet* invokes Kubernetes's APIs to create (or delete) function instances, which takes less than 1 ms.
- **CS** is a key module of the centralized cache controller in *CacheManager*. It is invoked only when cache-miss occurs. Thus, even a high volume of concurrent requests may not invoke it many times at low cache-miss ratio. In our experiments, we measure that the CS algorithm incurs an overhead of 2 ms on a 384-function scale deployment. We also present *CacheManager*'s average scheduling time under different scheduling concurrency pressures. As shown in Figure 11(a), the CS generates about 240 ms of decision latency under 200 of the scheduling concurrency (i.e., 200 cache misses per second). Thus, given 1% of a coldstart ratio, **Flame** is able to handle up to 83K concurrent requests per second theoretically ($max\_throughput = \frac{200}{0.24 \times 0.01\%}$).

Additionally, **Flame** adopts a lightweight design and takes only a small amount of resources to run. Our measurement shows that the *CacheManager* takes up 0.03 CPU cores and 175 MB of memory during its runtime, while each *Cachelet* consumes only 0.05 CPU cores and 100 MB of memory. Assuming that **Flame** is deployed in a production cluster with 500 servers, including 8400 CPUs and 17TB memory that serves over 20,000 functions, **Flame** would require 2 CPUs and 8GB of memory to run its *CacheManager*, while all *Cachelet* would consume about 25 CPU cores and 48 GB memory, which is relatively small in a production system.

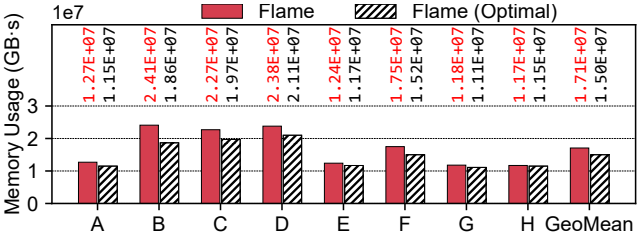


Figure 12: Comparison of memory usage between the native Flame and Flame (Optimal) under <1 of coldstart ratio.

Table 4: Comparison of cache cost (Coldstart ratio <1%).

	TTL	FaasCache	Flame	Icebreaker
Mem usage/req	1132 MB·s	663 MB·s	370 MB·s	988 MB·s
Cache cost/req [$]	1.85×10^{-5}	1.08×10^{-5}	6.08×10^{-6}	1.62×10^{-5}

However, in an extremely large-scale cluster, the centralized controller of **Flame** may encounter scalability issues. To mitigate this problem, higher scheduling parallelism and workload partitioning can be employed. We also present the evaluation results of using workload partitioning technique to reduce *CacheManager*'s decision pressure. As shown in Figure 11(b), the average decision latency for each function in the non-partition design exceeds 1000 ms under a concurrence pressure of 300K reqs/s. When using a 3-partition technique in **Flame**'s front-ends, the average decision latency for each function can be reduced to approximately 200 ms, resulting in more than a 5× performance improvement. Therefore, we recommend using the partitioning technique to address the scalability issues of **Flame** in extremely large-scale serverless scenarios.

6.6 Discussion

We finally discuss the potential cost benefit for serverless provider with **Flame**'s improvement.

Economic benefit: Flame can help cloud providers reduce the resource cost in caching serverless functions by more than 40%. To evaluate the economic benefits of **Flame**, we record the total memory consumption of each caching policy and derive the average memory cost per invocation. We set the price of 1 GB of memory to 0.06048$/hour, following the setting of the AWS Lambda service in the Eastern United States (Ohio) [25]. Table 4 shows that **Flame** reduces the memory cost per request by approximately 44%, 67%, and 62% when compared to FaasCache, TTL-based keep-alive, and Icebreaker, respectively. Considering the scale of our production serverless cluster, which deploys more than 40,000 serverless functions serving 1.3 billion requests per day, the memory cost of caching functions is approximately 8,810,000$ per year. If we use **Flame** to replace the existing keep-alive policy, the memory cost will decrease to 2,880,000$ per year, saving approximately 5,900,000$ every year.

Optimality: Decoupling the cache reclamation control to *Cachelet* enables Flame to reduce its system overhead, while this only causes approximately 20% of performance loss compared to the optimal system design. As previously demonstrated, offloading partial caching management controls from *CacheManager* to the underlying *Cachelets* reduces decision-making overhead in concurrency scenarios, while this may also lead to a certain

amount of throughput loss, we also compare **Flame** with Flame (Optimal) to better understand its roofline performance. Flame (Optimal) is a monolithic system design that integrates the *Cachelet* functionalities (e.g., cache reclamation) inside the *CacheManager*, which enables it to make optimal caching decisions from a centralized cache control. Figure 12 depicts their performance difference. We see that **Flame** can reach nearly 86% of the performance on average from Flame (Optimal). The experimental results indicate that the *CacheManager* contributes most of the cache resource saving in **Flame**'s cache management, and this also demonstrates the effectiveness of the existing **Flame** architecture design.

7 RELATED WORK

Startup acceleration: To mitigate the coldstart overhead, various techniques have been proposed to the sandbox initialization time or application startup time. At the runtime layer, Mohan et al. [20] preallocate virtual network interfaces for newly launched containers to reduce the startup time. SOCK [21] caches commonly used Python libraries to reduce function initialization time. Catalyzer [6] and REAP [28] use a new checkpoint and snapshot mechanism to reduce the startup time of their gvisor-based [12] security containers. At the application layer, LambdaLite [31] proposes an application-level performance optimization approach for reducing the code and dependency size for Python applications in serverless platforms. Additionally, microVMs [1] and unikernels [19] are being used in several serverless platforms to reduce the coldstart overhead. However, as shown in previous work [8], these fast startup methods do not eliminate the coldstart problem since the application initialized overhead still remains.

Our approach aims to "avoid" coldstart operations by retaining the entire execution environment, which can provide near-zero coldstart latency for many short-lived functions. Moreover, our cache-based approach is orthogonal to these techniques.

Function caching: Function caching has also recently gained attention. Prior work [10, 17, 24] focuses on static keep-alive policies or simple "warm container pools", these mechanisms are easy to implement but suffer from a high keep-alive cost. Several studies predict workload behaviors to help make cache decisions, thus improving cache reuse rate or reducing keep-alive cost. For example, Shahrda et al. [27] use a hybrid histogram policy (HHP) to predict request arrival intervals and dynamically cache functions through a "prefetching" approach. IceBreaker [26] uses a time-series prediction approach to determine function cache decisions in a heterogeneous serverless system, thus reducing the function's keep-alive cost.

Although these prediction-based caching methods can help reduce the keep-alive cost, their effectiveness may be affected by workload variations. Our approach adopts a reactive caching policy, like FaasCache [8], which is designed to be general and workload-agnostic, and it can be combined with prediction-based caching methods to further improve system-wide performance. Moreover, in contrast to these existing caching policies, our approach focuses on hot-spot contention and cache redundancy issues in local cache control, and is the first to explore and demonstrate the effectiveness of centralized cache control.

Instance sharing or reusing: SAND [2] warms subsequent function invocations within an application by reusing the same sandbox environment. Faastlane [15] colocates functions in the same container with threads so that it can bypass the subsequent coldstart invocations in child functions. Fifer [11] uses a function-aware container scaling technique and request batching mechanism to bin packing jobs to fewer containers, thus reducing coldstarts in function-chains. Pagurus [18] proposes an inter-function container sharing technique to reduce the function keep-alive cost in prewarm-based methods. It alleviates the coldstart by re-purposing a warm but idle container from another function. These solutions improve the instance reuse rate by modifying the function system architecture or sandbox operation mechanisms. Our approach can also be combined with these techniques to further improve instance reuse rate.

8 CONCLUSION

In this paper, we focus on the function caching to mitigate coldstart problem in serverless computing and find the existing local cache control suffer from hotspot contention and cache redundancy issues, which can result in high cache miss ratio and cache resource usage. To address this problem, we propose a centralized caching system that coordinates caching across servers. Our preliminary exploration shows that such a system can achieve better cache efficiency than local cache control. We believe that this approach could be particularly useful in addressing the coldstart problem. In the future, we plan to explore multi-layer caching mechanisms by separating the sandbox, language runtime, and user code in serverless functions to improve system cache efficiency even further.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China No.2022YFB4500702; project ZR2022LZH018 supported by Shandong Provincial Natural Science Foundation; the National Natural Science Foundation of China under grant 62141218, 62372322; the open project of Zhejiang Lab (2021DA0 AM01/003) and the CCF-Huawei Populus euphratica Innovation Research Funding CCF-HuaweiSE2022006.

A ARTIFACT

A.1 Abstract

Flame is a technique that reduces the coldstart rate and function caching cost of serverless functions, which is designed to deploy in serverless platforms by cloud providers. Flame's basic idea is to identify the hotspot functions in serverless workload scenarios and manage their cache resources allocation and scheduling from a centralized cache controller. Flame consists of two major components: (1) a cluster-level CacheManager, and (2) a Cachelet that runs in each node. The CacheManager identifies the cluster-level hotspot functions and makes cache decisions from a global view of serverless cluster. The Cachelet receives the cache decisions from CacheManager and manages the cached function instances in each node. Our artifact includes a local Flame implementation for function caching written in Java. The artifact packages the scripts for setting up and launching Flame. It also contains the data obtained in our experimentation. The artifact is available at the following link: https://doi.org/10.5281/zenodo.8398112.

It includes the following:

- The framework and source code of Flame's two-layer cache controller implementation.
- Scripts and instructions to set up the benchmarks and serverless invocation trace.
- Scripts to set up the control system of Flame which runs from a local environment.
- Function caching cost data of executing Flame and other competing techniques.
- Coldstart ratio of executing Flame and other competing techniques.
- Data of executing Flame and other competing techniques for different request arrival patterns in workload A-H.

Flame is evaluated based on the Microsoft Azure function invocation trace, and it can be configured with different server cache capacity for each workload arrival pattern from the trace.

A.2 Artifact check-list (meta-information)

- **Algorithm:** A hotspot detection algorithm and a hotspot-aware cache scheduling algorithm.
- **Program:** Benchmarks from SeverlessBench and FunctionBench were used to evaluate Flame.
- **Data set:** The data set consists of 8 different Microsoft Azure serverless function traces. Each trace provides 2 weeks of serverless invocation data from 384 functions. From this trace, per function invocation arrival timestamp, per function instance memory usage, per function startup time and invocation time are obtained.
- **Compilation:** Java 1.8.0_333.
- **Run-time environment:** Ubuntu 16.04.
- **Hardware:** Intel x86 servers connected via 10 Gbps, full-bisection bandwidth Ethernet.
- **Metrics:** Coldstart ratio, and cache resource usage are the major metrics of evaluation.
- **Output:** Overall coldstart ratio and keep-alive cost. Along with these, the number of per function cached instances and cache resource usage timeline are also recorded in the system log files.
- **Experiments:** The experiments measure the overall coldstart ratio and cache resource usage per competing techniques. In our evaluation, we have experimented with the 5 different competing techniques under the server memory configurations for all 8 workload traces. The server memory configurations are chosen to maintain an equal capital cost across all of them.
- **How much disk space required (approximately)?:** 10 GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** About 1 hour per workload trace, a total of 10+ hours.
- **Publicly available?:** Yes.
- **Archived (DOI)?:** https://zenodo.org/record/8398112.

A.3 Description

A.3.1 How to access. The source code, scripts, and instructions are available on Zenodo: https://doi.org/10.5281/zenodo.8398112.

A.4 Evaluation

Users need to prepare a local testbed with Linux operating system (e.g., Ubuntu 16.04), and download the source code and scripts from Github to the server. The base workspace is named cacheResearch. The following is the directory structure of the source code, scripts, and instructions:

- **README.md** This file has detailed instructions to conduct experiments.
- **sourceCode** The source code of Flame for CacheManager and Cachelet.
- **cacheTrace** The workload trace files in Flame's evaluation.
- **workspace** The workspace of Flame's evaluation, which contains the compiled binary files of Flame's, the scripts to launch Flame's CacheManager and Cachelet, the scripts to start Flame's evaluations, and the scripts to display the comparison results of Flame.
- **coldstartRate** The evaluation results of Flame's competing techniques.

Installing Flame: After downloading the source code and scripts, users can choose to manually compile the source code of Flame's CacheManager and Cachelet. For easy-to-use, we provide the compiled binary files of Flame's core components, these runnable files (.jar) can be listed as following commands:

```
$ mkdir /home/tank/1_yanan/ && cd /home/tank/1_yanan/
$ git clone http://github.com/Flame/cacheResearch/
$ cd cacheResearch/workspace/
$ ls
...
cacheLoadGen_linux_flameManual.jar
cachelet-linux.jar
cacheManager-linux.jar
result.jar
start-flame.sh
...
```

Launching Flame: Then, using the following commands to launch Flame system and conduct the evaluations:

```
$ source /etc/profile
$ sh start-flame.sh 23040 a rehash 300000 0.5f
..... Evaluation start .....
1000 0.1%
2000 0.2%
...
3110000 99.9%
1921s rehash
..... Evaluation stopped .....
```

The start-flame.sh file is the script to launch an experiment of Flame's cache management. It requires five parameters, including the server memory configuration, the trace type, the load balancing policy, the keep-alive time for non-hotspot functions, and the hotspot region size.

We can change the input parameters to evaluate Flame's effectiveness under different workload traces (a-h). For example,

- sh start-flame.sh 23040 a rehash 300000 0.5
- sh start-flame.sh 50176 b rehash 300000 0.5
- sh start-flame.sh 81920 c rehash 300000 0.5
- ...
- ...
- sh start-flame.sh 38912 h rehash 300000 0.5

When the experiment finishes, the evaluation result of Flame is recorded in directory '.../cacheResearch/coldstartRate/temp/', user

Table 5: Applications in the benchmark pool.

ID	Source	Function	Memory	Pamamter	Execution Time (ms)			Startup Time (ms)		
					min	avg	max	min	avg	max
1	ServerlessBench	paralle-alu	128MB-1536MB	loop_time=[10,100,1000,10000] parallism=[1,2,4]	10	233	999	2103	2133	3586
2	ServerlessBench	key-downloader	128MB-1024MB	file_size=[512,1024,4096]	16	22	32	2171	2402	2749
3	ServerlessBench	extract-image-metadata	128MB-1536MB	input=imagenet_file	37	80	188	2200	2485	2673
4	ServerlessBench	store-image-metadata	128MB-2048MB	input=imagenet_file	174	311	731	2430	2744	3240
5	ServerlessBench	thumbnail	128MB-1792MB	input=imagenet_file	258	475	1127	2660	2973	3370
6	ServerlessBench	tail-bigparam	128MB-10240MB	dataset=[200000,1000000,5000000]	65	638	1774	2312	3047	3972
7	FunctionBench	chameleon	128MB-8192MB	cols=[100,200,500,1000] rows=[100,200,500,1000]	55	1878	7213	2176	4315	9732
8	FunctionBench	float-operation	128MB-1536MB	loop_time=[10000,100000,1000000,10000000]	8	2351	16889	2115	4762	19059
9	FunctionBench	linpack	256MB-8192MB	array_len=[200,1000]	61	2031	12664	2417	4485	15134
10	FunctionBench	matmul	128MB-2048MB	mat_dim=[20,50,100]	4	162	570	2307	2663	3237
11	FunctionBench	pyaes	128MB-2048MB	msg_len=[200,500,1000] iterations=[20,50,100]	877	1437	3412	3120	3943	5861
12	FunctionBench	dd	128MB-1536MB	block_size=[1024,2048,4096] count=[256,512,1024]	45	62	105	2183	2445	2693
13	FunctionBench	giz-compress	128MB-1792MB	file_size=[1024,4096,8192]	63	480	1898	2395	3010	4433
14	FunctionBench	random-diskio	128MB-1792MB	file_size=[1024,4096] block_size=[128,256,512]	103	155	291	2267	2628	2943
15	FunctionBench	image-process	256MB-8192MB	input=imagenet_file	2134	2843	6712	4311	5299	9286
16	FunctionBench	image-rotate	256MB-4096MB	input=imagenet_file	545	750	1709	2743	3172	3978
17	FunctionBench	image-scale	256MB-4096MB	input=imagenet_file	1723	2263	5090	3836	4749	7710
18	FunctionBench	cnn-image-classification	256MB-2560MB	input=imagenet_file	51	203	601	1774	2216	3979
19	FunctionBench	video-process	128MB-2048MB	dpi=[1280x720] video_size=[1024,2560,3042]	2023	5076	14892	4363	7502	17511
20	FunctionBench	lr-prediction	256MB-2048MB	data_size=[2,5,10,20]	72	139	230	1733	3001	6838
21	FunctionBench	face-detection	256MB-2560MB	video_size=[256,512,1024]	569	1150	2801	2189	2868	3901
22	FunctionBench	rnn-generate-char-level	256MB-2560MB	lang=[English,German] input_len=[20,50,80,100]	5	270	1864	1007	1356	3551

should first collect the results of Flame in specific directory using the following commands:

```
$ cd ../coldstartRate/temp/
$ mv *csv Flame/rehash/
```

Comparison Result: We have conducted the evaluation results of four competing techniques (CHRLU, FaasCache, Icebreaker and Keepalive) in directory 'coldstartRate/temp/', each of them is stored in a separated fold which is named by their system names. User can use the following commands to compare the cache efficiency of the competing systems:

```
$ java -jar result.jar a 23040
Evaluation Output:
————————
Keepalive policy:
# of Requests   CacheUsage (MB)  ColdstartRate
xxx   xxx   xxx
FaasCache policy:
# of Requests   CacheUsage (MB)   ColdstartRate
xxx   xxx   xxx
...
Flame policy:
# of Requests   CacheUsage (MB)   ColdstartRate
xxx   xxx   xxx
```

A.5 Related Materials

Available benchmarks: https://github.com/ykiauz/Flame/tree/main/benchmark

B MATERIALS

B.1 Benchmarks

As we cannot directly use the user's real-world business function code from the production system, we collect 22 serverless benchmarks from ServerlessBench and FunctionBench and build a large set of serverless functions to evaluate **Flame**'s cache efficiency. By configuring these benchmark applications with different memory sizes and input parameters, we are able to generate a total of 384 functions with diverse workload behaviors. Table 5 summarizes these functions and their configurations. The table has 22 rows and 11 columns. Each row represents an individual benchmark application in the benchmark pool and the different configurations used in its variants, including their memory size and input parameters. We also show the execution times and start times of the variants, which cover a wide range of latencies.

We use Azure's function traces to generate production-like workload arrival patterns. Since the raw traces do not contain function code, we deploy 384 variants and generate requests towards them following Azure request arrival timestamps. The memory usage and execution time per request are also replaced with the

actual values of the variants. The real benchmarks are available at https://github.com/ykiauz/Flame/tree/main/benchmark.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[3] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 730–746. https://doi.org/10.1145/3492321.3524270

[4] Cliff Click and John Rose. 2002. Fast subtype checking in the HotSpot JVM. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande 2002, Seattle, Washington, USA, November 3-5, 2002*, José E. Moreira, Geoffrey C. Fox, and Vladimir Getov (Eds.). ACM, 96–107. https://doi.org/10.1145/583810.583821

[5] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward Decay: A Practical Time Decay Model for Streaming Systems. In *2009 IEEE 25th International Conference on Data Engineering*. 138–149. https://doi.org/10.1109/ICDE.2009.65

[6] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[7] Jim Esch. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 10–13. https://doi.org/10.1109/JPROC.2014.2374752

[8] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 386–400. https://doi.org/10.1145/3445814.3446757

[9] Alexander Fuerst and Prateek Sharma. 2022. Locality-aware Load-Balancing For Serverless Clusters. In *HPDC '22: The 31st International Symposium on High-Performance Parallel and Distributed Computing, Minneapolis, MN, USA, 27 June 2022 - 1 July 2022*, Jon B. Weissman, Abhishek Chandra, Ada Gavrilovska, and Devesh Tiwari (Eds.). ACM, 227–239. https://doi.org/10.1145/3502181.3531459

[10] Google Cloud Function. [n. d.]. https://cloud.google.com/functions/. Referenced 2022.

[11] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, Dilma Da Silva and Rüdiger Kapitza (Eds.). ACM, 280–295. https://doi.org/10.1145/3423211.3425683

[12] gVisor: Application Kernel for Containers. [n. d.]. https://gvisor.dev/. Referenced 2022.

[13] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama (Eds.). IEEE, 502–504. https://doi.org/10.1109/CLOUD.2019.00091

[14] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[15] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[16] Kubernetes. [n. d.]. http://k8s.io. Referenced 2022.

[17] AWS Lambda. [n. d.]. https://aws.amazon.com/lambda/?nc1=h_ls. Referenced 2022.

[18] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[19] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 461–472. https://doi.org/10.1145/2451116.2451167

[20] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*, Christina Delimitrou and Dan R. K. Ports (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[21] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[22] OpenFaas. [n. d.]. https://docs.openfaas.com/. Referenced 2022.

[23] Docker pause. [n. d.]. https://docs.docker.com/engine/reference/commandline/container_pause/. Referenced 2020.

[24] Apache OpenWhisk: Open Source Serverless Cloud Platform. [n. d.]. https://openwhisk.apache.org/. Referenced 2022.

[25] AWS Lambda Price. [n. d.]. https://aws.amazon.com/cn/lambda/pricing/. Referenced 2022.

[26] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 753–767. https://doi.org/10.1145/3503222.3507750

[27] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[28] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 559–572. https://doi.org/10.1145/3445814.3446714

[29] Keeping Functions Warm. [n. d.]. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html. Referenced 2022.

[30] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2022. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. *CoRR* abs/2203.10225 (2022). arXiv:2203.10225 https://arxiv.org/abs/2203.10225

[31] Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, Xin Jin, and Xuanzhe Liu. 2022. LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing. *CoRR* abs/2207.08175 (2022). https://doi.org/10.48550/arXiv.2207.08175 arXiv:2207.08175

[32] Wenfeng Xia, Peng Zhao, Yonggang Wen, and Haiyong Xie. 2017. A Survey on Data Center Networking (DCN): Infrastructure and Operations. *IEEE Commun. Surv. Tutorials* 19, 1 (2017), 640–656. https://doi.org/10.1109/COMST.2016.2626784

[33] Mahendra Pratap Yadav, Nisha Pal, and Dharmendra Kumar Yadav. 2021. A formal approach for Docker container deployment. *Concurr. Comput. Pract. Exp.* 33, 20 (2021). https://doi.org/10.1002/cpe.6364

[34] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 768–781. https://doi.org/10.1145/3503222.3507709

[35] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 30–44. https://doi.org/10.1145/3419111.3421280

[36] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, predicting and scheduling serverless workloads under partial interference. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 22:1–22:15. https://doi.org/10.1145/3458817.3476215