*electronics*

MDPI

*Article*

# µFuncCache: A User-Side Lightweight Cache System for Public FaaS Platforms

**Bao Li, Zhe Li, Jun Luo, Yusong Tan and Pingjing Lu ***

School of Computer, National University of Defense Technology, Changsha 410073, China
* Correspondence: pingjinglu@nudt.edu.cn

**Abstract:** Building cloud-native applications based on public "Function as a Service" (FaaS) platforms has become an attractive way to improve business roll-out speed and elasticity, as well as reduce cloud usage costs. Applications based on FaaS are usually designed with multiple different cloud functions based on their functionality, and there will be call relationships between cloud functions. At the same time, each cloud function may depend on other services provided by cloud providers, such as object storage services, database services, and file storage services. When there is a call relationship between cloud functions, or between cloud functions and other services, a certain delay will occur, and the delay will increase with the length of the call chain, thereby affecting the quality of application services and user experience. Therefore, we introduce µFuncCache, a user-side lightweight caching mechanism to speed up data access for public FaaS services, fully utilizing the container delay destruction mechanism and over-booked memory commonly found in public FaaS platforms, to reduce function call latency without the need to perceive and modify the internal architecture of public clouds. Experiments in different application scenarios have shown that µFuncCache can effectively improve the performance of FaaS applications by consuming only a small amount of additional resources, while achieving a maximum reduction of 97% in latency.

**Keywords:** function as a service (FaaS); public cloud function; user-side; lightweight cache



**Citation:** Li, B.; Li, Z.; Luo, J.; Tan, Y.; Lu, P. µFuncCache: A User-Side Lightweight Cache System for Public FaaS Platforms. *Electronics* **2023**, *12*, 2649. https://doi.org/10.3390/electronics12122649

Academic Editor: Bahman Javadi

Received: 12 May 2023
Revised: 8 June 2023
Accepted: 12 June 2023
Published: 13 June 2023



## 1. Introduction

Cloud computing aggregates hardware and software in a datacenter into a large pool and offers users on-demand IT resources with pay-as-you-go pricing [1]. However, the early cloud computing phase, based on virtual machine (VM) provision (known as Infrastructure as a Service, IaaS), only migrates workloads to the cloud, and users are still responsible for the management, configuration, and scaling of VMs. At the same time, in order to deal with sudden surges, users often prefer to rent more resources than usual, resulting in significant resource and budget waste [2].

Function as a Service (FaaS) is a new paradigm of cloud computing that has emerged in recent years, currently as an embodiment of serverless computing [3–6]. Compared with traditional cloud computing, users on FaaS platforms no longer need to care about the maintenance and management of VMs, especially the complex, time-consuming, and error-prone configurations involved in distributed applications. They only need to upload the code through the interfaces provided by the FaaS platform, and set parameters, such as expected usage resources and runtime, to run it as a completely distributed and scalable application. At the same time, users no longer need to purchase redundant resources to cope with rare high concurrency scenarios, and they are only charged based on the duration of a single task execution and the usage of resources. Since the release of the Lambda [7] cloud function platform by Amazon Web Services (AWS) in 2014, many public cloud providers, such as Azure, Google Cloud, and Alibaba Cloud have provided similar services [8–10], and several open-source platforms (e.g., OpenWhisk [11], OpenFaaS [12], and Knative [13]) were presented. Each cloud function on FaaS platforms is a short-lived

(e.g., the maximum run time of functions in AWS Lambda is 15 min [3]), stateless task unit encapsulated in a container or other kinds of sandbox, including the code package to be run, a series of parameter settings, and related configuration files. Every cloud function can independently execute its own business logic and interact with other cloud functions through function calls. Usually, a function instance will execute only when invoked by different event triggers (e.g., HTTP request, timer) and sleep immediately after handing the event [3].

In recent years, a wide range of applications have been adapted to FaaS architecture, including machine learning [14,15], big data analytics [16,17], Internet of Things [18,19], scientific computing [20,21], etc. In order to achieve high elasticity, applications developed and deployed on FaaS platforms usually consist of multiple different cloud functions, thus communications in the form of function call are required. Meanwhile, because of the "data-shipping" architecture [22] incurred by the stateless nature and inability of the addressable network, in order to share state information or ensure data persistence, FaaS applications will also need to access other services in the cloud ecosystem (known as Backend as a Service, BaaS [3,5]), such as database services for important user data and object storage services for large files. Therefore, there are two types of call relationships: calls between cloud functions, and calls between cloud functions and other cloud services. Regardless of the type of call relationships, there will be a certain amount of latency. Although the latency of a single call between cloud functions is very low, once multiple calls are made on a single request to form a long call chain, the latency will accumulate to a large value. In addition, if the size of the data transmitted is too large, latency would also significantly increase. As shown in Figure 1, when a user request reaches Function 1, if the call chain between Function 1 and Function n is too long, the response delay may not only be affected by the network transmission time, but also by the processing time of cloud functions with a heavy workload. In addition, when cloud functions request other cloud services, such as database services or object storage services, the response time will also depend on the size of the data.

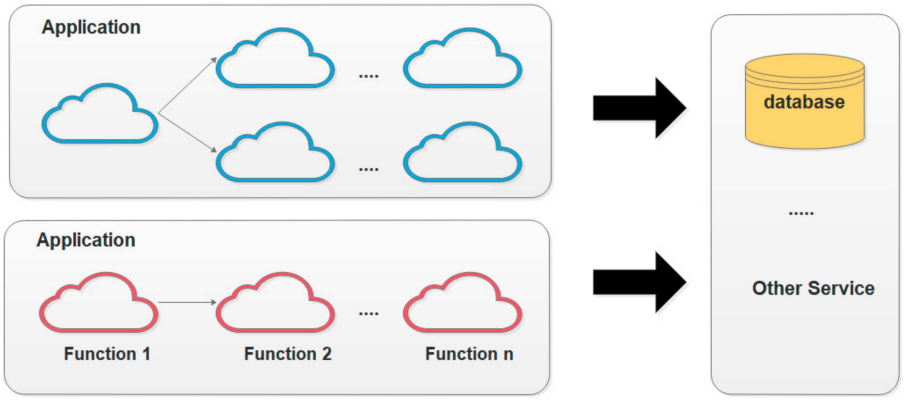


**Figure 1.** Function call chain in FaaS applications.

To validate the above inference, we conducted a test on the Alibaba Cloud function computing platform Function Compute (FC) [10]. The test includes calls between cloud functions and calls from cloud functions to other cloud services. We created a cloud function to perform a read and write operation on Alibaba Cloud Object Storage Service (OSS) [23], and then continuously added new functions to form a chain. All cloud functions in the chain do not execute any complex task, and only send requests to the next one or OSS in the end. The response time of the request is recorded every time. The maximum memory allowed for a single function is set to 512 MB, and the maximum running time allowed is set to 10 s. The programming language is Java with JDK 1.10.

Before each test, we called the function chain several times to ensure that all cloud functions have live instances, in order to avoid the impact of a cold start (i.e., set-up time required that is invoked when downloading the code, starting the instance, and initializing

the runtime, and so on, for the first time) [3,6], which has been studied extensively with some work also focusing on the optimization for the chain structure [24,25]. The call chain length was set from 1 to 4 and, for each run, we conducted 20 tests and calculated the average time. The results are shown in Figure 2. As can be seen, as the length of the call chain increased, the response time continuously increased, and both the read and write curves showed positive linear relationships. Meanwhile, it can be found that the delay of a single call was actually low. However, as the length increased, when the call chain length reached three, the latency was even around 180 ms. On the other hand, since all cloud functions do not require long local processing time, it can be concluded that the network latency between functions in Alibaba Cloud FC remained almost stable during the tests.

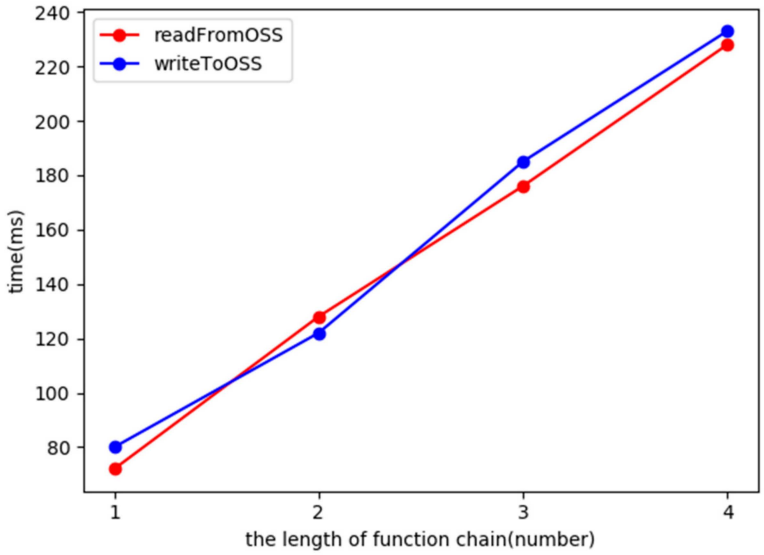


**Figure 2.** Response time of function chains with different lengths in Alibaba Cloud FC.

However, since one of the main design goals of the public FaaS service is to abstract and hide as many resource management details as possible, users are generally unable to perceive and modify the internal architecture of FaaS platforms to reduce the aforementioned latency. Further limited by this reason, existing research work on this issue has to validate the proposed optimization method on open-source FaaS platforms such as OpenWhisk [26] or additional public cloud resources need to be resorted to build separate storage or caching systems [27,28].

Therefore, this paper proposes, for the first time, the optimization of the delay of cloud functions from the user side, based on two typical features of public FaaS platforms. On the one hand, public FaaS platforms usually adopt reuse mechanisms to reduce the number of cold starts. Under the strategy of container reuse, when a cloud function instance handles a request, it will not be immediately destroyed, but will continue to survive for a certain period of time. For example, function instances will survive for 5 min and 20 min in AWS and Azure, respectively [29,30]. When a function instance is within its lifetime and receives new requests, it will be reused to eliminate cold start delays. The remaining survival time of the container instance will be refreshed with the arrival of new requests. On the other hand, function instances in public cloud functions generally have redundant memory. Based on the analysis of current public FaaS platforms, it is found that users usually request excessive memory for function instances. For example, the average memory usage of function instances in AWS Lambda is 27% [31], and 90% of applications in Azure Functions use no more than 400 MB of memory while 50% of applications use no more than 170 MB of memory [32].

Motivated by the above two characteristics of public FaaS services, we introduce a user-side lightweight cache system, named μFuncCache, to optimize the call latency in FaaS applications. By introducing a local cache on the first cloud function at which requests

arrive, overall processing time, after the head node on the entire function chain, can be reduced when the cache is hit, achieving a reduction in function call latency without the need to perceive and modify the internal architecture of public FaaS platforms. Analysis and experiments in different application scenarios have shown that µFuncCache can effectively improve the performance of FaaS applications by consuming only a small amount of additional resources, thereby reducing the response time of the service.

To summarize, this paper makes the following contributions:

1. We propose an architecture design of a user-side lightweight cache system for public FaaS services, which leverages the container delay destruction mechanism and overbooked memory.
2. We present an efficient implementation of the cache system, which only consumes a small amount of additional memory with very small changes to the user's code.
3. We make a comprehensive evaluation of the proposed method in three typical scenarios, and show that a maximum reduction of 97% in latency can be achieved.

This paper is organized as follows. In Section 2, we give an overview of the related research work. Section 3 presents the architecture design and implementation details of the proposed cache system. We describe the evaluation in Section 4. Then, Section 5 discusses the advantage and pitfalls of our method. Finally, in Section 6, we summarize our findings and conclude the paper.

## 2. Related Work

We focus our discussion of related work on reducing data access latency in FaaS applications, and organize them into two categories: storage for FaaS and cache for FaaS. We also discuss the related orthogonal work, which can improve the overall performance of FaaS applications together.

Storage for FaaS mainly implements cloud storage that can match FaaS in terms of data consistency, scalability, and efficiency. Since function instances are stateless, FaaS applications have to access remote cloud storage when data or state is required, thus significant overhead occurs. To address the data transfer issues, Shredder [33] implemented a low-latency multi-tenant cloud storage that allows small compute units to execute directly in the storage node. Storage tenants provide Shredder with JavaScript functions that can interact directly with data without having to move it over the network. Further towards this goal, LambdaObjects [34] proposes a new serverless computing abstraction model, which stores data and computation together. This model enables the direct execution of functions on nodes where the data is stored by encapsulating the data as objects, and computation corresponds to the methods of accessing and modifying the data in the object. Pocket [27] implements an elastic distributed store that automatically scales and delivers the performance required by applications at a lower cost. Pocket dynamically sizes resources across multiple dimensions (CPU cores, network bandwidth, storage capacity) and leverages multiple storage technologies to minimize costs and ensure that applications will not experience I/O bottlenecks. Locus [35] combines cheap and low storage with expensive and fast storage to implement a FaaS-based data analysis system, achieving high performance while improving cost performance. At the same time, Locus provides a performance model to guide users in choosing the appropriate storage type and quantity. To minimize inter-host communication due to the state externalization of stateless cloud-native applications, ref. [36] designed a state layer architecture suitable for any kind of key-value store, which consists of a data placement algorithm that places states across the hosts of a data center, and a state access pattern-based replication scheme that decides the proper number of copies for each state, to ensure a 'sweet spot' in short state access time and low network traffic. Based on this work, ref. [37] proposed an edge computing platform for latency-critical FaaS applications, which decrease end-to-end latency for mobile end devices via orchestrating functions and in sync data.

Cache for FaaS is designed to improve the data access performance of FaaS instances and implement a distributed cache that meets latency and scalability requirements.

OFC [26] implements a transparent, vertically and horizontally elastic extended memory cache system, which uses machine learning models to estimate the actual memory resources required for each function call, and also organizes the redundant memory into a cache layer. InfiniCache [28] implements an in-memory object-caching system built entirely on FaaS instances for more general application scenarios. It combines erasure codes, intelligent billed control, and effective data backup mechanisms to maximize data availability and cost-effectiveness, while reducing the data loss and performance risks caused by the short-lived function instance. From the perspective of improving the function computing architecture, Lambdata [38] uses existing cloud object storage (such as AWS S3) to store data, but adds a caching layer where each node has its own object cache. When scheduling function instances, Lambdata prioritizes scheduling functions that handle the same data to the same node. Cloudburst [39] supports stateful function computation, and implements data storage based on the automatically scalable key-value storage system, Anna. It also introduces local caching to optimize access to multiple function instances. HydroCache [40] also builds a distributed cache based on Anna, and introduces a multisite transactional causal consistency (MTCC) protocol to improve the application performance while ensuring transactional consistency for the causal consistency problem, which would occur when multiple function instances of the same application execute across different nodes. Faa$T [41] has designed and implemented a transparent and automatically scalable distributed cache to compensate for the lack of data-access-oriented design in existing caching mechanisms. Each application has its own cache and automatically unloads it when the function instance stops executing. At the same time, it would be warmed up before the next run, which can improve application performance while reducing costs.

Table 1 summarizes the differences between µFuncCache and the above work mentioned. As can be seen, we compared the related work with µFuncCache from four dimensions, including whether additional storage space is needed (which will lead to additional budget expenditure), whether distributed management is required (which will lead to the need to deal with issues such as data consistency, reliability, and so on), whether the FaaS platform needs to be modified (which will affect the applicability of the work, and the public FaaS platforms cannot be modified by ordinary users), and whether the user code needs to be modified (which will increase development costs and bring potential compatibility issues).

**Table 1.** Summary of related work review.

| Work | Additional Storage Space | Distributed Management | Platform Modification | Application Modification |
|---|---|---|---|---|
| OFC [26] | Y | Y | Y | |
| Pocket [27] | Y | Y | | Y |
| InfiniCache [28] | Y | Y | | Y |
| Shredder [33] | Y | Y | Y | Y |
| LambdaObjects [34] | Y | Y | Y | Y |
| Locus [35] | Y | Y | | |
| [36] | Y | Y | Y | Y |
| [37] | Y | Y | Y | Y |
| Lambdata [38] | Y | Y | Y | Y |
| Cloudburst [39] | Y | Y | Y | Y |
| HydroCache [40] | Y | Y | Y | Y |
| Faa$T [41] | | Y | Y | |
| µFuncCache | | | | Y |

Y represents that the work has the corresponding characteristic.

As the primary design goal of µFuncCache is a user-side lightweight cache layer, it does not rely on other cloud services such as object storage, as well as on the need to modify the internal architecture of public FaaS platforms. By making use of local redundant memory to build a local cache, it reduces the management complexity and overhead caused

by the distributed cache management. Thus, μFuncCache can achieve independence of extra storage services and platforms, as well as maintain simplicity.

In addition, cache-based methods are currently widely used to reduce the startup delay of functions. Unlike the work in this article, these tasks involve caching the data required for the function instance itself to run, such as the runtime environment, in order to reduce cold start overhead. SOCK [42] caches interpreters and commonly used library files and provides a lightweight isolation mechanism. Nuka [43] designed a local packet caching mechanism to import the required software packages. SAND [44] and Photos [45] adopt the method of sharing runtime in function instances, while refs. [46–48], respectively, reduce loading and running costs by sharing network resources, image data, and the container itself. FaaSCache [49] considers keeping function instances running as a caching problem and reduces the number of new functions created by optimizing instance survival strategies. Recent work, including refs. [50,51], designed for serverless computing platforms in edge scenarios, also uses cache mechanisms to quickly create function instances.

Another type of research that is orthogonal to our work aims at memory sharing between function instances located on the same physical server, thereby reducing data movement overhead. FAASM [52] implements a new function runtime based on WebAssembly and allows for the memory sharing of functions within the same address space. Nightcore [53] implements efficient inter-function communication through shared memory by scheduling multiple requests for a function to the same container. Fastlane [54] achieves data sharing by introducing simple load/store instructions, treating functions within the same workflow as threads within a shared virtual address space. Ref. [55] designs a message-oriented middleware to achieve memory sharing for function instances located on the same physical server. Our method is not limited to the premise that function instances are located on the same physical server, and can improve the overall performance of FaaS applications together with these works.

## 3. Design and Implementation

For applications built on FaaS, if a cache can be introduced and hit on the first cloud function that the request reaches, the data access time after the head node on the entire call chain can be reduced significantly. However, unlike traditional cloud computing applications, cloud function instances do not survive for long periods of time, and individual instances have smaller resources. Therefore, it is necessary to redesign a lightweight cache that can adapt to the characteristics of cloud function instances. We first analyzed the principles that need to be considered when designing the target cache, and then designed and implemented the lightweight cache mechanism μFuncCache based on the basic working principle of the cache.

### *3.1. Design Principles*

Distributed caching often provides reliable caching services in the form of clusters, encapsulating complex topology structures and exposing an accessible interface to provide easy-to-operate services for applications, while adopting redundancy mechanisms to ensure high availability and scalability. The characteristics of cloud function instances determine that distributed caching mechanisms cannot be directly applied to FaaS platforms, but the aforementioned ease of use, reliability, and other features are still factors that need to be considered when designing a lightweight cache for cloud function instances. In addition, it is also necessary to consider that the maximum allowable memory allocated to cloud function instances is not as large as virtual machines, and they cannot run for a long time like traditional cloud computing nodes. In this work, μFuncCache is proposed and designed, considering the following aspects in order to design such a cloud function cache.

Usability. For cloud functions that use cache, it is necessary to define standardized usage interfaces. The interface needs to have a certain degree of stability, that is, regardless of how the logic inside the cache changes, the interface still needs to be guaranteed to

remain unchanged. This ensures that the parts of the business code that use cache do not need to be modified every time the cache structure changes.

Scalability. This feature does not need to be manually added or deleted on the function computing platform, like a distributed cache, but can be automatically implemented using the dynamic scaling characteristics of cloud functions. If a single machine cache code is added to a cloud function, the number of caches increases with the number of instances, and the number of instances is determined by the request. The more requests arrive, the more likely it is that duplicate items appear, and the cache hit rate increases accordingly.

Availability. The essence of availability is, in fact, to avoid a single point of failure in the entire cache cluster. For public FaaS platforms, we adopt a local cache to avoid this issue. Each function instance has its own lightweight cache, and the reliability of the instance is guaranteed by the platform.

Memory Cost. The memory allocated by a single function instance itself is not particularly large (usually between 128 MB and 3000 MB). If a stand-alone cache must be introduced in the business code, then little memory should be provided, not only to the task to be executed but also to the cache item to be stored. If the cache itself is too complex, it is likely to be worse than not caching at all. Therefore, the memory occupied by the stand-alone cache itself must not be too significant.

Access Cost. The cache was introduced to address the high response time caused by long call chains, but if accessing the cache itself takes too much time, then introducing caching will only have the opposite effect. Therefore, when designing a cache, it is necessary to consider the access time of the cache itself and analyze whether its costs and benefits are reasonable to some extent.

Data Freshness. The cache entries are actually obtained from remote storage (which may be the result of other cloud functions stored in object storage or databases). Once the remote data changes, it is necessary to update the corresponding cache entries in the shortest possible time. For this purpose, a timestamp is introduced into each cache entry and corresponding thresholds are set according to different scenarios. During each access, if a cache entry is hit, the cache can automatically detect whether the cache entry has expired. Once it has expired, new data will be obtained from remote storage to update it.

Cache Hit Ratio. Cache hit ratio refers to the percentage of requests that have received data from the cache as a percentage of the total number of requests, and is determined by the order of requests as well as the content of the requests. For this reason, when designing a cache, we need to consider the sequence of cloud function requests and choose an appropriate cache-replacement strategy based on the characteristics of different request sequences.

### *3.2. Architecture Design*

The architecture design of μFuncCache is shown in Figure 3. Every instance of cloud functions with cache code carries an independent standalone cache, and the caches of the instances do not need to communicate with each other. At any time, all cache contents are also independent of each other, which depends on the requests processed by the corresponding instance of the cache. In Figure 3, there are three function instances, corresponding to the three states when the request arrives. When a request arrives, if the cache on the instance can hit, the hit item will be directly returned; otherwise, the request needs to be handed over to the next cloud function or other cloud service for processing. In this design mode, the high availability and scalability issues of the cache are solved with the dynamic scaling characteristics of the function instance. In addition, the single-node cache, as it does not require communication, will only retain two operations: fetch and save. Therefore, its structure will be very simple, thus greatly saving the memory consumption of the cache itself.

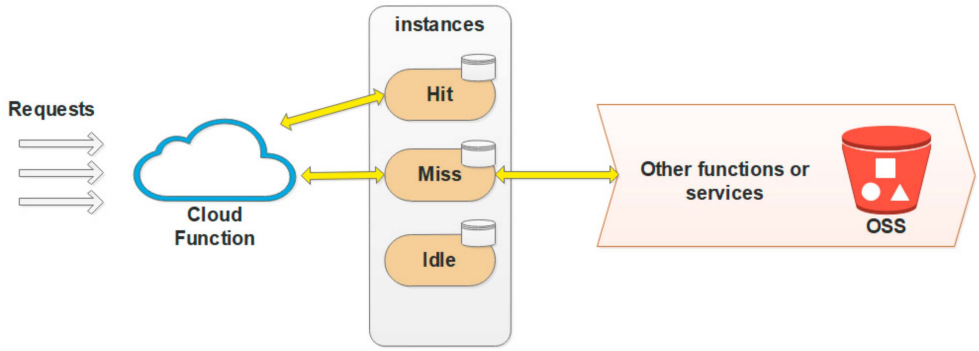


**Figure 3.** Architecture of μFuncCache.

Regardless of the type of cache, hit ratio has always been the core issue. It is obviously not feasible to increase the number of cached items on cloud function instances where memory allocation is already limited. Meanwhile, the request characteristics of FaaS applications, and appropriate cache replacement algorithms, should be considered to improve cache hit ratio. Common cache replacement strategies include First-In-First-Out (FIFO), Least-Frequently-Used (LFU), Least-Recently-Used(LRU), and Size. FIFO and LFU are generally suitable for sequential access mode. The former is the simplest first-in-first-out logic, while the latter determines the replacement weight based on the frequency of data item access over a long period of time. The replacement strategy of LRU only considers the time when access arrives, that is, the replaced data items are always the ones that have not been accessed for the longest time. Size, on the other hand, is different from other methods. The replacement strategy in Size is determined by the size of the cached data, swapping out the largest data item to accommodate more small data.

In high concurrency scenarios, where container reuse often occurs, requests often exhibit temporal locality, meaning that some previously referenced objects may be frequently referenced later. For example, in the takeaway delivery service, if a customer in a specific location sends a delivery request at a certain time, there is a high probability that multiple order requests will occur at that location during that period (e.g., during lunch time in an office building). For the same delivery service point around that location, there is a high possibility of duplicate planning tasks to obtain its shortest paths to customers the results of which can be reused to reduce cost. This temporal locality is precisely suitable for LRU replacement strategy, not for the FIFO and the LFU that only consider the access order.

The LRU algorithm execution process is briefly described below. The total number of cache items that can be stored in the cache is N. When the data is obtained according to the keyword, the corresponding cache item is searched throughout the cache. If found, the data in the corresponding cache items is returned. Otherwise, the data is fetched from the remote storage and added to the cache. If the number of cache items reaches N, the data obtained remotely replaces the data of the least recently accessed one. In addition, when considering data freshness, after cache hits, it is also necessary to determine whether the data in the corresponding cache item has expired. Cache entries are usually designed as a chain structure as they are accessed at different times. In order to facilitate the operation of individual nodes, we used a double-linked list to store cache items. Although the single move of a node is relatively simple, the time complexity still reaches $O(n)$. Table 2 shows the pseudocode symbol definitions for μFuncCache when performing data access operations. When the cache space is full, the new data items replace the data items that are the least recently accessed. The basic data structure of μFuncCache consists of a double-linked list and a hash table (defined as *list* and *map* in Table 2, respectively). A *node* corresponds to a cache entry, containing a keyword *key* and a value *data*, which is a copy of the data fetched. A node will be saved in *list* and *map* simultaneously.

**Table 2.** Symbol definition in the μFuncCache pseudocode.

| Symbol | Definition |
|---|---|
| *list* | double-linked list of nodes |
| *map* | hash table of nodes |
| *size* | current number of cache items |
| *capacity* | maximum number of cache items |
| *timestamp* | time stamp of a node |
| *threshold* | threshold of data freshness |
| *head* | head pointer in the list |
| *tail* | tail pointer in the list |

Algorithm 1 shows the pseudocode of the PUT operation in μFuncCache, which describes how to reduce the complexity of a store operation to $O(1)$ through a hash table, based on the LRU replacement strategy. All cache items are stored in *list* as node, and then every node in *list* is also stored in the hash table *map* (where the keywords accessed by the hash table are the keywords held by the node). Since the access complexity of the hash table is $O(1)$, it is possible to quickly determine whether there is a node with corresponding keyword in *list* during each operation. If an existing node holds a keyword queried, the old data is directly overwritten with new data, otherwise a new node is created to store the key and the data to be saved. If the total number of nodes exceeds the maximum number of allowed nodes after creating a new node, the next node pointed to by *head* is removed. Then, by moving the node to the previous position of *tail*, this indicates that the node is the node that μFuncCache has recently used. Finally, the timestamp in the node is updated, indicating the "production time" of the data.

**Algorithm 1** The pseudocode of PUT operation

**Input:** key $k$ and data $d$
**Output:** node *temp*

```
1:  if map contains k then
2:      get node n from map with key k
3:      replace the original data in n with d
4:      temp ← n
5:  else
6:      if capacity == size then
7:          delete the next node pointed by head in list
8:          size ← size − 1
9:      end if
10:     create a new node n′ with <k, d>
11:     size ← size + 1
12:     put <k, n′> into map
13:     temp ← n′
14: end if
15: move temp to the previous node of tail in list
16: update the timestamp of temp to system current time
```

Algorithm 2 shows the pseudocode of the GET operation in μFuncCache, during which the data freshness needs to be evaluated. If the difference between current system time and the timestamp in the node are larger than *threshold*, the data has expired and the latest data needs to be retrieved. After the data is fetched, it is cached by calling the PUT operation.

**Algorithm 2** The pseudocode of GET operation

**Input:** key $k$
**Output:** data $d$
**if** *map* contains $k$ **then**
  $temp \leftarrow$ get node from *map* with key $k$
  **if** system current time − timestamp of $temp \leq threshold$ **then**
    $d \leftarrow$ get the data from $temp$
    move $temp$ to the previous node of *tail* in *list*
    return $d$
  **end if**
**end if**
$d \leftarrow$ request for data from data source with the parameter of $k$
**if** $d$ is not null **then**
  PUT($k$, $d$)
**end if**
return $d$

It can be seen that, whether for a PUT operation or a GET operation, there is no traversal process for *list* or *map*. The time complexity of accessing *map*, all operations on node pointers, and freshness determination of data are all $O(1)$, so the time complexity of both operations in μFuncCache is $O(1)$.

### *3.3. Implementation*

We first define a double-linked list as the basic data structure of the LRU algorithm. The double-linked list will contain two special pointers, *head* and *tail*, the middle part of which is used to store cache entries. Each node in the linked list contains data items and pointers to the two nodes connected before and after the current node. In addition, the node also includes a timestamp variable related to data freshness, as well as the keyword key required to obtain the data. The nodes in the linked list are arranged in order based on the most recent time accessed. When the cache space is full, the node pointed at the tail will be the last to be eliminated, while the head node will be the first to be eliminated.

Both PUT and GET operations require $O(n)$ complexity, because the structure of the linked list determines that it can only be traversed from beginning to end when performing lookups. To this end, we use hash lookup to optimize, saving each node in the double-linked list in the *HashMap* in the *Java Util* package. The overall data structure of the entire cache is shown in Figure 4. The storage items in *HashMap* are in the form of key-value pairs, where the key is the one saved in the linked list, and the value is the corresponding linked list node. In addition, *HashMap* uses the zipper method to solve hash collisions, and, when the length of the conflicting linked list reaches a certain threshold (the default value is 8), it will be converted into the form of a red-black tree, so that the double-linked list can be accessed efficiently at a time complexity of approximately $O(1)$. When the PUT or GET operation needs to change the position of a linked list node, the node can be retrieved with $O(1)$ complexity from the hash table, and then placed at the tail again.

The latest cached data is automatically detected after the timestamp is set. Initially, the cache is specified with a size (the maximum allowed number of cache items to be stored) and a data expiration threshold. When data is accessed, μFuncCache will search the corresponding node from the *HashMap* according to the key. If the node exists, the difference between the current time and timestamp is compared with the threshold to determine whether the data has expired. If it has not expired, the current data is directly returned and the update method is called to update the timestamp. Otherwise, it is obtained from the remote storage and stored in the node. If the node does not exist, μFuncCache will obtain the corresponding data from the remote storage, store the key and the data in the newly generated node, and store the key and the node in *HashMap*. The node that has been accessed each time needs to be placed at the tail of the linked list, indicating that the data was the most recently accessed among all data (corresponding to the data at the head of the linked list, indicating that the data was the earliest accessed among all data). When

adding new nodes to the cache, it is also necessary to determine whether the cache exceeds the specified size. If it does, the head node is deleted.

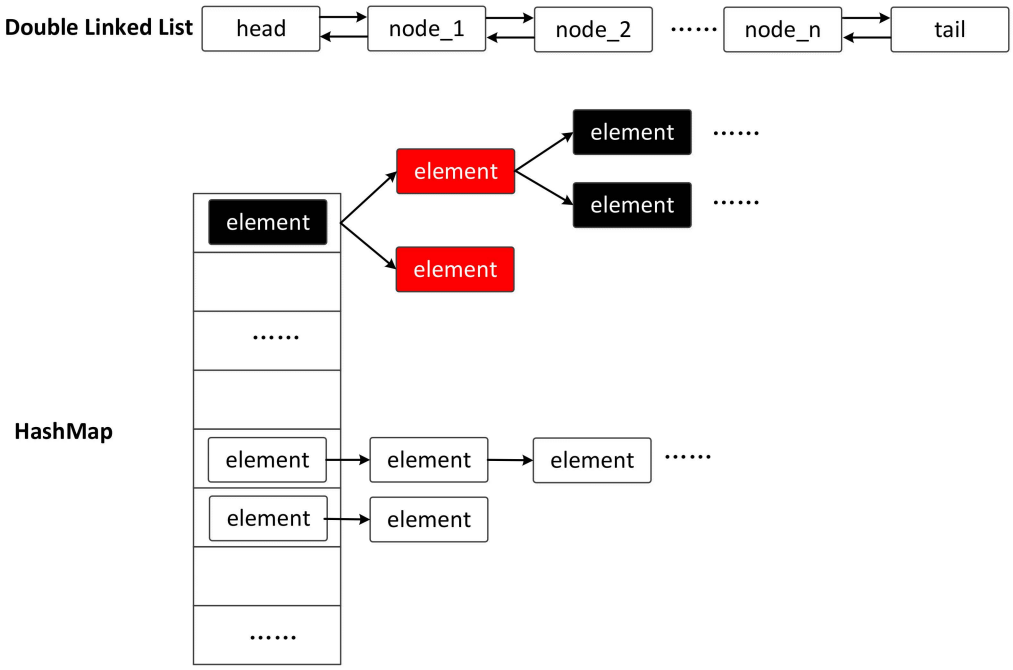


**Figure 4.** The data structure of μFuncCache.

When a FaaS application is deployed, the cached code is placed at the head of the call chain of cloud functions (similar to a gateway), and developers can set the size of the cache according to specific scenarios to store as much data as possible within limited memory resources. In addition, developers also need to set the corresponding data expiration threshold based on the freshness requirements of the data. If the data freshness requirements are high (such as real-time news, current hot images), it can be set to a smaller value. On the contrary, if the requirement for data freshness is not particularly high (such as user information, product information), it can be set to a larger value.

## 4. Evaluation

In this section, we will verify the effectiveness of μFuncCache through different experimental scenarios. The experimental cloud environment includes Alibaba Cloud FC, Alibaba Cloud Database Service MySQL, and Alibaba Cloud OSS. The programming language is Java with JDK 1.10, and APIs of Alibaba Cloud Database Service MySQL as well as Alibaba Cloud OSS are used in the tests. The local development and test environment is a personal computer (PC) with i7-8550U CPU, 8 GB memory, 512 GB SSD, Windows 10, JDK1.10, and Maven 3.6.1. The maximum allowed memory size for cloud functions is set to 512 MB, the maximum allowed running time is set to 60 s, and the trigger type is HTTP request.

It should be noted that, since μFuncCache is a user side caching mechanism and does not adopt a distributed architecture, we evaluate its performance improvement for FaaS applications through typical application scenarios. We did not compare it with the related work because they all adopt distributed architectures, which themselves have a certain complexity and time cost. We believe that an objective comparison cannot be made since μFuncCache can be seen as a local solution, and different applications would use different languages and implementations.

### *4.1. Test of Cloud Service Latency*

In order to further test the performance of cloud function instances, we tested the speed of local read and write, the remote access to databases, and the execution speed

of small computing tasks. The test results are shown in Table 3. It can be seen that, when reading and writing locally on cloud function instances or executing tasks with low computational complexity (simulating actual task load), its performance is basically consistent with that on the PC side. Once remote service access is involved, the access speed on the cloud function instance is much faster than PC, indicating that Alibaba Cloud has very fast internal network access speed. In addition, to ensure that the same cloud function instance is used for testing, a series of requests are sent on the PC side in the single-threaded synchronous request mode (the next request will be sent only after the result of the previous request response is obtained).

**Table 3.** The overhead of performing different tasks on local PC and cloud functions (ms).

| | Local R/W | Access Local Database | Access Cloud Database | Access OSS | Compute Fibonacci(30) |
|---|---|---|---|---|---|
| PC | 9/15 | 340 | 728 | 157 | 12 |
| Cloud Function | 8/13 | - | 63 | 62 | 11 |

In actual scenarios, applications based on FaaS are usually designed into multiple cloud functions based on different tasks, accompanied by one or even multiple function call chains. The form of call chain may be between cloud functions, such as when an application's functionality is split into multiple cloud functions, but only one gateway is exposed to the visitor. In this case, the corresponding request needs to be forwarded through the gateway cloud function. The form of concurrent call chains may also be between cloud functions and other services, such as the object storage services or database services mentioned earlier. The following sections will validate the benefits of adding cache through three experiments. It has been experimentally proven that the network delay on the call chain increases linearly with the length of the call chain, as described in the Section 1. If the cache is placed in the cloud function at the head of the call chain, once hit, no matter how long the subsequent call chain is, there is no need to make additional requests to obtain data. Therefore, to simplify the experimental process, we set the call chain length to 1.

### *4.2. Database Access*

Database access is a common scenario in cloud applications. After receiving the request, the cloud function queries the remote database and returns the results. The user requests the gateway through HTTP, and the gateway passes the parameters to the backend, then reads the data from the database and returns it. In this experiment, multiple rounds of testing were conducted to further observe the relationship between cache hit ratio and overall latency reduction. The content in the database table has only one user ID and one user name, and the size of the data is 676 (randomly generated). Each round of testing is randomly accessed 1500 times.

Figure 5 shows the relationship between the cache size and the average access time of 1500 accesses. We can see that, as the cache size continues to increase, the average processing time for a single time will significantly decrease, from 63 ms to 42 ms. However, clearly, we cannot set the cache size to a particularly large value, because the cache itself also requires additional memory, and the specific setting depends on the memory setting of the cloud function itself and the total data volume. Figure 6 shows the relationship between the cache size and hit rate per round (1500 accesses), and it can be observed that the hit ratio also increases linearly as expected, which is consistent with the general cache utility. We have summarized the correlation between the total time saved on cloud functions and the cache hit ratio, as shown in Formula (1).

$$\text{TotalSavedTime} = (\text{AllLinksTime} - \text{CacheTime}) \times \text{RequestTimes} \times \text{HitRatio} \quad (1)$$

When only a single cache is considered (the cache exists in the cloud function instance at the head of the call chain), the total time saved (TotalSavedTime) will be determined by

the network delay (AllLinksTime) (that is, the accumulation of each call time on the call chain in a single request), the delay (CacheTime) of a single access to the cache, the total number of requests (RequestTimes), and the cache hit ratio (HitRatio). The cache hit ratio is positively correlated with the cache set size. We can see that the cache will affect the overall time savings through the single access time and total hit ratio, which indicates that we need to minimize the additional time spent accessing the cache itself as much as possible. At the same time, setting the cache size to an appropriate value based on the memory situation of the cloud function instance can maximize the reduction in total latency.

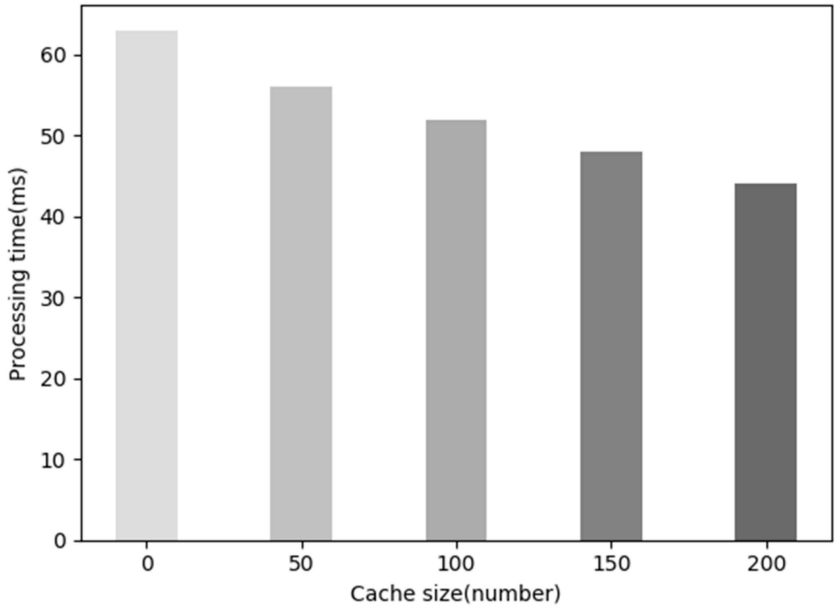


**Figure 5.** Performance of cloud functions with database access under different cache sizes.

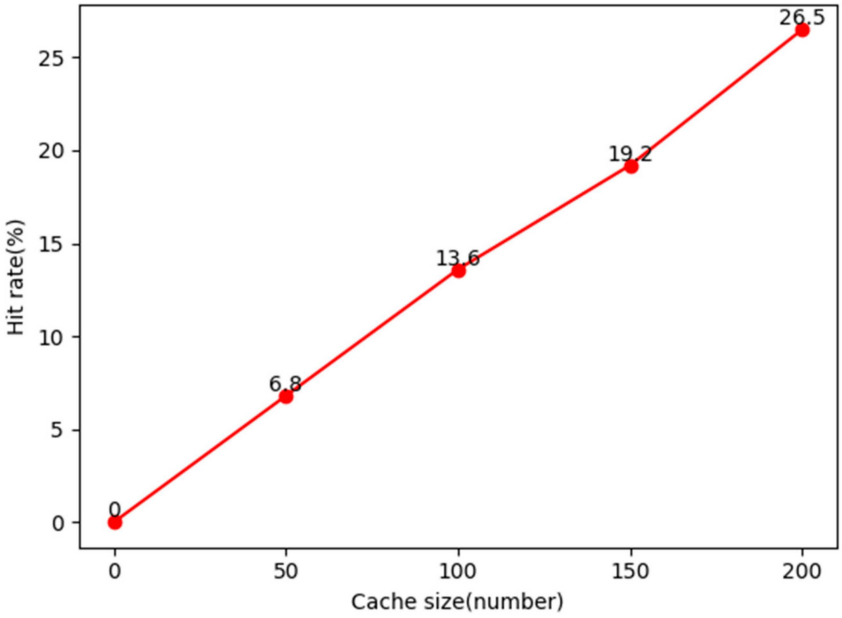


**Figure 6.** Hit ratio under different Cache sizes.

### *4.3. Path Planning*

Path planning is a typical compute-intensive processing task. When there are a large number of duplicate computing tasks or intermediate states, using cache for temporary storage can save a lot of computing resources and time. In the takeaway delivery application, there are always order requests from the same location during the peak ordering

period, such as schools, companies and other places where people gather. In this scenario, order requests also exhibit strong temporal locality, meaning that merchants with good reputation and close proximity are likely to receive a large number of orders from the same location during peak dining period. Therefore, there may be a large number of duplicate planning situations in the corresponding path planning for delivery. In order to reduce the repeated computation problem during path planning, we selected a small-scale map for simulation, which has 64 intersections and nearly 105 roads. We tested the request time with and without cache, respectively.

As shown in Figure 7, the request includes two parameters: starting point a and destination b. When the request is received by the cloud function instance, the program determines whether the local container already contains the map data required for path planning. If the map data does not exist, it needs to be downloaded from OSS. After the map file is downloaded locally, the program loads it into memory and saves the map in the form of an adjacency matrix. Otherwise, if the map already exists locally, the cache will be used to determine whether the paths from a to b have been calculated and exist in the cache. If they exist, the results will be directly returned after obtaining them (if freshness detection is required, the obtained paths need to be automatically determined whether they have expired through the threshold and timestamp parameters). Otherwise, the shortest path between a and b is calculated using the Dijkstra algorithm, and the calculation result is saved to the cache (a cache entry in the cache stores the shortest path from a certain starting point to all other nodes, which is the result obtained by executing any Dijkstra algorithm). The size of the cache is set by the user when the code is uploaded, and its elimination mechanism is triggered by the cache's automatic detection (storing new data will replace the oldest data that has not been accessed).

For the case where the cache was not added, we conducted 100 random requests for testing. The test results are shown in Table 4. When the cache is not added, the average response time for a single response reaches 102 ms. This is not only because the calculation task of path planning needs to be carried out every time, but also because of the network delay caused by the need to access OSS every time the map file is re-downloaded.

**Table 4.** Request response time in path planning (ms).

| | **Without Cache** | **With Cache** |
|---|---|---|
| Average response time for a single request | 102 | 3 |

Next, we conducted 300 tests with cache. Due to the fact that the Dijkstra algorithm can calculate all the shortest paths from a certain starting point to the remaining nodes each time it is executed, we cached all the shortest paths from a certain starting point as an item. In the cache, a certain starting point is used as a keyword, and its path to the remaining other nodes is saved as the cache value. The cache size was set to 6, which means it could store all the shortest paths contained in 6 starting points, approximately 10% of the total data volume (6/64).

From Table 4, it can be seen that, in compute-intensive tasks, cache can greatly reduce the execution time of the original cloud function, with an average processing time of only 3 ms per transaction, and nearly 97% time is saved. The main reason for the significant decrease in execution time is that cache avoids two main time-consuming steps on this task. It is necessary to download maps from OSS to the local container every time without cache, which is the bottleneck of this task due to network latency. The download time for each map is approximately 95 ms. At the same time, the path needs to be recalculated every time without cache, which also incurs additional time. On the other hand, after adding the cache, when the cache of the map file is not expired, 300 visits only require downloading the map once and, once hit, this can save time in calculating the path. Moreover, the time required for a single request to cache is even less than 1 ms, indicating that the additional cost brought by cache is very low and, even if there is a miss, it will not generate high latency.


**Figure 7.** Flowchart of path planning.

### *4.4. Hotspot Images*

The most significant difference between the hotspot image and the previous two experimental scenarios is that the cache items have priority characteristics, simulating scenarios similar to hotspot images accessed in online social networks. The data to be cached also becomes larger images rather than simple string forms. In this scenario, certain images are set as hotspot images in the search bar due to high user traffic. During a certain period of time, the number of visits to hot images will also exhibit temporal locality, meaning that, after a user completes their visit, it will trigger other users to access the image in the near future. Unlike the previous two experiments, we used two caches to solve this problem, one for storing hotspot images and the other specifically for regular images. We have 200 different images stored on OSS, each with a size of 4 MB, including 10 hotspot images. The keyword key in the cache is the image name, and the value is the absolute path of the image on the container.

Figure 8 shows the flowchart of image requests. Due to the optimization effect brought by cache, we did not simulate the process of generating follow-up queries for hotspot images. Instead, we specified some images in advance as the hotspot images in the experiment. The clients use the image name as the request parameter and, when the request reaches the cloud function, the program will determine whether the image is a hotspot image. Once the cache is hit, whether it is a hotspot image cache or a regular image cache, automatic timeout detection is required for the timestamp in the cache entry. If it is still within the threshold range, the cached images of the container instance will be directly

returned to the client. If there is a miss in the cache, the program will issue a request to the OSS to download the required image from the OSS to the local container, store its absolute address in the cache, and then return the image to the client.

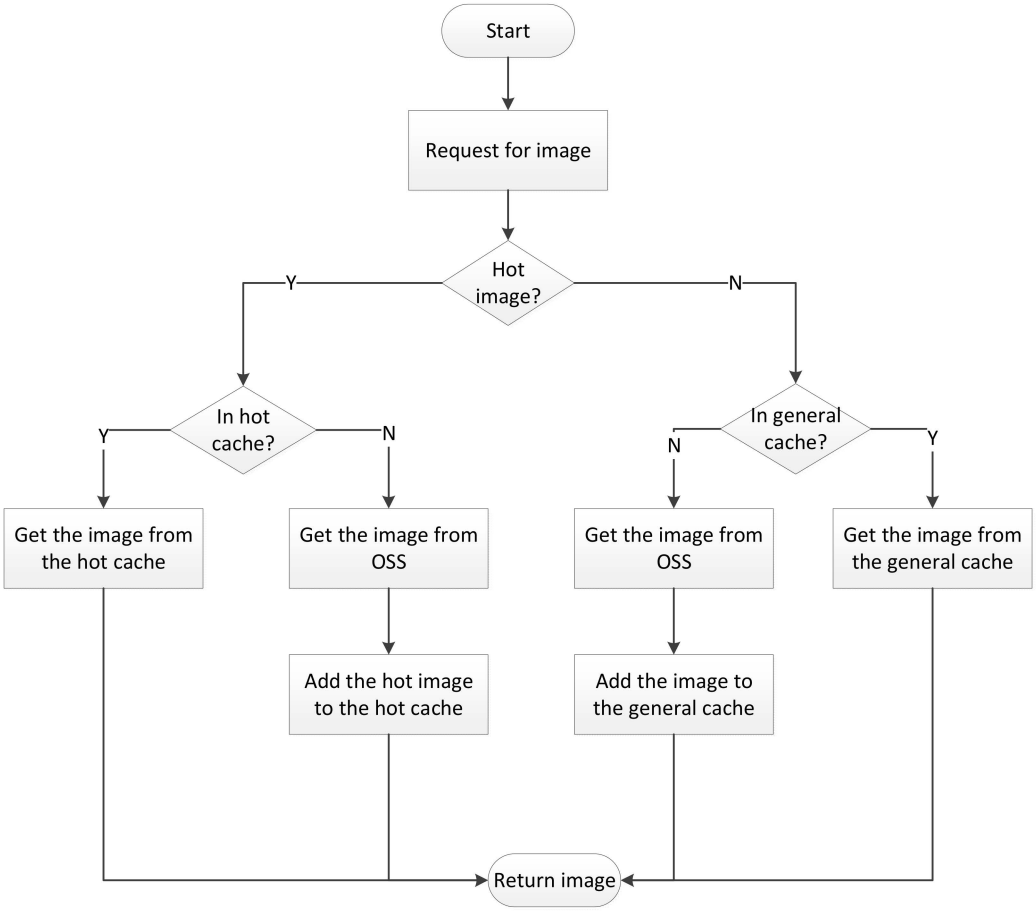


**Figure 8.** Flowchart of requests in hotspot images.

Since hotspot images are relatively fewer in comparison with regular images, but they more likely to be accessed, we controlled the probability of single request access to hotspot images by randomly generating numbers during the access process. The test is divided into two rounds, with each round requesting 400 requests from the cloud function, and the size of both caches is set to 10. In the first round, we set the probability of a single visit to a hotspot image to 1/2 and, in the second round, we set it to 1/3.

The results are shown in Table 5. We can see that the processing time after adding cache was significantly reduced, and the reduction effect was more significant when the number of hotspot image requests was higher. In addition, we also separately tested the additional overhead caused by the cache itself. After a single hit, the average processing time was 27 ms, which was used to read the local image file in the function instance into the response stream. The average processing time for a single miss was 641 ms, which was 20 ms longer than that without cache. This is because, if there is no caching, the request stream obtained by OSS can be directly read into the response stream returned to the user without the need to write to the local cache. The additional overhead of around 20 ms is very low compared to the original processing time of 600 ms, and hotspot images often have a high cache hit rate (the access ratio of hotspot images is relatively high). Therefore, it is obvious that cache can effectively reduce the original latency.

**Table 5.** Request response time in hotspot image (ms).

| | Without Cache | 1st Round | 2nd Round |
|---|---|---|---|
| Average response time for a single request | 621 | 381 | 417 |

## 5. Discussion

Serverless computing significantly reduces the cost for users, however, sharing or exchanging intermediate data between functions is challengeable. Existing public FaaS platforms usually isolate functions in short-lived, stateless containers, thus data should be duplicated or serialized repeatedly, incurring unnecessary performance and resource costs, especially in applications with one or more function chains.

At present, almost all the work related to reducing the data access latency of FaaS applications adopts distributed architectures, whether storage-based (such as [27,33–37]) or cache-based (such as [26,28,38–41]), which bring high elasticity but also introduce challenges such as managing data consistency or reliability. Most of the works require additional storage space, even some distributed cache systems, such as OFC [26], which also use over-booked memory and require additional object storage space for data backup. In terms of platform independence, some work, such as ref. [27], can only be implemented on open-source platforms, or only be conducted by researchers with privileges to directly develop on their own public FaaS platforms [41], which greatly limits the applicability of the proposed methodology. Finally, due to additional storage layers (e.g., [35]), caching layers (e.g., [28]), or even due to the introduction of new function execution models (e.g., [34]), the user's application code needs to be modified to achieve a performance improvement. Although this also limits the applicability of the proposed method, the method with little impact on user code can also achieve a higher performance cost ratio.

As can be seen from Section 4, µFuncCache adds local, lightweight cache in the first cloud function that the request reaches, thus the data access overhead in the applications can be reduced significantly. Furthermore, when the results can be reused in compute-intensive tasks, e.g., the Path Planning scenario, cache can also greatly reduce the execution time of the following cloud functions. On the one hand, µFuncCache makes use of the over-reserved memory commonly found in public FaaS services, so it does not need to occupy additional cloud storage space and depend on extra storage services. By adding local cache to function instance, and passing data access requests between functions or storage services using a unified API, this simple and lightweight design greatly avoids the implementation complexity brought about by distributed caching architecture. Meanwhile, as a caching solution on the user side, µFuncCache only requires a few changes to applications, and is therefore independent of the public FaaS platforms, making it easier to choose different platforms according to user budget.

On the other hand, as a lightweight caching system on the user side, although relatively easy to implement, µFuncCache requires modification to the user's code, which will increase development costs and bring potential compatibility issues. At the same time, it is necessary to occupy some of the memory originally planned for used by workload, which may have a certain adverse impact on the performance of applications. However, with the performance improvement and reduced complexity that accompanies it, we argue that these costs are acceptable.

## 6. Conclusions

The stateless nature of FaaS services not only brings significant delay, but also limits the potential application areas of this new paradigm. In order to eliminate the impact of this bottleneck as much as possible while keeping good elasticity, the existing research mainly revolves around building storage for serverless computing and distributed cache for FaaS platforms to reduce the data access latency. Although some progress has been made, improvements are still needed in terms of the occupation of additional storage space,

complexity, and platform dependence. This work proposes μFuncCache for addressing the problem of data access delay in function chains for applications built on public FaaS platforms. The architecture design, as well as implementation details, of the proposed cache system are presented. As a user-side lightweight cache mechanism, μFuncCache makes full use of the container-deferred destruction mechanism and over-booked memory, which are both common in public clouds, to reduce function latency with no need to perceive and modify the internal architecture of public FaaS platforms. The effectiveness of μFuncCache has been verified in three common applications, and the advantages and shortcomings of it were discussed.

As part of the future work, we will continue to evaluate μFuncCache on other public FaaS platforms, especially focusing on its impact on processes of FaaS-based application development, since users should implement the cache layer in their code. One possible solution to mitigate the impact of this problem is to design a common reference implementation template for different public FaaS platforms and languages, so that users can quickly develop and integrate them into their codes. Meanwhile, we will explore machine learning methods to predict the actual memory usage of each function instance, in order to more accurately determine the memory allocated to the cache and minimize the impact on the applications.

**Author Contributions:** Conceptualization, B.L. and Y.T.; methodology, B.L. and J.L.; investigation, Z.L.; data curation, B.L. and P.L.; writing—original draft preparation, review and editing, B.L. and P.L.; supervision, J.L. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was partially funded by the National Natural Science Foundation of China under the grant number U19A2060.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A.D.; Katz, R.H.; Konwinski, A.; Lee, G.; Patterson, D.A.; Rabkin, A.; Stoica, I.; et al. Above the Clouds: A Berkeley View of Cloud Computing. 2009. Available online: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf (accessed on 21 April 2023).
2. Cinar, K.; Justin, M.R.; Aadharsh, K.; Preston, M. Usage patterns and the economics of the public cloud. In Proceedings of the 26th International Conference on World Wide Web (WWW), Perth, Australia, 3–7 April 2017; pp. 83–91.
3. Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J.M.; Krauth, K.; Yadwadkar, N.; et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. 2019. Available online: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf (accessed on 21 April 2023).
4. Castro, P.; Ishakian, V.; Muthusamy, V.; Slominski, A. The rise of serverless computing. *Commun. ACM* **2019**, *62*, 44–54. [CrossRef]
5. Schleier-Smith, J.; Sreekanti, V.; Khandelwal, A.; Carreira, J.; Yadwadkar, N.J.; Popa, R.A.; Gonzalez, J.E.; Stoica, I.; Patterson, D.A. What Serverless Computing Is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* **2021**, *64*, 76–84. [CrossRef]
6. Wen, J.; Chen, Z.; Jin, X.; Liu, X. Rise of the planet of serverless computing: A systematic review. *ACM Trans. Softw. Eng. Methodol.* **2023**. *accepted*. [CrossRef]
7. AWS Lambda. Available online: https://aws.amazon.com/lambda/ (accessed on 21 April 2023).
8. Azure Functions. Available online: https://azure.microsoft.com/en-us/products/functions/ (accessed on 21 April 2023).
9. Google Cloud Functions. Available online: https://cloud.google.com/functions/ (accessed on 21 April 2023).
10. Alibaba Cloud Function Compute. Available online: https://www.alibabacloud.com/product/function-compute (accessed on 21 April 2023).
11. Apache OpenWhisk. Available online: https://openwhisk.apache.org (accessed on 21 April 2023).
12. OpenFaaS. Available online: https://www.openfaas.com/ (accessed on 21 April 2023).
13. Knative. Available online: https://knative.dev/ (accessed on 21 April 2023).
14. Wang, H.; Niu, D.; Li, B. Distributed Machine Learning with a Serverless Architecture. In Proceedings of the IEEE INFOCOM 2019—IEEE Conference on Computer Communications, Paris, France, 29 April–2 May 2019; pp. 1288–1296.
15. Yu, M.; Jiang, Z.; Ng, H.C.; Wang, W.; Chen, R.; Li, B. Gillis: Serving Large Neural Networks in Serverless Functions with Automatic Model Partitioning. In Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS), Washington, DC, USA, 7–10 July 2021; pp. 138–148.

16. Giménez-Alventosa, V.; Moltó, G.; Caballer, M. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Gener. Comput. Syst.* **2019**, *97*, 259–274. [CrossRef]
17. Enes, J.; Expósito, R.R.; Touriño, J. Real-time resource scaling platform for Big Data workloads on serverless environments. *Future Gener. Comput. Syst.* **2020**, *105*, 361–379. [CrossRef]
18. Jindal, A.; Gerndt, M.; Chadha, M.; Podolskiy, V.; Chen, P. Function delivery network: Extending serverless computing for heterogeneous platforms. *Softw. Pract. Exper.* **2021**, *51*, 1936–1963. [CrossRef]
19. Michael, Z.; Chandra, K.; Rich, W. Edge-adaptable serverless acceleration for machine learning Internet of Things applications. *Softw. Pract. Exper.* **2021**, *51*, 1852–1867.
20. Shankar, V.; Krauth, K.; Vodrahalli, K.; Pu, Q.; Recht, B.; Stoica, I.; Ragan-Kelley, J.; Jonas, E.; Venkataraman, S. Serverless linear algebra. In Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC), Seattle, WA, USA, 19–21 October 2020; pp. 281–295.
21. Rohan, B.R.; Tirthak, P.; Devesh, T. DayDream: Executing dynamic scientific workflows on serverless platforms with hot starts. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Dallas, TX, USA, 13–18 November 2022; pp. 1–18.
22. Hellerstein, J.M.; Faleiro, J.; Gonzalez, J.E.; Schleier-Smith, J.; Sreekanti, V.; Tumanov, A.; Wu, C. Serverless computing: One step forward, two steps back. *arXiv* **2018**, arXiv:1812.03651.
23. Alibaba Cloud Object Storage Service. Available online: https://www.alibabacloud.com/product/object-storage-service (accessed on 21 April 2023).
24. Daw, N.; Bellur, U.; Kulkarni, P. Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments. In Proceedings of the 21st International Middleware Conference, Delft, The Netherlands, 7–11 December 2020; pp. 356–370.
25. Lee, S.; Yoon, D.; Yeo, S.; Oh, S. Mitigating Cold Start Problem in Serverless Computing with Function Fusion. *Sensors* **2021**, *21*, 8416. [CrossRef] [PubMed]
26. Mvondo, D.; Bacou, M.; Nguetchouang, K.; Ngale, L.; Pouget, S.; Kouam, J.; Lachaize, R.; Hwang, J.; Wood, T.; Hagimont, D.; et al. OFC: An Opportunistic Caching System for FaaS Platforms. In Proceedings of the 6th European Conference on Computer Systems (EuroSys), Virtual, UK, 26–28 April 2021; pp. 228–244.
27. Klimovic, A.; Wang, Y.; Stuedi, P.; Trivedi, A.; Pfefferle, J.; Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, 8–10 October 2018; pp. 427–444.
28. Wang, A.; Zhang, J.; Ma, X.; Anwar, A.; Rupprecht, L.; Skourtis, D.; Tarasov, V.; Yan, F.; Cheng, Y. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, 24–27 February 2020; pp. 267–281.
29. Mikhail, S. Cold Starts in AWS Lambda. Available online: https://mikhail.io/serverless/coldstarts/aws (accessed on 21 April 2023).
30. Mikhail, S. Cold Starts in Azure Functions. Available online: https://mikhail.io/serverless/coldstarts/azure (accessed on 21 April 2023).
31. Ran, R. What AWS Lambda's Performance Stats Reveal. Available online: https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/ (accessed on 21 April 2023).
32. Shahrad, M.; Fonseca, R.; Goiri, Í.; Chaudhry, G.; Batum, P.; Cooke, J.; Laureano, E.; Tresness, C.; Russinovich, M.; Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In Proceedings of the USENIX Annual Technical Conference (ATC), Virtual, 15–17 July 2020; pp. 205–218.
33. Zhang, T.; Xie, D.; Li, F.; Stutsman, R. Narrowing the Gap between Serverless and its State with Storage Functions. In Proceedings of the ACM Symposium on Cloud Computing (SoCC), Santa Cruz, CA, USA, 20–23 November 2019; pp. 1–12.
34. Kai, M.; Andrea, C.A.; Remzi, H.A. LambdaObjects: Re-Aggregating Storage and Execution for Cloud Computing. In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage), Virtual Event, 27–28 June 2022; pp. 15–22.
35. Pu, Q.; Venkataraman, S.; Stoica, I. Shuling, fast and slow: Scalable analytics on serverless infrastructure. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, 26–28 February 2019; pp. 193–206.
36. Szalay, M.; Mátray, P.; Toka, L. State Management for Cloud-Native Applications. *Electronics* **2021**, *10*, 423. [CrossRef]
37. Pelle, I.; Szalay, M.; Czentye, J.; Sonkoly, B.; Toka, L. Cost and Latency Optimized Edge Computing Platform. *Electronics* **2022**, *11*, 561. [CrossRef]
38. Tang, Y.; Yang, J. Lambdata: Optimizing serverless computing by making data intents explicit. In Proceedings of the IEEE 13th International Conference on Cloud Computing (CLOUD), Beijing, China, 19–23 October 2020; pp. 294–303.
39. Sreekanti, V.; Wu, C.; Lin, X.C.; Schleier-Smith, J.; Gonzalez, J.E.; Hellerstein, J.M.; Tumanov, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.* **2020**, *13*, 2438–2452. [CrossRef]
40. Wu, G.; Sreekanti, V.; Hellerstein, J.M. Transactional Causal Consistency for Serverless Computing. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 83–97.
41. Francisco, R.; Gohar, I.C.; Inigo, G.; Pragna, G.; Paul, B.; Neeraja, J.Y.; Rodrigo, F.; Christos, K.; Ricardo, B. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In Proceedings of the ACM Symposium on Cloud Computing (SoCC), Seattle, WA, USA, 1–4 November 2021; pp. 122–137.

42. Oakes, E.; Yang, L.; Zhou, D.; Houck, K.; Harter, T.; Arpaci-Dusseau, A.; Arpaci-Dusseau, R. SOCK: Rapid task provisioning with serverless-optimized containers. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18), Boston, MA, USA, 11–13 July 2018; pp. 57–70.
43. Qin, S.; Wu, H.; Wu, Y.; Yan, B.; Xu, Y.; Zhang, W. Nuka: A generic engine with millisecond initialization for serverless computing. In Proceedings of the IEEE International Conference on Joint Cloud Computing, Oxford, UK, 3–6 August 2020; pp. 78–85.
44. Akkus, I.E.; Chen, R.; Rimac, I.; Satzke, M.S.K.; Beck, A.; Aditya, P.; Hilt, V. SAND: Towards high-performance serverless computing. In Proceedings of the 2018 USENIX Annual Technical Conference (ATC), Boston, MA, USA, 11–13 July 2018; pp. 923–935.
45. Vojislav, D.; Rodrigo, B.; Ankit, S.; Gustavo, A. Photons: Lambdas on a diet. In Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC), Seattle, WA, USA, 19–21 October 2020; pp. 45–59.
46. Mohan, A.; Sane, H.; Doshi, K.A.; Edupuganti, S. Agile cold starts for scalable serverless. In Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Renton, WA, USA, 8 July 2019; pp. 1–21.
47. Yan, B.; Gao, H.; Wu, H.; Zhang, W.; Hua, L.; Huang, T. Hermes: Efficient cache management for container-based serverless somputing. In Proceedings of the 12th Asia-Pacific Symposium on Internetware, Singapore, 1–3 November 2020; pp. 136–145.
48. Solaiman, K.; Adnan, M.A. WLEC: A not so cold architecture to mitigate cold start problem in serverless computing. In Proceedings of the IEEE International Conference on Cloud Engineering, Sydney, NSW, Australia, 21–24 April 2020; pp. 144–153.
49. Alexander, F.; Prateek, S. FaasCache: Keeping serverless computing alive with greedy-dual caching. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Virtual, 19–23 April 2021; pp. 386–400.
50. Chen, C.; Nagel, L.; Cui, L.; Tso, F.P. S-Cache: Function caching for serverless edge computing. In Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys), Rome, Italy, 8 May 2023; pp. 1–6.
51. Pan, L.; Wang, L.; Chen, S.; Liu, F. Retention-aware container caching for serverless edge computing. In Proceedings of the IEEE Conference on Computer Communications (INFOCOM), London, UK, 2–5 May 2022; pp. 1069–1078.
52. Simon, S.; Peter, P. FAASM: Lightweight isolation for efficient stateful serverless computing. In Proceedings of the USENIX Annual Technical Conference (ATC), Virtual, 15–17 July 2020; pp. 419–433.
53. Jia, Z.; Witchel, E. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Virtual, 19–23 April 2021; pp. 152–166.
54. Swaroop, K.; Ajay, N.; Vinod, G.; Arkaprava, B. Faastlane: Accelerating function-as-a-service workflows. In Proceedings of the USENIX Annual Technical Conference (ATC), Virtual, 14–16 July 2021; pp. 805–820.
55. Andrea, S.; Lorenzo, R.; Armir, B.; Luca, F.; Antonio, C. A Shared memory approach for function chaining in serverless platforms. In Proceedings of the IEEE Symposium on Computers and Communications (ISCC), Athens, Greece, 5–8 September 2021; pp. 1–6.