A Fast Cold-Start Solution: Container Space Reuse Based on Resource Isolation

Bin Li 1, Yuzhuo Zhan 1,* and Shenghan Ren 2,*

Abstract: With the emergence of cloud-native computing, serverless computing has become a popular way to deploy intensive applications due to its scalability and flexibility, and it has been increasingly applied in the field of big data processing on service platforms. Currently, the development momentum of cloud-native computing is strong, and serverless computing has become more attractive to the growing number of Internet services. However, how to more effectively address the issues of container resource usage and service startup time for serverless computing remains a huge challenge when exploring its potential. Our research is based on the complete life cycle of serverless functions and improves the performance of serverless computing by changing the original method of exchanging space for time or time for space. We focus on how to shorten the cold-start time of serverless computing while maximizing the usage of container resources. The research innovation is the dynamic control of functions and container pools, which mainly includes three aspects. First, we create a container pool with the classification identification based on the usage rate of functions. Then, we use namespace technology to achieve container resource reuse in the security isolated state. Next, we adaptively match the correspondence between functions and reusable container resources through system resource monitoring. Finally, the test results prove that converting the remaining space resources of the container into a prewarm container for new functions can effectively reduce the resource waste caused by idle function-containers, and container resource reuse can further shorten the cold-start time while ensuring the safety and isolation of functions. Compared to other open-source serverless platforms, our solution can reduce the cold-start time of general function calls to less than 20 ms and improve the ability to alleviate cold starts by 90% without enabling container prewarming.

Keywords: serverless computing; cold-start; container resources reuse; functions safety isolation

1. Introduction

Deploying intensive applications using serverless computing has gained increasing popularity and recognition among users. Existing platforms, such as Google Cloud Functions [1], IBM Cloud Functions [2], Azure Functions [3], and AWS Lambda [4], have successfully verified the value and potential of serverless computing on the road to commercialization. However, despite the achievements of serverless computing while demonstrating its excellent generalization capabilities, it still faces two major challenges: the occupancy of container resources and the cold starting of function-services. We believe that such challenges will attract more experts and institutions to join the research work on serverless computing. Therefore, our current research focuses on how to further reduce the cold-start time of the function-services while improving container resource usage.

For serverless computing, the key elements to ensure service capability and quality when providing cloud services with different features and performance to users include the following:
First, isolation of functions needs to be implemented in a sandbox that ensures performance and functional security. Typically, cloud providers use containers or virtualization technologies to provide a reliable runtime environment for different tenants, such as the current mainstream sandbox technologies Docker [5], gVisor [6], Kata [7], Firecracker [8], etc.;

Second, it is necessary to provide various middleware that can support function customization, communication data collection, and operation status monitoring to achieve loose coupling of functions with the underlying platform [9]. In addition, to expedite the initialization of instances to start, functions are usually wrapped with a one-to-one or one-for-all prewarm approach [10];

Then, the availability and stability of user applications need to be ensured by dynamic container scheduling, not only to avoid resource contention but also to perform the recovery of idle resources. For example, the literature [11–13] has implemented resource scheduling optimization from different levels;

Finally, a series of BaaS components are needed to provide functional services, such as queue management [14], trigger binding [15], data caching [16], and DevOps tools [17], as a way to better achieve flexible system orchestration.

Based on the above analysis we can see that the storage and invocation of functions require virtualized resource allocation to create the runtime environment. Container technology can provide a safe and isolated sandbox for functions under the premise of memory and CPU overhead so that functions can be loaded and executed in the virtualized sandbox. Since serverless service content may contain a large number of function calls, containers need to accomplish function wrapping by mapping virtual resources from the application layer to the underlying physical resources, consuming some physical resources and scaling their capacity from MB to GB. An obvious example is that, on a machine with 16 GB of RAM, the maximum number of containers created is usually limited by the amount of available memory, which may support only a few thousand containers. This limitation can negatively impact the platform’s storage resources and application service scaling if resource utilization is not improved during container initialization, dynamic scaling, and recycling. On the other hand, although smaller memory and CPU overhead can be achieved using container technology compared to virtual machines, the start-up time of functions and isolated container resources remains a paradox. We can understand this issue as a game between space and time, which happens to be the biggest problem to be solved when deploying serverless computing since container startup times of a few seconds or hundreds of milliseconds imply a completely different serverless service experience. To address the above issues, our research and contributions include the following:

- Unlike the mainstream open source serverless platform, which classifies the status of function containers as “running” and “idle”, we use Prometheus to monitor function containers by extracting key parameters including memory usage, CPU usage, and QPS. Based on these parameters, we add a function container status of “low usage”, which aims to obtain more reusable resources from the idle and low usage container pools without consuming additional system resources.

- We innovatively design a function wrapper with host privileges based on the nesting technique (namespace) and resource restriction technique (cgroups), which can “steal” the remaining resources from idle containers and low usage containers, and we transform these resources into a temporary container space in the form of a child process to meet the new function resource usage requirements in a safe and isolated manner.

- In our designed cold-start solution, when a new function needs to establish an initialized runtime environment, we use the nearest similarity rule of the CPU and memory resources to dynamically match suitable container resources from the idle container pool or the low usage container pool, and the runtime only needs to prepare the dependencies of the new function itself, which can effectively achieve a fast response to a cold start.
Our research goal is to achieve dynamic orchestration and management of instance-level localized containers based on Docker containers. The paper shows the process of implementing cold-start time compression for functions using technologies, such as function status monitoring, container namespace nesting, and virtual resource constraints. Sections 1 and 2 are the introduction and related work, respectively. Section 3 discusses the setting of container pools, Section 4 addresses function encapsulation, and Section 5 concerns dynamic management of container resources. Finally, we demonstrated the significance of our research work through a large number of tests, and we discuss future research content.

2. Related Work

1. Virtual resource isolation

To provide high-performance and flexible virtualization services for users, how to balance the isolation performance of virtual resources and the startup latency of user applications is a key consideration in serverless computing. Containers are a common function isolation mechanism that utilizes a Linux kernel to perform resource isolation and create containers as different processes in the host [18]. For a created container, it can be isolated from other processes in the shared system kernel through namespace. In the absence of hardware isolation, serverless computing uses containers to create sandboxed environments that encapsulate functions, which can achieve lower application startup latency compared to the traditional VM-based virtual machine manager VMM (Virtual Machine Manager) [19,20]. The most representative container engine technology is Docker, which has been widely used in serverless systems. By packaging software into a lightweight, standardized RunC container, this technology can meet the needs of different operating environments, such as libraries or runtime. In addition, some Docker-based container runtime optimization techniques have been proposed to better adapt to the requirements of applications in serverless systems. For example, SOCK proposed an integrated solution for serverless RunC containers [21], and CNTR split the container image into “fat” and “slim” parts [22]. Due to the relatively low isolation level of the containers in serverless computing, some cloud services will adopt a secure container solution, which can avoid adverse situations, such as privilege escalation or information disclosure. For example, Microsoft proposed a Hyper-V container for running instances in highly optimized microVMs [23]; Google has the lightweight sandbox solution gVisor; and a FireCracker technology for creating and managing multi-tenant containers was proposed by AWS. The performance comparison of container technologies is shown in Table 1.

<table>
<thead>
<tr>
<th>Virtualization</th>
<th>Startup Latency (ms)</th>
<th>Isolation Power</th>
<th>OSkernel</th>
<th>Hotplug</th>
<th>OCI Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional VM</td>
<td>&gt;1000</td>
<td>Strong</td>
<td>Unsharing</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Docker</td>
<td>50–500</td>
<td>Strong</td>
<td>Host-sharing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SOCK</td>
<td>10–50</td>
<td>Weak</td>
<td>Host-sharing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Kata</td>
<td>100–500</td>
<td>Strong</td>
<td>Unsharing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Hyper-V</td>
<td>&gt;1000</td>
<td>Strong</td>
<td>Unsharing</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>gVisor</td>
<td>100–500</td>
<td>Strong</td>
<td>Unsharing</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FireCracker</td>
<td>100–500</td>
<td>Strong</td>
<td>Unsharing</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Unikernel</td>
<td>10–50</td>
<td>Strong</td>
<td>Built-in</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

As shown in Table 1, these container technologies provide a strong security isolation capability for the host kernel and tenants, but they cannot effectively solve the issue of instance startup latency. Furthermore, some container technologies sacrifice flexibility to highlight security and performance, such as Unikernel [24].
2. Container prewarm startup

Although we can reduce the container’s cold-start latency by providing a lightweight sandbox mechanism, the compatibility of the sandbox mechanism with the container or VM may not be perfect. Therefore, to balance the responsiveness and compatibility of serverless computing, container prewarm startup has become a widely used solution. Currently, the prewarm methods for containers can be generally divided into two types: one-to-one and one-for-all. Typically, one-to-one prewarms from a fixed resource pool, such as Azure Functions presetting a fixed size prewarm pool for each function instance [25], or dynamic predictive prewarms based on historical data tracking, such as the implementation scheme proposed in reference [26]. One-for-all prewarm is the execution of function instances in a precached sandbox, such as the global prewarm scheme proposed in reference [27].

One-to-one prewarm reduces cold-start latency by exchanging memory resources. However, the obvious drawback concerns how to ensure the rationality of memory resource allocation by accurately measuring the prewarm time of the container [28], as well as ensuring the accuracy of prediction results when historical tracking data are insufficient. In contrast, one-for-all can reduce additional memory resource overhead, but there are still some challenges in related research work, such as the huge template image size or conflicts among various pre-imported libraries [29]. Thus, which container prewarm method should we choose, “suiting the remedy to the case”, is crucial [30].

3. Container resource elasticity management

When dozens, hundreds, or even more functions coexist on a single host, a difficult task is to reasonably schedule resources to ensure the performance of each function. In the traditional serverless model, researchers typically introduce a load balancing manager and a resource monitoring manager to perform the scheduling and orchestration of container resources, and they generate scheduling policies at three levels: the resource level, instance level, and application level [31,32]. The scheduling strategy of the resource level is to avoid wasting resource supply, and it adaptively adjusts resources through dynamic control of the cluster to meet elastic workloads [33,34]. The scheduling strategy of the instance level is to achieve load balancing for multiple containers based on hash methods or multi-objective methods and to optimize throughput, response time, resource usage, etc. [35,36]. The scheduling goal of the application level is to better balance the workload of each node and to perform data-driven scheduling of internal calls through application level topology, which can reduce the overhead of system orchestration [37,38].

3. Container Pool Resource Classification

3.1. Idle Container Identification

We set two types of container pools based on the running status: an idle container pool and a low usage container pool. The idle container pool consists of some function containers that have not been requested for a long time and are waiting to die. The low usage container pool contains function containers with low CPU or memory usage.

The maximum value of container resources is allowed to be set by the user, such as memory (M) and CPU (CP). The system log information, such as memory, CPU, and QPS for localized deployment of serverless computing, comes from Prometheus, the job of which is to collect information for each POD. We designed a container pool resources classification service based on the POD information obtained by Prometheus. For the QPS indicator, we introduced a timer to measure idle time. The calculation of idle time is the time difference between two or more adjacent QPS indicators that are 0, and we use $T_1, T_2, \ldots, T_m$ to identify the idle time difference between adjacent POD containers. When the accumulated idle time $T$ exceeds the idle time threshold $T_{idle}$ (usually, we understand that $T_{idle}$ should not exceed 1/2 of the function idle and wait for the automatic death time based on mature project experience), such containers are identified as idle containers.
(C_idle). Once the QPS index of the POD container is greater than 0, the idle time T needs to be recalculated:

$$\begin{align*}
    C_{idle} & \{ \\
         T & = T_i + T_{i+1} + \ldots + T_m \quad (i = 1, 2, \ldots, m, \ i < m) \\
         T & > T_{idle}
    \}
\end{align*}$$

We set the default value of $T_{idle}$ to 30 s. Considering that the setting of $T_{idle}$ may affect the identification of idle containers, we evaluated this impact in subsequent tests (see Section 5.2 for details) and recorded the M, CP, and PODIP information of containers meeting the conditions of Formula (1).

### 3.2. Low Usage Container Identification

Because functions do not always occupy all the resources of the container during their running lifecycles, our question concerns how to identify these containers with low usage rates and rationally reuse the resources in these containers. Our solution is to use indicator $C_u$ to record container memory and CPU usage, and we set the statistical duration of all container usage indicators to 5 min (assuming that the stored data are M). Based on our engineering experience, we have set the lower limit of CPU and memory usage to 30%. If during the statistical period of container resource usage, 95% of CPU and memory index data are less than 30%, we label such containers as low usage containers:

$$\begin{align*}
    & C_t = C_{u(0.95m)} \\
    & C_u^M = \frac{M_u}{M} \\
    & C_u^C = \frac{C_P u}{C_P} \\
    & (C_u^M, C_u^C) \leq (30\% M, 30\% C_P)
\end{align*}$$

$M_u$ is the usage of memory resources in the POD container; $C_u^M$ is the usage of POD container resources; $C_P u$ is the usage of CPU resources in the POD container; $C_u^C$ is the usage of CPU resources in the POD container; and $C_t$ represents 95% ($C_u^C$, $C_u^M$) less than 30% ($M$, $C_P$).

The goal of our design is to identify containers that are not highly used and relatively smooth. To avoid duplicate statistics of container resources, we do not record idle containers $C_{idle}$ and containers $C_{re}$ that are being reused. All statistical information related to POD containers is stored in ETCD, and $C_{lu}$ is used to represent a low usage container pool.

### 4. Function Wrapper

#### 4.1. Wrapper Design

We provide a public function wrapper that can provide four types of interfaces for instance-level function reuse, including cold-start triggering of reused resource functions, startup or death of the function runtime framework, request forwarding processing of function data, and health monitoring of functions.

The main work of the wrapper is divided into two parts: one is the function request processor, and the other is the function runtime framework and code. The processor and functions are linked through grpc. The wrapper uses namespaces and cgroups to run functions, ensuring safe isolation of functions. Unlike other types of function containers, the wrappers have the operation permissions of the host. When a container is recognized as an idle or low usage container, the container resource classification service starts a new namespace sub-process and runs the GRPC service. The resource settings of this subprocess are derived from the remaining resource size of the container. The container function service subprocess preinstalls all dependent packages for the to-be-helped functions. When the container is stolen by a new function, the processor needs only place the new function in the specified directory. Because the idle container function no longer receives requests, we stop the idle container function process to achieve maximum resource usage. When the
original function in the idle container receives a request again, regardless of whether the free container function has been stolen or not, we provide a fast cold-start service for the function with the highest priority. The main code-lists for the wrapper implementation is shown in Figure 1.

Figure 1. The main code implemented by the wrapper. (a) Initialization registration of function; (b) initialization of child function.

4.2. Container Resource Reuse

A function image contains two parts: a function wrapper and a function code package. The function code package is stored in the form of a zip file package. Function mirroring can render our platform compatible with mainstream scaling solutions, and the zip package of the function code can be used when reusing function resources. In the system we designed, there are two ways to start functions: one is to start the original function with the function wrapper (such as function image A), and the other is to reuse the function container resources with a cold-start function (such as function B).

Figure 2 shows the process of container resource reuse in two states:

- As shown in (a), function A is recognized as an idle function container. The processor will end the subprocess of function A and start a new namespace subprocess in the new directory of the container. The size of the new function subprocess and that of the container’s resources (i.e., requests, limits) is the same, and all dependent packages for the functions to be helped are installed. When a new request for function B arrives, the processor receives a cold-start request from function B forwarded by the scheduling service. Then, the processor sends a request to the internal function subprocess, which downloads the code package for function B. After the code for function B is ready, the processor forwards the request to function B, and the cold-start of function B ends. If the request for function A arrives again, the idle container of function A checks whether the to-be-helped function (such as function B) is being stolen. If a function is being stolen, because function A has the highest priority permission, the idle container of function A refuses to receive new requests. The processor removes the code and log information of the helped function B and then downloads the function A code package to process the request for function A. After the new function A image is prepared, the request for function A is redirected to the new function A image instance, and the idle container life cycle of function A ends.

- As shown in (b), function A is recognized as a low usage function container. The processor starts a new namespace subprocess in the new directory of the container. The resource size of the new function subprocess is consistent with the remaining available resources of the container, and all dependent packages for the functions to be helped are installed. When the processor processes the cold-start request of function B, if the image of function B is not ready, the helped function B continues to provide
services. Once the image of function B is ready, the processor receives a request to clean up the helped function B to provide services for other to-be-helped functions.

![Diagram](image)

**Figure 2.** Resource reuse-based two-container states. (a) Resource reuse of idle container; (b) resource reuse of low usage containers.

5. Dynamically Matching Reusable Container Resources

5.1. Scheduler Service

To dynamically match container resources, we implemented a request scheduler service. The scheduler can dynamically adjust the service scheduling method based on the set rules. Specifically, the scheduler accesses ETCD when a new request arrives and queries whether there are any function container instances that can provide services. If confirmed, the scheduler service forwards the request to the function instance. Otherwise, the scheduler performs two operations in parallel: one is to start function mirroring (not accepting first requests); the other is to select available containers from the container pool to perform resource reuse, which can meet the needs of a fast cold start. Of course, if there are no matching container resources in the container pool, the function waits for the function mirror to start successfully and forward the first request. The dynamic matching process of container resources is shown in Figure 3.

![Diagram](image)

**Figure 3.** The dynamic matching process of container resources.

As shown in Figure 3, until the function mirroring service is ready, requests continue to be forwarded to the helped function. When the function image is successfully started, the
scheduling service writes the service-ready flag in ETCD. The scheduling service detects that there is already a function instance providing service when the next request arrives, and the request is forwarded to the function instance. The function processor receives a request for the scheduling service to kill the helped function resources. After the last request is processed, the reused resources are released.

5.2. Relieve Cold-Start

There are two types of container pool resources in our system, so there are two types of cold-start situations (as shown in Figure 4): one is function G stealing idle container C to relieve a cold start; the other is function H stealing low usage container D to relieve a cold start.

The two situations of cold start.

The cold-start process of the system is as follows:

Step 1 (G-1, H-1): The request for function G and function H arrives;

Step 2 (G-2, H-2): The scheduler service accesses the ETCD and checks whether function instances G and H are providing services. If the scheduler service confirms that there are function instances providing the service, continue with step 3. Otherwise, skip to step 4;

Step 3 (G-3, H-3): The scheduler service forwards requests to function instances G and H, respectively;

Step 4 (G-4, H-4): The scheduler service accesses the ETCD and checks whether there are available container resources in the container pool. If so, continue with the following steps. If not, immediately trigger a cold start;

Step 5 (G-5, H-5): The scheduler service selects the idle container C to execute the fast cold start of function G, and it selects the low usage container D to execute the fast cold start of function H. At the same time as the fast cold start, the mirror containers of functions G and H are also triggered;

Step 6 (G-6, H-6): The scheduler service forwards requests from functions G and H;

Step 7 (G-7, H-7): After the scheduler service determines that the function G container and function H container have been started, it forwards subsequent requests to the mirror containers of function G and function H, respectively;

Step 8 (G-8, H-8): The scheduler service sends resource release requests for functions G and H.

There are two points to note during the execution of the above steps:

- The idle container: If there are new requests for function C during the reuse process, we prioritize ensuring the service requests of function C. Function G no longer receives new requests and immediately releases the current resource after processing the current request. In addition, when the function G image does not start successfully,
the scheduler service attempts again to search for reusable container resources in 
the container pool. If there are still no suitable container resources to match, or the 
function G image does not start, function G waits for its image to start successfully.

- The low usage container: If the QPS of function D suddenly increases during the 
  process of reusing function D, and function H is also processing requests, it causes 
an increase in resource usage. In extreme cases, function D satisfies this request through 
the container scale-up. After the processing of function H is completed, the reused 
function D resources are released, and this process only briefly affects the container 
scale-up of function D.

5.3. Container Resource Selection

Another key point that we need to address in the process of container resource reuse
is how to quickly select suitable container resources from the container pool. Our approach
is to prioritize the use of the idle container pool and then use the low usage container pool.
According to the actual usage of the CPU and memory, the principle of matching POD
container resources is to round up, ensuring that the container pool can perform cold starts
for more functions. The query process for dynamic matching of container pool resources is
shown in Figure 5.

For monitoring resource indicators, including the CPU, memory, and PODIP, we use
HASH tables for storage. The CPU and memory are measured in units of 10 Mi and 10 m,
respectively, and the KEY value is obtained by rounding down, for instance, M: 509 Mi, CP:
702 m, PODIP: 172.0.0.4, the storage format of the HASH table: [500: [700: [172.0.0.4]]]. If
there are multiple HASH values that need to be stored, we can perform an insert operation
after the array. When we need to query these stored data, the method we use is still
rounding up, for example, the stored data are [M: 491 Mi, CP: 696 m], the query keywords
is [M: 500 Mi, CP: 700 m].

Assuming that there is a PODIP in the returned query results that can meet the
requirements of resource reuse, we mark it as reused and update the record in the ETCD
to avoid duplicate selection of the PODIP. In the returned query results, there may be
more than one PODIP that can perform resource reuse. We choose the first value in the
ranking results based on the principle of priority matching for the free container pool. The
PODIP that has been successfully matched and reused is removed from the query list.
Sometimes we may encounter situations in which the query result is empty, so we need
to adjust the input query criteria. For the query keywords M and CP, the query range for
each modification is 10 units. We can repeatedly modify the query values until we select a
container that meets the conditions. In extreme cases, we may not be able to find a suitable
container after traversing the entire container pool due to bad luck. This process will be
frustrating, but all we can do is wait for the cold start of the function image.
6. Experiment

6.1. Benchmark Configuration

The configuration of benchmark in the testing environment that we built is shown in Table 2.

Table 2. The baseline test conditions.

<table>
<thead>
<tr>
<th>Option</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>CPU: Intel(R) Xeon(R) Gold 6226R CPU @ 2.90 GHz</td>
</tr>
<tr>
<td></td>
<td>Cores: 8, DRAM: 16 G, Disk: 100 GB SSD</td>
</tr>
<tr>
<td>Software</td>
<td>Operating system: Linux version 4.15.0, Docker: 20.10.13</td>
</tr>
<tr>
<td></td>
<td>Runc version: 1.0.3, Containered version: 1.5.10</td>
</tr>
<tr>
<td>Container</td>
<td>Container runtime: Python-3.10.0, Linux with kernel 4.15.0</td>
</tr>
<tr>
<td></td>
<td>Function container limit: 20 for each function on each node</td>
</tr>
<tr>
<td></td>
<td>Prewarm pool size in OpenWhisk: 2 on each node</td>
</tr>
</tbody>
</table>

Based on the configured environment, we set up 300 functions that comply with the Pareto distribution in the testing platform.

6.2. Comparison Relationship of Function Containers

By tracking and monitoring the resource reuse of idle function containers and low usage function containers in the container pool, our testing objective is to observe the quantity correspondence among idle function containers, low usage function containers, and resource reused function containers during the process of alleviating cold starts. The test results are shown in Figure 6.

![Figure 6. Correspondence between the numbers of different types of function containers.](image)

As shown in Figure 6, during the 12-h testing process, we randomly divided 300 functions into two types: one is to set the request interval of the function to 2 min; the other is to set the request interval of functions to 20 s, which ensures that these functions can be accurately recognized by the system as idle function containers and low usage function containers. For the remaining functions, we randomly select a portion for cold-start testing.

From the above test results, we can see that the number of idle function containers and low usage function containers is significantly higher than that of cold-start function containers, and the resources in the container pool can meet the needs of cold-start functions. It should be noted that, when a cold-start request arrives, there may be a situation in which the container type is not recognized by the system, and such container resources cannot be stolen by the cold-start function. Therefore, the cold-start containers recorded in the test results represent resource reuse function containers, which are inversely proportional to the number of idle function containers and low usage function containers.
6.3. The Impact of Idle Function Threshold

Next, our testing involves changing the value of $T_{idle}$ and observing the situation in which 300 functions are identified as idle function containers.

We set the idle time thresholds $T_{idle}$ to default values $T_{default}$ of 30 s, 90 s, and 120 s in the same testing environment. Figure 7 shows that the number of idle function containers in the system is not sensitive to changes in $T_{default}$ (we named the test system Alpheidae). The test results indicate that, if the container is recognized as an idle function container after 30 s, the probability of the container receiving requests again in a later period of time is lower. Of course, our experiment has limitations, but even if a function container has been identified as an idle function container, if the function receives a request again, regardless of whether the current container resources have been stolen, our system immediately starts the idle function container pool to provide fast cold-start services for the function, ensuring the reliability and stability of the function.

![Figure 7. The impact of idle time threshold setting on system recognition of idle function containers.](image)

6.4. Comparison Test of Cold-Start Latency

Because there may be deviations in the cold-start time when functions are applied on different platforms, we take the function scheduling of a node as an example to verify the performance of our system under the same testing conditions through cold-start testing. For our comparison test object, we chose OpenWhisk, which is an open source serverless platform that can respond to events of different sizes by running functions. During the test, we set two running states based on OpenWhisk, which were “OpenWhisk-Prewarm-Disabled” and “OpenWhisk-Prewarm”. We prepared four prewarm containers in OpenWhisk-Prewarm and deployed 10 functions in Alpheidae in advance. When Alpheidae was running, we set the function call interval to 20 s to keep the functions active all the time. In this case, Alpheidae identified the functions as low usage function containers. We do so to ensure that Alpheidae has resources available for reuse when a new function request arrives. Of course, Alpheidae alleviates cold-starts better if more function containers exist, indicating that there may be more function container resources that can be reused. The test results are shown in Figure 8.

![Figure 8. Comparison test of cold-start latency for functions.](image)
To ensure that the results are fair and intuitive, we schedule the functions for the same type of node and record the cold start time of each function application. In the testing process, we first divide the selected nine servers with the same configuration into three groups equally and deploy the kubernetes system uniformly, in which each group of servers is set up with one master node and two node nodes. Next, we deploy OpenWhisk-Prewarm, OpenWhisk-Prewarm-Disabled, and Alpheidae on each of the three groups of servers. Then, we select a function to be loaded into each of the three platforms for the first call and record the cold-start time of the function. After the function naturally dies out, we repeat the call again and record the cold-start time. Finally, we perform the above operation a total of 10 times, and the average statistics are shown in Figure 8. From the test results, we can see that, when OpenWhisk-Prewarm responds to more than four concurrent requests for functions, it leads to a greater cold-start delay (when the number of concurrent requests is five, the cold-start delay of one function significantly increases, while when the number of concurrent requests increases to nine, the number of functions with an increase in cold-start delay becomes five). The reason for this situation is that the number of functions requested to be called exceeds the size of the prewarm container pool, resulting in the inability to obtain sufficient prewarm containers from the prewarm container pool to alleviate the cold start. Of course, we can solve this problem by setting a larger prewarm container pool, but doing so will consume more resources and make the problem “unsolvable”. Alpheidae does not have such problems because it does not need to use additional resources to alleviate the cold-start problem, and it decreases the cold-start time by 90% compared to OpenWhisk-Prewarm-Disabled. When OpenWhisk has the “Prewarm” option enabled, it is as good or better than Alpheidae if the prewarm container pool is set large enough. In contrast, Alpheidae performs better, and at least it does not use additional system resources.

6.5. Testing of Resource Reuse Solutions

In the test system Alphidae, we conducted two separate sets of tests to observe the impact of functional resource reuse on existing functions in the container:

- The idle function container testing: In the initial stage, the function container stops requesting calls after a period of frequent calls. Then, when Alphidae discovers that the resources of the function container have been reused by other functions, it initiates a new request for the original function in the container. What we need to do is observe whether there is a significant change in the cold-start latency of the original function or even if the request for the function fails.

- The low usage function container testing: We use a call every 20 s to reduce the frequency of function container usage. When Alphidae discovers that the resources in the container have been reused by other functions, it begins to increase the QPS of the existing functions in the container. We also observe whether there is a significant change in the cold-start latency of the original function in the container.

The cold-start latency for the idle function container resource reuse scheme is shown in Figure 9.

Figure 9 depicts the scenario in which four idle functions initiate call requests again. In the first 60 s, the system continuously initiates function call requests, and in the following 60 s, we stop the function calls to ensure that System Alphidae can recognize the functions as idle function containers and has the opportunity to allow the remaining resources in the container to be reused by other functions. After 120 s, we once again initiated call requests for the original four idle functions. From the statistical results, we can see that the startup latency of the four functions did not significantly increase, they are still within an acceptable range, and there were no instances of function request failures. This outcome well demonstrates the feasibility of our designed idle function container resource reuse scheme.

Figure 10 depicts the scenario where four functions in a low usage state initiate call requests again. In the first 10 min, we initiate call requests for these low usage functions
every 20 s, ensuring that System Alphidae recognizes these functions as low usage function containers, and the remaining resources in the containers can be reused by other functions. After 10 min, we performed a large number of request calls on these functions again. From the results of the test statistics, it can be seen that the low usage function container resource reuse scheme that we designed can still effectively alleviate the cold-start latency of the functions.

Figure 9. The statistical results of cold-start latency for idle function container resource reuse scheme.

Figure 10. The statistical results of cold-start latency for low usage function container resource reuse scheme.
7. Discussion

Cold starts have always been the most concerning issue in the application process of serverless computing, and the industry has been continuously improving and innovating around this difficulty. Our research approach in solving the problem is to alleviate cold starts by improving resource utilization. In fact, this approach introduces into the research another challenge for serverless computing regarding how to manage container resources, and it uses the concept of “stealing” to temporarily reuse the remaining resources inside the container. Although we have verified the feasibility of this solution through a series of tests, there are still some areas for discussion in our work due to limitations in the testing conditions and limited research results.

- We found through container monitoring that resource management inevitably leads to resource waste in two situations: one is that the reusable resources in the container are relatively small and may not be able to meet new function reuse requests, and these remaining resources continue to exist until the container dies; and the other is that reusable resources have been provided for new functions, there are still some redundant resources in the container, but these resources are relatively small and can no longer be reused, so the accumulated fragmentation resources are wasted as the number increases. For a node, the resources that it can provide are limited. If resource waste cannot be resolved, the system may fail to respond to services due to insufficient resources. In our design scheme, the remaining resources in the container are reused for a short period of time only to alleviate the cold start. After the reused function image is successfully started, the reused function traffic is forwarded to the newly established function image container, and the reused function is released again. This series of system operation processes can be completed in a few hundred milliseconds or seconds. In view of the above two kinds of resource waste, our next research work will focus on fragmentation resource management and provide cold-start alleviation services for more new function calls.

- When setting the idle time threshold, we consider the situation of a general function container; for example, we set the threshold to 30 s in our research. In theory, due to the different types of functions, the setting of time thresholds should also be different. If there are few applied functions in the actual environment, we can manually set the time threshold. However, when there are multiple types of functions, this approach significantly increases labor costs. Moreover, if there is a conflict between this static configuration and the actual operation of the function, it still needs to be resolved manually, which entails a large workload and a time-consuming process. Therefore, our next research plan is to dynamically set the time threshold based on the actual running status of the function, which can quickly and accurately identify the idle function container.

- When there are no container resources in the container pool that meet the function’s needs, the function will be forced to perform a cold start. To alleviate the cold-start problem in this extreme situation, we will explore solutions for automatic scaling of the prewarm container pool based on the situation of an idle function container pool, a low usage function container pool, and waiting for start function. Of course, prewarming the container pool may require additional resources. We will combine the first part of the discussion to convert the remaining unused resources into new container resources waiting for reuse. We will do so to find a good balance between resource utilization and cold start. If this approach still does not provide sufficient resources for the function, we may consider using vertical scaling to solve the problem.
8. Conclusions

The occupancy of container resources and the cold start of function services are key to evaluating serverless performance, and our work focuses on how to reduce serverless cold-start time based on these two key metrics. Therefore, we used function container resource reuse to create an Alphidae platform that can alleviate the cold start of serverless computing, while preventing function services from using more system resources. Alphidae performs the classification of container resources based on the memory usage, CPU usage, and QPS of functions, and it counts the reusable container resources from the identified idle container pool and low usage container pool. When a new function call request arrives, we redesign a function wrapper in Alphidae that incorporates nesting techniques (namespace) and resource limiting techniques (cgroups), encapsulating the remaining container resources into a securely isolated container space. Considering the resource usage requirements of new functions during the initial runtime, Alphidae can dynamically match suitable container resources for new function calls according to the nearest similarity rules of the CPU and memory resources, thus achieving a fast response to cold starts. The test results based on real systems validate the feasibility of the cold-start solution that we designed, Alphidae can reduce dependency on the additional container resources of the system without affecting the operation of existing functions, and if the platform does not support the open container prewarm function, or the number of prewarm containers set cannot meet the call of new functions, Alphidae can still provide cold-start services. Compared to other open-source serverless platforms, Alphidae can improve the ability to alleviate cold starts by 90% under extreme conditions (if the platform does not support the container prewarm feature), and its cold-start time can be reduced to less than 20 ms for general function calls.

Author Contributions: Conceptualization, B.L., Y.Z. and S.R.; data duration, Y.Z.; formal analysis, B.L.; investigation, B.L., Y.Z. and S.R.; methodology, B.L., Y.Z. and S.R.; supervision, Y.Z.; validation, S.R.; writing—original draft preparation, B.L.; writing—review and editing, Y.Z. and S.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China, under Grant No. 82230065, The Key Research and Development Program in the Shaanxi Province of China, under Grant Nos. 2023-YBSF-319 and 2023-YBSF-258, and the Fundamental Research Funds for Central Universities, under Grant No. XJS221203.

Data Availability Statement: Due to privacy restrictions, the test data of the paper cannot be made public.

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

References


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.