Tackling Cold Start in Serverless Computing with Multi-Level Container Reuse

Amelie Chi Zhou[*], Rongzheng Huang[†], Zhoubin Ke[†], Yusen Li[‡], Yi Wang[†], Rui Mao[†]
[*]Hong Kong Baptist University [‡]Nankai University
[†]Guangdong Provincial Key Lab of Popular High Performance Computers, Shenzhen University

Abstract—**In Serverless Computing, function cold-start is a major issue that causes delay of the system. Various solutions have been proposed to address function cold-start issue, among which keeping containers alive after function completion is an easy and commonly adopted way in real serverless clouds. However, when reusing warm containers for function warm starts, existing systems only match functions to containers with the same configurations. This greatly limits the warm resource utilization. Our analysis of real-world applications reveals that many serverless applications share the same operating system and language frameworks. Thus, we propose *multi-level container reuse* that tries to reduce the startup latency of functions using "similar" containers to greatly improve warm resource utilization. Due to the complexity of selecting the best container reuse solutions, we designed a Deep Reinforcement Learning (DRL) based scheduler to efficiently and effectively address the problem. Moreover, we released a new serverless benchmark named FStartBench that contains detailed package information for comparing the effectiveness of different function cold-start methods. Experiments based on FStartBench show that, given a warm resource pool with fixed size, our DRL-based scheduler can achieve up to 53% reduction on the average function startup latency compared to state-of-the-art solutions.**

I. Introduction

When running an application with the emerging serverless infrastructure, users only need to focus on the applications by defining *functions* that contain the business logic of the applications, while the infrastructure will automatically provision and manage cloud resources as needed for the functions. Currently, all major cloud platforms are providing serverless computing services, such as Amazon Lambda [1], Microsoft Azure Functions [2] and Google Cloud Functions [3]. Serverless frameworks such as OpenWhisk [4] and OpenFaaS [5] have also been utilized to support serverless applications in private cloud or local clusters.

Serverless functions usually execute in a sandbox (e.g., a container or a virtual machine) instantiated according to user-defined execution environments. Typically, a function instance is started on receiving a service request. The startup of the instance contains several steps, including creating and launching the sandbox, fetching and installing the required codes and runtime, as well as runtime and function initialization, after which the function can execute in the sandbox. Since functions are usually short-running (e.g., milliseconds to seconds [6]), the startup overhead in serverless computing is negligible. Reducing the startup overhead of functions is a key challenge in serverless computing [7], [8], [6].

Func.	OS	Lang.	Runtime	Img Size (MB)
F1	Debian	Python	Flask+ Numpy	202
F2	Debian	Python	Flask	124
F3	Debian	Python	Flask+ Numpy+ Pandas	246
F4	Debian	Java	SpringBoot	313
F5	Alpine	Go	Gin	15

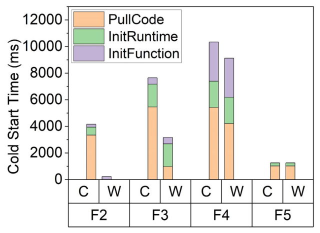


Fig. 1: Startup time breakdown of functions F2 to F5 when there is a warm container of F1. C and W represent two different ways of reusing the warm container.

One common way of mitigating this overhead is to reduce the chance of cold starts by taking advantage of "warm" function instances through pre-warming [9], [7], [10] or keep-alive [6]. For example, AWS Lambda keeps an idle instance alive for 5-7 minutes to avoid the cold-start penalty [11]. However, keeping functions warm requires additional cloud resources and makes it more costly to host serverless platforms. To improve the utilization of warm cloud resources, existing studies mainly focus on accurate workload prediction and optimized selection of alive containers for function invocations. The effectiveness of these methods rely on the likelihood of warm functions being invoked again in the near future. However, according to the Azure serverless workload [12], around 19% of the functions were only invoked once, and thus are not able to benefit from the keep-alive methods.

How can we make the alive functions more efficiently utilized? To answer this question, we studied the utilization of warm containers more in-depth with a simple example. Consider five functions selected from existing serverless benchmarks as shown in Figure 1. After F1 is executed, we keep the container warm and invoke functions F2 to F5 individually using OpenWhisk. Figure 1 shows the startup time breakdown of the four functions with two different ways of reusing the warm container of F1: 1) the warm container is used only when the same function is invoked (denoted as C and all F2-F5 have cold starts); 2) the warm container is always adopted and the invoked functions pull and install missing packages as needed before execution (denoted as W). Results show that "W" can accelerate the startup of functions by up to 14x compared to "C". A large portion of the cold start overhead in "C" is spent on pulling codes, although the required codes may already exist in the warm container (e.g., F2). Unfortunately, most existing clouds adopt the same way as "C" to reuse warm resources, which motivates us to revisit the current warm start solutions to serverless computing.

In this paper, we propose to reuse *different but similar* containers for function warm start. By analyzing popular containers on Docker Hub [13], we found that many real-world applications share the same Operating System (OS) and Language framework. For example, among the top-1000 most popular images, four most popular base (OS) images consume 77% of the total pull counts (details in Section III). This means, we can achieve better warm resource utilization when allowing to reuse containers with the same OS and/or language packages as a function invocation. However, it is non-trivial to efficiently reuse warm containers considering the large number of similar containers and diverse function request patterns.

Given a fix-sized warm resource pool, our goal in this paper is to find optimal warm container reuse solution for function invocations, such that the average startup latency of all functions is minimized. To address this problem, we proposed Multi-Level Container Reuse (MLCR) to reduce the solution space. Specifically, all packages in a container are grouped into three categories, including OS, language and runtime. When matching a function to warm containers, packages in the same category are compared as a whole to reduce the complexity of warm container matching. We further designed a Deep Reinforcement Learning (DRL) based algorithm to effectively and efficiently obtain optimal container reuse solutions. Existing serverless benchmarks do not provide function package information and thus are not suitable for comparing the effectiveness of different methods addressing cold start problem. To fill this gap, we propose FStartBench, which contains 13 functions and seven sets of serverless workloads revealing three important metrics that can distinguish different function reuse solutions. Our experiments on FStartBench show that our DRL-based method can obtain up to 53% reduction on the average function start latency compared to state-of-the-art comparisons.

This paper makes the following contributions.

- To improve the resource utilization of warm containers, this paper proposes *Multi-Level Container Reuse (MLCR)* based on the observation that there exist popular packages commonly used in different real-world applications.
- To address the multi-level container reuse problem *efficiently* and *effectively*, we designed a DRL-based container scheduler to minimize the total startup latency of a stream of functions given a fix-sized warm resource pool.
- To enable fair evaluation of different solutions to function cold-start issue, we designed a new serverless benchmark named *FStartBench*. Evaluations based on FStartBench have demonstrated the effectiveness and efficiency of our proposed method. The benchmark is open-sourced at: https://github.com/DistMinds/FStartBench.

II. Background and Motivation

A. Cold Starts in Serverless Computing

With the serverless infrastructure gets more popular in cloud computing, the cold start issue of serverless functions has also caught great attention. Existing studies have shown that the cold start issue can bring significant performance degradation to function-based services [7], [14], [6], [15].

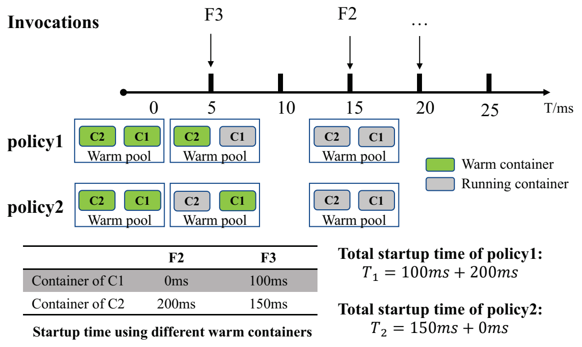
A cold start can happen when a function is invoked for the first time. When the cold start happens, the startup overhead is spent on creating and launching containers, pulling and installing the required codes and runtime, as well as runtime and function initialization. To demonstrate the importance of cold start issue, we run the applications from open-source benchmarks [16], [17] on Tencent Cloud [18] and obtain the following observations. *First*, the cold startup latency is not neglectable compared to function execution time (e.g., 1.3x-166x of function runtime). *Second*, the code pulling time occupies the largest portion of the entire cold startup latency (e.g., 47%-89% in our evaluation). Thus, how to efficiently cache the downloaded codes and runtime with limited cloud resources is thus a key problem to study. *Third*, the function initialization time varies from application to application. It is low for functions based on interpreted languages such as Python and Node (e.g., 6%), but can greatly increase for functions based on compiled languages such as Java and .NET (e.g., 45%). These features make it much complicated to efficiently address the function cold-start issue.

B. Existing Solutions to Cold Starts

Existing studies have proposed various ways to mitigate the startup overhead for serverless computing.

Some studies have proposed to reduce the cold start time of functions by designing *light-weight* containers, virtual machines or unikernels [8], [14], [19], [20], [21]. Although these methods can effectively reduce the creating and launching time of sandboxes, they cannot help mitigating other parts of the cold start overhead such as code pulling which takes a big portion of the overall startup latency. Some other studies proposed to reduce the impact of cold starts by taking advantage of "warm" function instances. For example, Shahrad et al. [12] proposed to *pre-warm* containers according to workload characteristics before a function invocation and thus making it a warm start. The effectiveness of pre-warming on reducing cold start overhead highly relies on the accuracy of workload prediction. *Keep-alive* is another commonly used technique which has been adopted as the default warming up option in both commercial clouds [1], [2] and open-source serverless platforms [4]. Such systems simply keep a container warm after its execution for a fixed amount of time. Some advanced studies have proposed to decide the duration of keeping alive according to the features of functions (e.g., frequency of invocations, initialization overheads and resource footprints) [6]. Due to the diversified requirements on resources, it is hard if not impossible to find a generally good keep-alive policy for different workloads.

The above-mentioned studies mainly focus on reusing containers of the same function, which can be inefficient considering that many functions are invoked infrequently (e.g., over 40% of the functions in Azure workload are invoked no more than two times within one day [12]). Recently, a



	F2	F3
Container of C1	0ms	100ms
Container of C2	200ms	150ms

Startup time using different warm containers

Total startup time of policy1: $T_1 = 100ms + 200ms$

Total startup time of policy2: $T_2 = 150ms + 0ms$

Fig. 2: An example of different function start solutions

few studies have investigated how to share warm containers across different functions. Suo et al. [9] proposed to reuse container runtime using a resource pool managed according to configurations of function requests and available containers to optimize the overall performance of serverless applications. Li et al. [22] proposed a container management scheme enabling container sharing between different functions without security issue. Although these studies demonstrate the feasibility and advantages of inter-function container sharing, finding a good sharing plan under limited resources remains challenging.

C. Motivations

Reusing warm containers to address function cold start problem is non-trivial and sharing resources across functions further complicates the problem. *Existing studies only share resources across functions when the warm container has all required packages of the invoked function [9], [22].* However, this is not good enough considering the complexity of real serverless workloads. According to Azure workload [2], over 40% of the functions are invoked no more than two times within one day, meaning the occurrences of functions requiring the same images in real serverless environments may be low.

In this paper, we propose to share warm containers across *different but similar* functions to increase the possibility of resource reusing. For example, as shown in Figure 1, if we reuse the warm container of F1 for a cold start of F2, we can greatly reduce the startup time by eliminating the PullCode overhead. However, how to reuse containers across functions *effectively* and *efficiently* is a challenging task.

Challenge 1: Effectiveness. Given a pool of warm containers, we can easily find a container that gives the most reduction on estimated startup time for a function invocation. However, to optimize the overall performance for a stream of function invocations, it is not always a good idea to select the warm container that gives the most time reduction. Figure 2 shows such an example. Consider a serverless platform with a warm resource pool to speedup function starts. Initially, the pool contains two warm containers C1 and C2. The table at the bottom of the figure shows the function startup time using different warm containers. An invocation of function F3 arrives at 5ms and there are three options for the startup of F3, i.e., cold start, warm start using C1 and warm start using C2. The best-effort Policy1 selects the warm container C1 for F3 that generates the lowest startup time for F3. However, when an invocation of F2 arrives, Policy1 has to adopt C2 to start F2. The total startup time of Policy1 is thus not optimal compared to Policy2, which considers all function invocations as a whole to obtain the best startup solution.

Challenge 2: Efficiency. As shown in the example above, selecting a good warm container reuse solution for a stream of function arrivals is non-trivial. Especially when inter-function container sharing is enabled, the solution space of the container reuse problem has become prohibitively large. Consider a stream of N function invocations and a pool of M warm containers. The search space for finding a good solution is $O(M^N)$ in the worse-case. On the other hand, most serverless functions are short-running. For example, 50% of the functions in Azure workload have execution time shorter than 1s on average [2]. Thus, it is important to obtain good container reuse solutions in a timely manner in order not to hinder the execution of the functions.

In summary, we propose to share warm containers across different but similar functions for better resource utilization. However, how to obtain a good inter-function container reuse solution in a timely manner is a key research problem.

III. Design Rationale

The complexity of inter-function container reuse comes from the fact that a large number of *similar* containers can be used for function warm start. This dramatically increases the solution space compared to existing studies [22], [9] that only share containers among functions with the same configurations (e.g., requiring the same image). To achieve the goal of efficiently obtaining good container reuse solutions, we need to address a key design issue: *how to effectively reduce the solution space without sacrificing too much solution quality?*

A function image usually contains many packages. The typical number can range from several to several hundreds. Through studying the popular images from Docker Hub [13], we found that the packages in an image can usually be classified into three levels, including operating system (OS), language and runtime (e.g., Figure 1). A *level* can be viewed as an abstract section of an image that contains multiple packages. Further, we observed some commonly used OSes and languages among functions, although the runtime may differ. For example, Figure 3 shows pull counts of the top-1000 most popular images on Docker Hub. Among all base images (i.e., OS), a few are much more popular than others, such as Ubuntu, Alpine, Busybox and Centos. Language packages such as Python, Openjdk and Golang are also much more popular than others. This means, in real serverless environments, many applications may share the same OS and language packages, while having different runtime for application-specific purposes.

The above observation motivates our proposal of **multi-level container reuse**, which is specially designed to address the effectiveness and efficiency challenges.

- We classify all packages in an image into three levels (i.e., OS, language and runtime) and perform level-by-level

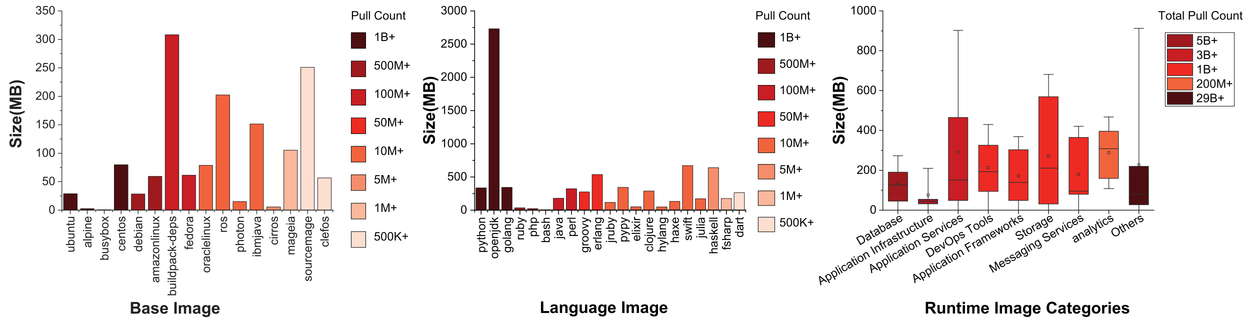


Fig. 3: Top-1000 most popular images from Docker Hub

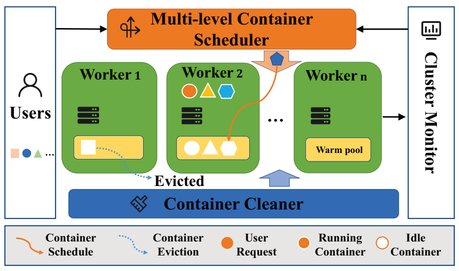


Fig. 4: System Overview

matching between a function and a warm container to reduce the complexity of finding a good container reuse solution (better efficiency).

- Inter-function container reuse allows warm starts at different levels. Due to the existence of highly popular packages at the OS and language levels, it is possible to obtain better chances of container reuse (better effectiveness).

A. *System Overview*

We briefly introduce our overall design. Consider a serverless platform with a *fix-sized* resource pool for warm containers to accelerate the start of continuously arriving function invocations. Our goal is to minimize the overall startup time of all arrived functions. The key of achieving this goal is to find a good overall *function to warm container (if exists) mapping* for all functions. Figure 4 gives an overview to our system.

Container scheduler. Our system is built on a cluster of workers, where each worker has a reserved memory space for the warm resource pool. Users continuously submit requests to the system with different function configurations. When receiving one request, we first need to start a container, either from the warm pool (warm-start) or by creating a new container (cold-start), and then schedule the function to the container for running. Finding a good startup solution for all function invocations is a complicated problem. The overall function startup time can be affected by many factors, including function configurations, arrival patterns, similarities between function images, etc. To achieve effective and efficient optimization for function starts, we proposed a Deep Reinforcement Learning (DRL) based multi-level container scheduler to solve the problem (details in Section IV-B).

Container eviction. After a container finishes the execution, it is put back to the warm resource pool to mitigate cold starts. Since the warm pool has limited resources, it is important to consider the container eviction problem to make good usage of the warm resources. A simple method for container eviction is the Least Recently Used (LRU) strategy. It has also been widely used in other cache-based systems [23], [24], [25], [26], [27]. Existing keep-alive studies [6], [10] have proposed various advanced methods to address container eviction. Since this problem is not our main focus, we simply adopt LRU for container eviction in this paper. Specifically, LRU is triggered when a container needs to be kept alive while the warm pool has reached its capacity.

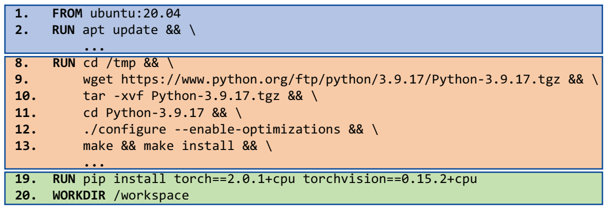
Container cleaner. Inter-function container sharing can lead to security problems. The code and user data that are stored persistently in a container can lead to information leakage if the container is reused by a different application. To address this problem, we designed a container cleaner which protects user data through volume mount/unmount. Specifically, when a container is created, volumes that persist data generated and used by applications are mounted to the container. Coping with the multi-level container reuse, volumes are classified into three groups, namely language volumes, runtime package volumes and user data volumes. Note that the OS package is written on the container writable level other than a volume. Container cleaning is performed when a warm container is selected to be reused. The container cleaner takes two steps to repack a warm container: 1) unmount the private package volumes from the warm container; and 2) mount the required package volumes from function database. Volume managing such as mount and unmount on running containers can be realised via tools such as podman [28].

In the following section, we introduce details of our multi-level container scheduler design.

IV. Multi-level Container Reuse

A. *Container Management*

To enable multi-level container reuse, we manage functions/containers according to their configurations/packages. For example, a function configuration can be represented using three lists $\{L_1, L_2, L_3\}$, where L_1 contains OS packages, L_2 contains language related packages and L_3 contains runtime related packages. Each list can contain more than one package. Figure 5 shows a concrete example of representing a container using three package levels. Specifically, the example shows a



```
1.   FROM ubuntu:20.04
2.   RUN apt update && \
         ...
8.   RUN cd /tmp && \
9.       wget https://www.python.org/ftp/python/3.9.17/Python-3.9.17.tgz && \
10.      tar -xvf Python-3.9.17.tgz && \
11.      cd Python-3.9.17 && \
12.      ./configure --enable-optimizations && \
13.      make && make install && \
         ...
19.  RUN pip install torch==2.0.1+cpu torchvision==0.15.2+cpu
20.  WORKDIR /workspace
```

Fig. 5: An example of categorizing packages into three levels, namely OS (blue), language (orange) and runtime (green).

TABLE I: Possible matching of function F and container C

Expression	Notation
$F.L_1 \neq C.L_1$	no match (cold start)
$F.L_1 = C.L_1$, $F.L_2 \neq C.L_2$	L_1 match
$F.L_1 = C.L_1$, $F.L_2 = C.L_2$, $F.L_3 \neq C.L_3$	L_2 match
$F.L_1 = C.L_1$, $F.L_2 = C.L_2$, $F.L_3 = C.L_3$	L_3 match (full match)

real dockerfile used in deep learning applications [29]. Line 1 indicates that the base image is ubuntu:20.04. Lines 8-13 install Python3 related packages. Line 19 installs other application-specific packages, such as the PyTorch machine learning library and the torchvision package specifically designed for computer vision tasks. There are different ways to obtain the package levels from configuration files, either with additional tags specified by users or using advanced image analysis techniques [30]. In this work, we rely on predefined tags given by users or experts. It is our future work to design an automated way for package categorization.

When matching a function invocation with warm containers, we compare the three lists individually and Table I shows all possible matching results. Consider, for instance, a function with a configuration file as depicted in Figure 5. A warm container utilizing the "ubuntu:20.04" base image offers a L_1 match for the function. Meanwhile, a container that encompasses both the "ubuntu:20.04" base image and "python-3.9.17" library provides a L_2 match. Note that, our matching method naturally achieves pruning. For example, when the OS level of a function invocation does not match with the OS level of a warm container, we consider it as "no match" and will not continue with the L_2 and L_3 comparisons. This is based on the prior knowledge that, when the OS needs to be reinstalled in a container, all language and runtime packages also need to be reinitialized. Thus, the benefit of reusing such a container is likely to be minor. The pruning helps us to greatly reduce the overhead of container matching.

With level-by-level container matching, given a function invocation, we are able to quickly find a container that gives the most function startup time reduction from a number of warm containers in the pool. However, achieving optimal container reuse is still challenging considering the diverse function configurations and unpredictable function arrivals. A simple greedy algorithm can hardly obtain *good* and *fast* scheduling decisions for short-running and continuously arriving function requests. In this paper, we adopt the more powerful Deep Reinforcement Learning (DRL) algorithm to solve the complicated problem.

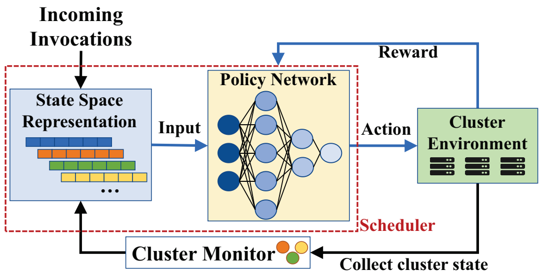


Fig. 6: DRL-based Container Scheduler

B. DRL-based Container Scheduler

DRL is a machine learning method that has shown great power on solving various decision making problems [31], [32]. Given the dynamic and unpredictable nature of serverless workloads, DRL provides a promising approach to adaptively optimize container reuse policies based on real-time feedback. Thus, MLCR is designed to adapt to various workloads by continuously learning and does not rely on workload prediction.

Among different DRL models, we choose Deep Q-Network (DQN) which strikes a balance between simplicity and efficiency. The goal of a DQN *agent* is to learn an optimal policy for making decisions in an *environment*. The agent aims to maximize the cumulative *reward* it receives over time by learning to estimate the expected future rewards (Q-values) for different *actions* in different *states*. Through a process called Q-learning, the agent iteratively updates its Q-value estimates based on the observed rewards and transitions experienced during interactions with the environment. Figure 6 shows the overall process of our DRL-based container scheduler. We first clarify the main DRL concepts in our problem setting.

Environment. The environment refers to the serverless platform where functions are submitted and a resource pool with fixed size is given to maintain warm containers. A LRU-based evictor in the environment makes sure that the memory consumption of warm containers does not exceed the pool size.

Agent. An agent is a decision maker, which in our problem refers to the multi-level container scheduler that makes function startup decisions for continuously arriving functions.

State. The state needs to uniquely describe an environment. In our problem, it needs to reflect the workload characteristics such as function arrival and system states such as warm pool capacity. Workload-related states include the OS, language and runtime packages of each function, as well as the function arrival interval to help the DRL model learn temporal characteristics of the workload. System-related states include both container-wide and cluster-wide information. The container-wide information includes the package information of the container, status of the container (e.g., idle, busy and waiting), the start time and duration of the container. Many existing methods such as random forest tree [31] can be used to filter out the most important features. Cluster-wide information includes global states such as the number of warm containers and the remaining capacity of the warm resource pool.

Action. We define *action* $= \{a_1, a_2, ..., a_{n+1}\}$, where n

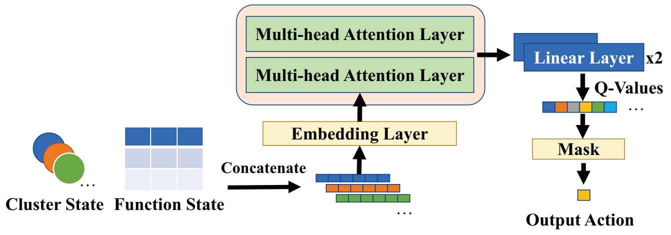


Fig. 7: Policy Network Design

is the maximum number of containers in the warm pool. a_i ($i \in [1, n]$) refers to the scheduling of a function to the i-th container in the pool and a_{n+1} refers to creating a new container for the function (i.e., cold start). If i is larger than the actual number of warm containers in the environment or the i-th container is busy, it also means cold start.

Reward. We set the reward according to the objective of our scheduler, namely $r_t = -lt$ where lt is the startup latency of the function when scheduled using action a_t.

Model Details. As shown in Figure 7, we first concatenate cluster and function states into a single vector representation, which is then normalized. The concatenated vector is passed to an embedding layer with size $(256, 512)$. The purpose of this step is to increase the dimensionality of the vector, enhancing the representation of data features. We then use a multi-head attention layer with size $(dims = 512 * 2, heads = 2)$ to learn the features. This module has better learning capabilities compared to RNN or LSTM according to our previous experiments. Finally, a linear layer with size $(512, action_{num})$ is used to output the value for each action, and the action with the highest value is selected as the output action after filtering out bad decisions using masking.

Training Process. The DRL model is trained using the offline training data mentioned in FStartBench. In each training iteration, invocations are repeatedly scheduled, with the state s_t being input to the model to generate action a_t. The environment executes action a_t, and the state transitions to the next state s_{t+1}, with the corresponding reward r_t. The model is updated by sampling a batch of experiences (s_i, a_i, r_i, s_{t+1}) from the experience pool E and applying the DQN algorithm for updates. Note that the experience pool can be circularly utilized in multiple rounds, as the DRL model is trained progressively and will generate different actions for the same state in different rounds. Algorithm 1 provides a full description of the model training procedure.

C. *Optimizing DRL for MLCR*

Due to the special challenges of multi-level container reuse (MLCR), we propose two optimizations to improve the effectiveness and efficiency of DQN for our problem.

First, as indicated by the example in Figure 2, making the best overall container reuse decisions relies both on the quality of the current decision (action to current environment) and also the awareness of overall function arrival patterns (knowledge of the future). This poses high requirement on the learning ability of our DRL model.

Algorithm 1 Training of DRL Model

```
Ensure: The well-trained DRL model
1:  Initialize replay experience $E$ to capacity $N$;
2:  Initialize the *Network* with random parameters $\omega$;
3:  repeat
4:      Initialize the cluster environment *Env*;
5:      for each $t \in$ *invocations* do
6:          $s_t \leftarrow$ Current cluster environment state;
7:          $a_t \leftarrow$ With probability $\epsilon$ select a random action $a_t$, otherwise $a_t$ = *Network*$(s_t)$;
8:          $s_{t+1}, r_t \leftarrow$ Apply action $a_t$ to the environment *Env*$(a_t)$;
9:          Save experience $(s_t, a_t, r_t, s_{t+1})$ to $E$;
10:         $(s_i, a_i, r_i, s_{i+1}) \leftarrow$ Randomly sample a batch of experiences from $E$;
11:         Update parameters $\omega$ in *Network* using $(s_i, a_i, r_i, s_{i+1})$ according to the gradient descent algorithm;
12:     end for
13: until The training process converges;
```

To help our model successfully capture the temporal features inherent in the invocations to make best decisions, we introduce two *multi-head attention* layers [33] in our policy network to help the network focus on relevant features and learn hidden relationships between the inputs. Multi-head attention is more commonly used in natural language processing (NLP) due to its ability to learn sequential features well. Output of the multi-head attention layers, namely transformed representations of the input states, are then input to two linear layers to map the learned representations to the Q-values for each possible action.

The second challenge is that, for large systems, the number of warm containers could also be large. This leads to a large action space and can usually cause slow convergence of the DRL training process. To overcome this issue, we introduce a *mask* after the linear layers. Instead of selecting the best action with the maximum Q-value, the mask can utilize prior knowledge to filter out manifestly erroneous decisions and select the best action from the remaining subset of actions. For example, if a warm container is "no match" (according to Table I) with the function invocation, this action should not be output or even explored. In this way, we not only prevent the DQN from engaging in purposeless exploration, thus accelerating the model training, but also can improve the overall performance of the model.

Overall, the multi-head attention layers and mask layer help to improve the effectiveness and efficiency of our DRL model.

V. Benchmark Design

Existing studies have proposed a number of benchmarks [16], [34] and open-source production traces [2] to evaluate the effectiveness of various system optimizations for serverless applications. For example, FunctionBench [34] provides a publicly available FaaS workload suite containing various workloads that can be deployed on cloud platforms, exposing the performance of different types of workloads on different platforms. ServerlessBench [16] is another benchmark for characterizing the performance of serverless platforms. The popular Azure trace [2] exposes the characteristics of various workloads in real production environments.

Although the above benchmarks and datasets provide rich data and analysis results that can inspire researchers to optimize serverless platforms, not enough details of the workloads are revealed for the evaluation and comparison of different function cold-start solutions. For example, the Azure workload uses function IDs to differentiate different functions, while the detailed information of the functions, such as image size and number of packages, are not exposed. Real-world serverless applications are complicated, which usually consist of many cooperating serverless functions with different images, arrival patterns, etc. As shown in Section II, existing algorithms tackling function cold-start problems may perform differently on different types of workloads. Thus, there is a need for new serverless benchmarks that contain detailed information of functions to reveal the impact of different serverless cold-start solutions. To fill the gap, we present *FStartBench*.

Function composition. FStartBench contains a suite of 13 real-world use cases covering five types of popular serverless applications. As shown in Table II, FStartBench includes 1) five simple Hello functions to compare the startup latency of different programming languages, 2) three data analytics functions to evaluate the impact of package dependencies to function startup latency, 3) a C++-based function that communicates with the object storage service on the cloud to represent network-intensive applications, 4) a CPU-intensive function performing simple arithmetic computations and 5) three functions representing popular cloud applications in web, image processing and machine learning.

The 13 functions are chosen from a wide spectrum of real-world applications. For example, Function 10 is the compute-intensive ALU application chosen from ServerlessBench [16] that performs arithmetic operations. Function 13 mimics a real-world AI application that performs image classification using a pre-trained model deployed on Tensorflow. Model inference results are sent back to users utilizing an Flask-based user interface. Although 13 functions cannot cover all kinds of serverless applications, we believe they are representative enough to evaluate the effectiveness of different warm start solutions. We will continue working on FStartBench to make it more extensive. Besides, the OS, language and runtime packages of the 13 functions are selected from the popular images on Docker Hub. For example, Alpine, Debian and CentOS are within the top 5 most pulled base images according to Figure 3. We create images for the 13 functions and deploy them on the Tencent Serverless Cloud Function (SCF) [18] to obtain their execution traces for the benchmark. All images and analysis of the functions are open-sourced on Github [35].

Workload composition. We combine different functions to compose the workloads, which mimics the real-world serverless platforms that have multiple types of applications concurrently running in the system. When composing the workloads, we adopt Poisson distributions to simulate function arrivals, following established practices [22]. We also create workloads using patterns of real Azure functions [2]. We create seven sets of workloads using functions in Table II according to three important metrics to distinguish different function reuse solutions.

Metric 1: Function similarity. A complex serverless application usually consists of many collaborating functions and cloud services. The effectiveness of reusing warm containers depends on similarities of functions. We define the similarity of two functions using the Jaccard similarity coefficient. That is, denote the sets of packages in two functions as $P1$ and $P2$, the similarity of the two functions is calculated as $|P1 \cap P2|/|P1 \cup P2|$.

TABLE II: Functions in FStartBench

FuncID	OS	Language	Runtime	Description
1	Alpine	Java	Springboot	Hello
2	Alpine	Nodejs	Express	Hello
3	Alpine	Go	Gin	Hello
4	Alpine	Python	Flask	Hello
5	Debian	Python	Flask	Hello
6	Debian	Python	Flask + NP (Numpy)	Data analytics
7	Debian	Python	Flask + NP + Pandas	Data analytics
8	Debian	Python	Flask + NP + Pandas + Matplotlib	Data analytics
9	CentOS	C++	\	Communication
10	Debian	Python	Flask	Simple arithmetic
11	Alpine	Nodejs	Express	Web service
12	Alpine	Java	Springboot	Image processing
13	Debian	Python	Flask + Tensorflow	Machine learning

We created two workloads denoted as “LO-Sim” (low similarity) and “HI-Sim” (high similarity), each of which includes 300 functions. For LO-Sim, the functions are picked from function types 1, 2, 5, 9 and 13 as shown in Table II. The resulted average similarity of all function pairs in the workload is 0.29. For HI-Sim, the functions are picked from function types 1, 2, 3, 4 and 11, and the resulted average similarity is 0.52. The arrival time of each function follows the Poisson distribution parameterized by λ.

Metric 2: Package size. Except the similarity between function packages, the size of packages also matters. Existing studies have shown that the memory usage of functions in real applications vary in a 4X range [2]. Given limited memory resources for storing warm containers, package size has great impact on the effectiveness of warm container reuse.

We created two workloads denoted as “LO-Var” (low package size variance) and “HI-Var” (high package size variance), each of which includes 300 functions. For LO-Var, the functions are picked from function types 1, 2, 5, 9 and 13. For HI-Var, the functions are picked from function types 1, 2, 3, 4 and 11. We calculate the variance using the sizes of all packages in the workload. That is, denote the set of packages in function type i of the workload as P_i, the variance of LO-Var is calculated as $Var(P_1 \cap P_2 \cap P_5 \cap P_9 \cap P_{13})$. As a result, the variances of workloads LO-Var and HI-Var are 54 and 769, respectively. The larger the variance, the harder to achieve good warm container reuse. The arrival time of each function follows the Poisson distribution parameterized by λ.

Metric 3: Function arrival. Functions arrive at serverless platforms under different rates and patterns. As described in existing studies [2], the inter-arrival time between functions vary from application to application, which makes it difficult to accurately predict the future arrival of functions.

To capture this feature of real serverless applications, we create three workload traces with different function arrival frequencies and patterns, namely “Uniform”, “Peak” and

"Random". For each workload, we trigger 300 functions picked from function types 1, 2, 5, 6 and 13. All functions arrive within a 6 minute period. For Uniform, there are 50 function invocations every minute and the functions evenly arrive at the system. For Peak, we interchangeably switch between high-workload and low-workload periods, each of which last one minute. During high and low periods, there are 80 and 20 function invocations arrived evenly per minute, respectively. For Random, there are 50 function invocations every minute and the arrival time of each function follows the Poisson distribution.

VI. EVALUATION

In this section, we evaluate the effectiveness and efficiency of our proposed Multi-Level Container Reuse (MLCR) approach using three sets of experiments. Specifically, in Section VI-B, we begin by evaluating the overall effectiveness of MLCR on reducing the startup latency of functions. In Section VI-C, we demonstrate the effectiveness and efficiency of MLCR across metrics with different workloads. Finally in Section VI-D, we analyze the overhead of our approach.

A. Experimental Setup

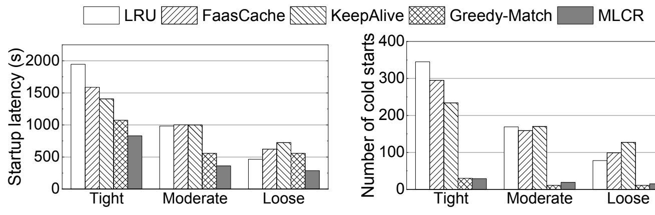
Workloads. All evaluated functions are selected from the FStartBench benchmark described in Section V. It should be emphasized that our benchmark contains applications from various fields, and we combine different functions to mimic the scenario of many real-world applications running concurrently in the serverless platform.

For the overall evaluations (SectionVI-B), we aggregate all 13 functions and execute them for 400 times in total. Each function arrives at the system following a Poisson distribution. The λ parameters of the distributions vary from 0 to 5 invocations per second randomly. We set the capacity of the warm pool to *Tight*, *Moderate*, and *Loose*, where Loose is set to the peak memory size of all running containers in the cluster, while Tight and Moderate are set to one-fifth and one-half of the size of Loose, respectively.

For the benchmark evaluations (Section VI-C), we adopt the three sets of workloads composed to follow different patterns (details in Section V). For each workload, we vary the size of the warm resource pool from 25%, 50%, 75% to 100% of Loose and evaluate the performance of different methods.

Comparisons. We select four representative warm-start approaches from existing work to compare with MLCR.

- *LRU* keeps finished containers in the warm pool. If the pool is already full, the Least Recently Used (LRU) idle container will be evicted to make space for the newly added container.
- *FaasCache* [6] is similar to LRU. The difference is that, when the warm pool is full, FaasCache calculates a priority (based on system optimization goals) for containers and evicts the one that has the minimal priority. The priorities are calculated according to different factors, including the invocation frequency, running time, and memory usage of each function.



(a) Total startup lantency (b) Cold start

Fig. 8: Total startup latency and cold starts of functions under different pool sizes

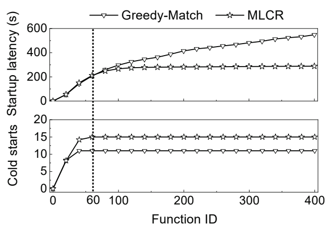


Fig. 9: Startup latency and cold starts under *Loose* pool

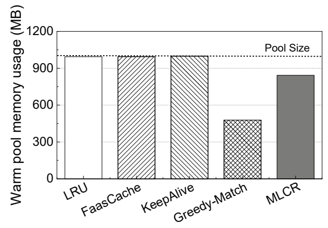


Fig. 10: Warm resource consumpt. under *Loose* pool size

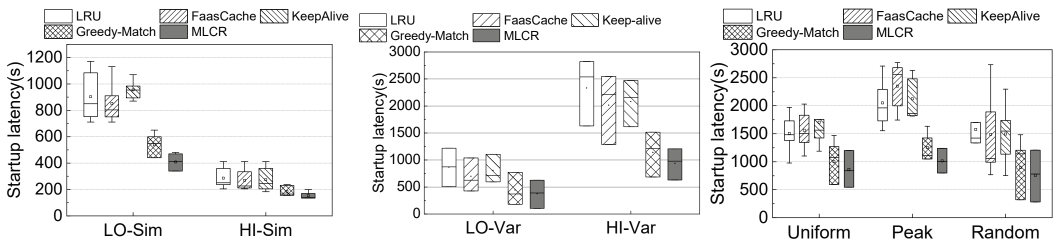
- *KeepAlive* represents the widely-used warm start mechanism in current public clouds, which keeps finished containers warm for 10 minutes in the pool. When the pool is full, we reject the keep-warm requests of newly finished containers as a simple and practical approach.
- *Greedy-Match* also adopts the multi-level container reuse idea like MLCR. The difference is that it uses a greedy strategy to schedule a function to the warm container that gives the best matching result according to Table I. Similar to MLCR, it also adopts the LRU to evict idle containers when the warm pool is full.

Testbed. We deploy MLCR in a simulator implemented based on OpenWhisk [4] to have more controlled evaluations. We repeat all experiments for 50 times and report the average results for each evaluation. All experiments are performed on a server with 1TB RAM, an Intel(R) Xeon(R) E5-2650 CPU and a NVIDIA Tesla V100 GPU. The operating system is Ubuntu 18.04.5 LTS.

B. Overall Evaluation

Figure 8 presents the overall startup latency of the 400 function invocations and the number of cold starts obtained by the compared methods. We have the following observations.

First, Figure 8a shows that, MLCR obtains the lowest function startup latency among all comparisons under different pool sizes. Specifically, MLCR reduces the total startup latency by 38%-57%, 47%-53%, 48%-52% and 22%-48% compared to LRU, FaasCache, KeepAlive and Greedy-Match, respectively. When the pool size increases from Tight to Loose, the total startup latency obtained by different comparisons all decrease. This is obvious since more warm resources can allow more warm starts, as evidenced by Figure 8b. Comparing different methods, MLCR obtains the highest latency reduction when the pool size is Tight. This is because when the



(a) Function similarity (b) Package size variance (c) Function arrival pattern

Fig. 11: Total startup latency results on different workloads from FStartBench benchmark

warm resources are limited, it is more important to utilize the resources smartly. In addition, Greedy-Match performs well since it also uses warm containers with multi-level matching.

Second, when looking at Figure 8b, it is clear that Greedy-Match and MLCR result in much less cold starts compared to the other three methods. This means, our multi-level container matching can effectively increase the warm resource utilization. As Greedy-Match always schedules a function to the currently best warm container, it is likely to utilize the warm resources to the utmost. Thus, Greedy-Match obtains even less cold starts than MLCR. However, less number of cold starts does not necessarily guarantee lower startup latency and the greedy scheduler of Greedy-Match may lead to a local optimal solution. This is especially so when the pool size is Loose, meaning that there is a large space for the scheduler to search. Figure 9 gives a detailed comparison between Greedy-Match and MLCR when the pool size is Loose. It shows the change of total startup latency and number of cold starts along the arrival of functions. At the arrival of Function 50, Greedy-Match selected a container C for function warm start (1.8s) while MLCR decided to have a cold start (3.8s). Although it seems that Greedy-Match outperforms MLCR at the moment, container C was later reused by MLCR for the warm start of another function which reduced the startup latency from 62.9s to 57.1s compared to Greedy-Match. Thus, the total startup latency of MLCR is 3.8s lower than that of Greedy-Match. This means, our DRL-based scheduler can better capture the function arrival patterns and make better scheduling decisions to minimize the total startup latency.

We further study the warm resource usage of the compared methods. Figure 10 shows the peak memory consumption of the warm resource pool obtained by the compared methods under Loose pool size. LRU, FaasCache and KeepAlive consume all the resources to store warm containers and trigger 28, 57 and 20 times of evictions in total, respectively, to add new containers to the pool. In contrary, thanks to the efficient multi-level reuse of warm resources, MLCR and Greedy-Match do not need to exhaust all resources from the warm pool. Greedy-Match consumes the lowest memory resources among all comparisons, due to its greedy way of reusing warm containers. Although MLCR consumes more memory resources, it leads to the lowest total function startup latency.

Overall, the results show that our multi-level container matching is effective on improving the utilization of warm resources given a fix-sized resource pool. However, the multi-level container matching cannot work well without the help of our DRL-based container scheduler. For example, when the pool size is Loose, Greedy-Match, which can be considered as LRU plus multi-level container matching, obtains even higher total startup latency than LRU. This is because the greedy scheduling method cannot find a good container reuse plan under the large solution space of Loose. By combining both techniques, MLCR obtains the best latency results among all.

C. Benchmark Evaluation

In this subsection, we evaluate the effectiveness of MLCR under different workload features, including different function similarities, package size variances and arrival patterns (details in Section V). For all evaluations, we plot box charts to show the distribution of the results under different pool sizes.

1) Function Similarity: Figure 11a illustrates the startup latency results obtained by different comparisons for HI-Sim and LO-Sim workloads. Functions in the two workloads are composed to have different similarities on packages.

Overall, all compared methods obtain lower startup latency under HI-Sim compared to LO-Sim. This is because HI-Sim contains more similar functions and thus makes it more feasible for container reuse. When comparing different methods, it is worth noting that MLCR obtains higher startup latency reduction compared to other comparisons when the function similarity is low (i.e., LO-Sim). For example, MLCR reduces the average startup latency by 16% and 24% compared to Greedy-Match for HI-Sim and LO-Sim workloads, respectively. This is because when the similarity is low, the resulted startup latency from two different container reuse solutions may differ a lot. Thus, it is more important to use an optimized scheduler to find a good solution.

2) Function Package Size: Figure 11b illustrates the startup latency results obtained by different comparisons for LO-Var and HI-Var workloads. The workloads are composed to have functions with different package size variances.

Overall, all compared methods obtain lower total startup latency under LO-Var compared to HI-Var. This is because when package sizes are almost the same (i.e., LO-Var), it is easier to efficiently utilize the limited warm resource pool to store more containers. Under HI-Var, MLCR is able to greatly reduce the total function startup latency compared to the other methods, demonstrating its ability to make good warm container reuse plans while being workload aware.

3) Function Arrival Pattern: Figure 11c shows the startup latency results obtained by different comparisons under Uniform, Peak and Random function arrival patterns. Detailed function composition can be found in Section V.

Overall, all compared methods obtain the highest total startup latency at the Peak workload. This is because the function arrival time is less predictable in Peak, thus causing difficulty for the compared methods to obtain good container reuse solutions. MLCR consistently outperforms the other compared methods under all function arrival patterns. Specifically, it reduces the total startup latency by 40%-57%, 26%-60%, 46%-62% and 12%-31% compared to LRU, FaasCache, KeepAlive and Greedy-Match, respectively. Among the three workloads, MLCR obtains the highest latency reduction under Peak, meaning that our DRL-based container scheduler can effectively improve the workload awareness of MLCR to help find a good solution.

Summary: MLCR has demonstrated effective warm container reuse under different function similarities, package sizes and function arrival patterns. Especially, it greatly outperforms state-of-the-art comparisons for complicated workload patterns, such as LO-Sim, HI-Var and Peak. This makes MLCR extremely suitable for real cloud platforms, where the diversity of serverless applications results in low function similarity, varied function sizes and unpredictable function arrivals [2].

Note that, the good performance of MLCR under different function features does not rely on accurate workload forecasting. Instead, we employ DRL to analyze and learn from historical arrival patterns of workloads in an offline environment. In addition to offline training, the DRL model also supports online fine-tuning to adjust model parameters accordingly. This method allows DRL to make more informed and effective decisions during the online operational phase.

D. System Overhead

Compared to KeepAlive, which is widely deployed in existing serverless platforms, MLCR introduces additional overhead due to the DRL-based multi-level container scheduler. Although the training of DRL models is usually time-consuming (e.g., over ten hours), the training is performed one-time offline. The trained model can then be used to make real-time decisions during runtime. Using one V100 GPU, the inference time (i.e., making one reuse decision) is only 3-4ms. Each scheduling decision made by the DRL scheduler results in a notable reduction in startup latency, ranging from tens of milliseconds to several seconds. The model can also be fine-tuned at runtime to adapt to workload/system changes. This adaptation process is typically lightweight and does not significantly impact system performance. Thus, MLCR is very suitable for short-running serverless applications.

VII. Related Work

Existing Cold-Start Solutions. Function cold-start is an important problem for serverless computing and a number of solutions have been proposed to address the issue. To reduce the startup time of individual functions, existing studies have designed *lightweight* containers, virtual machines and unikernels to effectively cut down sandbox creation and launching time [8], [14], [19], [20], [21]. Although these studies are effective on reducing the startup latency of sandboxes, they require redesign of serverless applications based on designed sandboxes and thus can restrict their application in practice. Shahrad et al. [12] put forward the concept of *pre-warm* containers, which are prepared based on workload predictions. Following their work, a few other solutions [36], [37] have been proposed to advance the pre-warming technology.

Some studies emphasize on container reuse across various functions as a strategy to mitigate cold starts. Suo et al. [9] advocated for a container runtime reuse strategy, managed by a resource pool that adjusts based on function requests and available container configurations. However, in their work, only containers with exactly the same configurations as required by functions will be reused. Li et al. [22] proposed to reuse containers across different but similar functions. Specifically, it proposed the concept of zygote containers, which may contain packages of multiple functions. For example, if a zygote container has *all* packages required by a function A and that of a function B, it can be reused by both functions. Compared to [22], MLCR has several advantages: 1) MLCR does not require the existence of all required packages in a warm container (or a zygote container) for it to be reused; 2) by grouping packages into three levels, we can perform container matching more efficiently; 3) the DRL-based container selection is effective and robust to workload/environment changes.

Scheduling using Deep Reinforcement Learning (DRL). Reinforcement learning (RL) has recently been adopted to address the function cold-start issue. For example, Vahidinia et al. [10] proposed to use RL to discern function invocation patterns over time, aiming to determine the necessary prewarmed containers. However, in situations where the state space is vast, continuous, or high-dimensional, traditional RL methods can encounter challenges. To tackle these challenges, DRL, which incorporates deep learning techniques, has been identified as an effective solution for scheduling and resource allocation issues in distributed systems [38], [39], [40], [32]. For instance, Yu et al. [38] used DRL to derive optimal strategies for resource harvesting and re-assignment in serverless computing systems. Similarly, Wang et al. [39] applied DRL techniques to dynamically invoke functions in a serverless environment for distributed machine learning. Li et al. [40] implemented DRL for the strategic placement of prolonged containers in expansive computing clusters. Mao et al. [32] employed DRL in orchestrating DAG job scheduling. Despite the wide adoption of DRL in various scheduling problems, our DRL-based scheduler contains unique optimizations, including the multi-head attention and mask layers, specifically designed for the multi-level container matching problem.

VIII. Conclusion

Function cold-start is a significant issue in serverless computing and has great impact on system latency. In this paper, we address the problem with a novel multi-level container reuse approach. We developed a DRL-based scheduler to efficiently choose the best container reuse solutions amidst the complexity of the problem. To facilitate the evaluation of different function cold-start methods, we introduced FStartBench, a new serverless benchmark encompassing detailed package information. Results obtained from experiments utilizing FStartBench have shown a reduction in average function startup latency by up to 53% compared to existing solutions. As future work, we plan to extend this paper along two directions. First, we will expand the functions in FStartBench to cover more real-world applications and improve its usability. Second, we will design automatic tool to facilitate the level-wise package classification for efficient container matching.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (62172282, 62272252, 62072311, 62122056, U2001212 and U23B2040), Guangdong Natural Science Foundation 2022A1515010122, Shenzhen Science and Technology Foundation (RCYX20221008092908029, RCJC20221008092725019, JCYJ20210324094402008).

References

[1] "AWS Lambda," https://aws.amazon.com/lambda/, 2023, accessed Jan 2024.

[2] "Azure Cloud Functions," https://azure.microsoft.com/products/functions, 2023, accessed Jan 2024.

[3] "Google Cloud Functions," https://cloud.google.com/functions, 2023, accessed Jan 2024.

[4] "Apache OpenWhisk: Open Source Serverless Cloud Platform," https://openwhisk.apache.org/, 2023, accessed Jan 2024.

[5] "OpenFaaS - Serverless Functions Made Simple," https://www.openfaas.com/, 2023, accessed Jan 2024.

[6] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ASPLOS '21*, 2021, pp. 386–400.

[7] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *HotCloud '19*, 2019.

[8] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Sock: Rapid task provisioning with serverless-optimized containers," in *USENIX ATC*, 2018, pp. 57–70.

[9] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling cold start of serverless applications by efficient and adaptive container runtime reusing," in *CLUSTER '21*, 2021, pp. 433–443.

[10] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," *IEEE Internet of Things Journal*, pp. 1–1, 2022.

[11] "Cold Starts in AWS Lambda," https://mikhail.io/serverless/coldstarts/aws, 2023, accessed Jan 2024.

[12] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC*, 2020, pp. 205–218.

[13] "Dockerhub - docker image repository," https://hub.docker.com/, 2023, accessed Jan 2024.

[14] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS '20*, 2020, p. 467–481.

[15] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *USENIX ATC*, 2018, pp. 133–146.

[16] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *SoCC'20*, 2020, p. 30–44.

[17] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *SoCC'19*, 2019, p. 477.

[18] "Tencent Serverless Cloud Function," https://cloud.tencent.com/document/product/583, 2023, accessed Jan 2024.

[19] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *USENIX ATC*, 2018, pp. 923–935.

[20] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[21] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *NSDI '20*, 2020, pp. 419–434.

[22] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing," in *USENIX ATC'22*, 2022, pp. 69–84.

[23] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *SIGMOD '93*, 1993, p. 297–306.

[24] Y. Wang, Y. Yang, C. Han, L. Ye, Y. Ke, and Q. Wang, "Lr-lru: A pacs-oriented intelligent cache replacement policy," *IEEE Access*, vol. 7, pp. 58 073–58 084, 2019.

[25] K. Cheng and Y. Kambayashi, "Lru-sp: a size-adjusted and popularity-aware lru replacement algorithm for web caching," in *COMPSAC2000*, 2000, pp. 48–53.

[26] B. Jiang, P. Nain, and D. Towsley, "On the convergence of the ttl approximation for an lru cache under independent stationary request processes," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 4, sep 2018.

[27] E. Friedlander and V. Aggarwal, "Generalization of lru cache replacement policy with applications to video streaming," *ACM TOMPECS*, vol. 4, no. 3, pp. 1–22, 2019.

[28] "Podman is a daemonless, open source, Linux native tool." https://docs.podman.io/en/latest/markdown/podman-volume-unmount.1.html, 2023, accessed Jan 2024.

[29] "Pytorch docker images," https://github.com/cnstark/pytorch-docker/, 2023, accessed Jan 2024.

[30] H. Ye, J. Zhou, W. Chen, J. Zhu, G. Wu, and J. Wei, "Dockergen: A knowledge graph based approach for software containerization," in *COMPSAC*, 2021, pp. 986–991.

[31] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "Drlpart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *HPDC '21*, 2021, p. 175–188.

[32] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *SIGCOMM '19*, 2019, p. 270–288.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[34] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *CLOUD'19*, 2019, pp. 502–504.

[35] "Fstartbench benchmark," https://github.com/DistMinds/FStartBench, 2023, accessed Jan 2024.

[36] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *ASPLOS '22*, 2022, pp. 753–767.

[37] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *SoCC*, 2021, pp. 122–137.

[38] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating serverless computing by harvesting idle resources," in *WWW'22*, 2022, pp. 1741–1751.

[39] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019*. IEEE, 2019, pp. 1288–1296.

[40] S. Li, L. Wang, W. Wang, Y. Yu, and B. Li, "George: Learning to place long-lived containers in large clusters with operation constraints," in *SoCC*, 2021, pp. 258–272.