2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00018

Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing

Kun Suo*, Junggab Son*, Dazhao Cheng†, Wei Chen‡ and Sabur Baidya§
*Kennesaw State University, †University of North Carolina at Charlotte
‡Automotive Vehicle Group, Nvidia, §University of Louisville
Email: {ksuo, json}@kennesaw.edu, dazhao.cheng@uncc.edu, weich@nvidia.com, shbaid01@louisville.edu

***Abstract*—During the past few years, serverless computing has changed the paradigm of application development and deployment in the cloud and edge due to its unique advantages, including easy administration, automatic scaling, built-in fault tolerance, etc. Nevertheless, serverless computing is also facing challenges such as long latency due to the cold start. In this paper, we present an in-depth performance analysis of cold start in the serverless framework and propose HotC, a container-based runtime management framework that leverages the lightweight containers to mitigate the cold start and improve the network performance of serverless applications. HotC maintains a live container runtime pool, analyzes the user input or configuration file, and provides available runtime for immediate reuse. To precisely predict the request and efficiently manage the hot containers, we design an adaptive live container control algorithm combining the exponential smoothing model and Markov chain method. Our evaluation results show that HotC introduces negligible overhead and can efficiently improve the performance of various applications with different network traffic patterns in both cloud servers and edge devices.**

***Index Terms*—Serverless, cold start, cloud, performance**

I. Introduction

As the traditional market of cloud computing turns mature and user requirement for microservices keeps growing, serverless computing, such as Amazon Lambda, Microsoft Azure Function, and Google Cloud Function, which provides high performance, high scalability, built-in availability, and fault tolerance, is becoming increasingly popular in the public clouds [35]. The serverless infrastructure allows developers to focus on the application and business logic itself instead of worrying about where to deploy their codes and how to tweak a large number of servers. For example, one traditional website can be decomposed into hundreds of microservices, which are small and efficient packaged functions [13] and can be scaled up rapidly for growing demands.

In order to execute the source code that users created or uploaded, a serverless system first allocates resources to the functions based on user-defined configurations and sets up triggers from other cloud services or events. When target events occur, the system creates containers, loads codes or functions inside, executes them, and finally returns the results. The serverless functions only consume CPU, memory, or other resources at runtime. However, such a design might also introduce performance loss due to the cold start, especially to those I/O-intensive applications. For instance, Amazon has reported that every 100 ms of latency costs them 1% in sales [5] and the page speed of websites is also treated by Google as one of the major ranking factors [16].

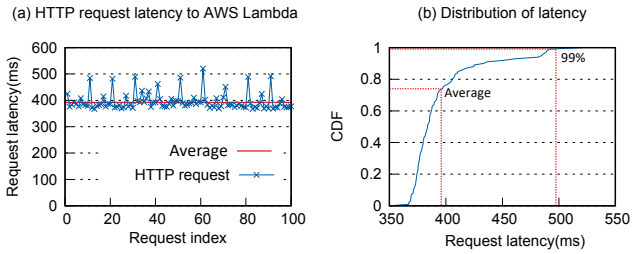

Fig. 1: The analysis of AWS Lambda requests. (a) HTTP request latency of 100 samples. (b) Latency distribution.

The cold start issue has been widely existing in computer systems for decades. For instance, a three-way handshake that requires both the client and server to exchange synchronization and acknowledgment packets has to be processed before each TCP connection. All HTTPS requests have to prepare security certification during accessing the webserver. However, different from the above, cold start in serverless infrastructure has its own characteristics and will significantly degrade the system performance and user experience. First, Function as a Service (FaaS) is designed to replace traditional local functions which can be called at any time and respond immediately. However, the cloud function needs to deploy a computing runtime including container startup, code download, runtime initialization, business logic initialization, etc., which increases the latency to millisecond or even second level. Second, a survey revealed that 75% of cloud functions execute for less than 10 seconds [27]. As the FaaS platforms usually charge based on the length of the request, the cold start might incur unnecessary costs for the users. Last, the serverless infrastructure is flexible to expand and contract based on current requests. When the amount of requests drops, the system will clean up and recycle resources while making the cold start periodically happen and the application performance unstable and unpredictable.

Figure 1(a) illustrates the latency of requests to AWS Lambda. We implemented a backend in Python generating a random number for the request. The client sends one request every second to the backend through API Gateway and lasts for 10 seconds. Then the client waits for 30 minutes and repeatedly executes the above. We identify whether the code

978-1-7281-9666-4/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00018

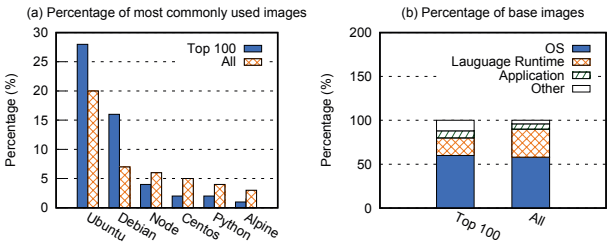

Fig. 2: Dockerfile statistics from Github.

is running on is a new one or an existing one by creating a temporary file in the container and checking whether that file already exists. As shown in the figure, the very first one in every 10 requests has the longest request latency due to the cold start in the Lambda architecture. The highest request latency is about 41.8% and 31.7% longer than the lowest and average latency, respectively. Different from the stable request on local functions where 99% of latency is almost the same, we observed significant long tail latency issues in the serverless architecture due to the cold start, as shown in Figure 1(b).

Another characteristic of serverless computing is that the packaged functions have high similarities and many of them execute in the same kind of container runtime which includes the OS image, programming language environment, network configuration, etc. For instance, Microsoft has revealed that about 40% of key jobs or services at Bing search rerun periodically [7]. We analyzed thousands of Dockerfiles from GitHub projects. As Figure 2(a) shows, both the top 100 popular and all surveyed projects are dominated by a few commonly used images, which mostly contain similar OSes, language runtimes, etc., or their combination. Figure 2(b) illustrates the OS, language, and application related configurations that dominate the majority setting of base images. Therefore, there exists a *semantic gap* between the cold start issue and the inefficient utilization of the highly similar container runtimes. Resource reuse has been widely adopted in industries and academic research to improve system performance. For instance, Android saves the metadata and JIT-compiled code onto disk and reuses them in the next JVM process. Interruptible Tasks [11], HotTub [20] and Skyway [24] proposed to reuse the runtime or intermediate data to reduce memory pressure, JVM warm-up overhead, and data transfer cost, respectively.

To address the cold start issue in serverless services, our key observation is that the runtime could be reused efficiently by leveraging the lightweight containers and the homogeneity of containerized serverless applications. Inspired by that, we proposed and developed HotC, a container-based runtime management framework that provides low-latency request handling while minimizing the performance overhead to applications. In brief, our major contributions are summarized as follows:

- First, we perform an empirical performance study of the long latency due to cold start in serverless architecture and reveal its reason as well as potential impact.
- Next, we design and implement HotC, a simple and lightweight solution to reuse the runtime based on client requests. Different from existing solutions arbitrarily keeping containers alive for a certain amount of time (i.e., 15 minutes in AWS Lambda) or periodically waking up containers to keep warm (i.e., Azure Logic), HotC maintains a runtime pool and efficiently reuses the containers upon user requests. To precisely control the resource and mitigate the periodic cold start, HotC leverages the container runtime history and combines exponential smoothing and the Markov chain model to improve the prediction accuracy.
- Last, our experimental results show HotC can achieve substantial performance improvement of various applications in both cloud servers and edge devices.

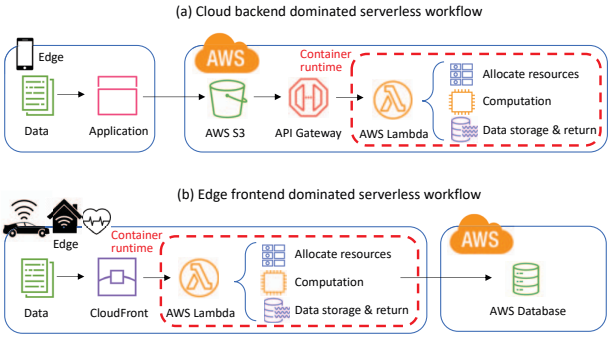

Fig. 3: Serverless application processing on edges and AWS.

The rest of this paper is organized as follows. Section II introduces the background and motivating examples. Section III presents the analysis of cold start issue and Section IV describes the system design and implementation. Experimental results are discussed in Section V. Section VI reviews the related work and Section VII concludes this paper.

II. Background and Motivation

A. How serverless computing works?

Different from the traditional applications or services relying on hardware configuration and resource orchestration, serverless computing provides a higher level of abstraction for application deployment and management. For instance, a typical Lambda application consists of three parts: *functions* – the business logic, *data* – the scenario input or output, and *events* – the interaction between the functions and data. The key part is the Lambda function, which is defined by users and then associated with the execution environment and system resources. Take Web service as an example, the traditional solution usually requires several virtual machines (VMs) to host different services and a database to store the business data. As the requests grow, it also needs a load balancer in the front to distribute the network traffic among the servers. In comparison, a serverless solution only requires Lambda functions to implement the application logic and an API Gateway to auto scale and balance the network traffic. All the resources are only consumed when it is actually needed and no backend is required to maintain.

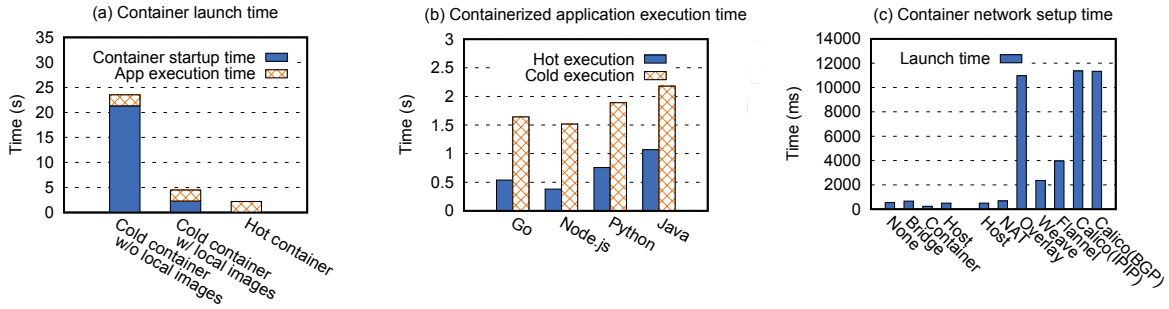

Fig. 4: Motivation examples of cold start in serverless applications.

Serverless is a native architecture based on the cloud connected to the edges. In a typical serverless scenario, the edges only send events or requests and do not require provisioning, maintaining, and administering servers for backend components such as computation, database, key-value storage, stream processing, message queueing, etc. Figure 3(a) depicts an image processing service (e.g., compression and watermark) as an example. A user first takes and uploads a picture (data in the figure) directly to the cloud through the application. AWS S3 is responsible for the temporary persistence of the images, then finds the processing function defined by the user through a gateway, and triggers the specified function. After that, the Lambda function starts to allocate resources, executes image compression and watermark processing, and finally performs persistence and returns the results.

In recent years, many users prefer to deploy the serverless at the edge, where most of the data processing is performed within the device itself. It allows running code close to the user application, thereby improving performance and reducing latency. Figure 3(b) shows the serverless processing on AWS IoT Greengrass. Take the self-driving vehicle as an example, most of the computations, including static object recognition (i.e., traffic lights, traffic signs, road intersections) and dynamic object recognition (i.e., vehicles, pedestrians, and bicycles), have to be performed on the edges in order to improve the performance and security. The data is first captured at the vehicle and then forwarded through CloudFront to the local Lambda functions. All operations of functions perform within the edge devices and only limited data will be uploaded to the cloud for persistence.

B. Why cold start happens?

Both the serverless in the cloud or at the edge might face cold start issues, which introduces inefficiency and unpredictability. A typical serverless application usually runs inside the containers and the first access or revisit after a long time will inevitably cause the cold start problem as it requires booting a new container and reallocating system resources [30], [31]. Such a cold start cannot be neglected as the serverless applications or functions are usually small and their operation time is short. Besides the above, there also exist some unique characteristics to the cold start in a serverless architecture. First, the functions are not daemon processes, which means they have to start whenever a new request comes, especially for IoT scenarios that involve little data and happen less frequently. If the function languages, e.g., *Java*, need to compile and interpret, the cold start time could be even longer. Second, as the function execution is stateless and the container runtime terminated once requests are handled, there exists no reuse for either program codes or processed data. Lastly, as serverless architecture is highly scaled, the next request could be handled on a new host distributed by the API server which makes the existing cache mechanism or the memory optimization inside a single server not effective. All the above issues either give rise to or exacerbate the cold start of serverless applications.

C. What is the impact of cold start?

Here we analyzed the potential impact of cold start in various scenarios. Details of the testbed and benchmark settings can be found in Section V. First, we measured the launch time of the container in our local servers. We wrote a program that downloads a 3.3MB pdf file from Amazon S3 and executes it inside Docker containers. As Figure 4(a) shows, the program execution time in cold start without local images is 23.5s while the cold start with local images is only 4.51s. In comparison, the execution inside a running container takes only 2.2s, which is dominated by the application execution itself while the runtime setup time can be neglected.

Next, we studied the execution time of containerized applications for various programming languages. Figure 4(b) depicts the execution time of one program which sums integer from 0 to 10 million but implemented in *Go*, *Node.js*, *Python* and *Java*, respectively. For the hot execution which the program executes in a running container, the time in *Node.js* is only 0.38s while it is 1.07s in *Java*. This is due to the fact that the *Java* program must be compiled into bytecode files and then translated and executed by the JVM. Compared to the hot execution, the cold execution which involves an additional container environment setup prolongs all cases. For instance, the execution time in *Go* with cold start is 3.06× of that in hot execution and the cold start even doubles the already long execution in *Java*.

Lastly, we also investigated the building time of various customized networks during the boot of container runtime, which is crucial to those short-lived or latency-sensitive work-

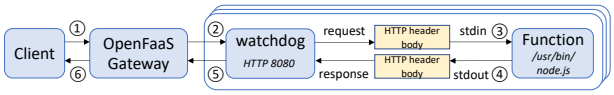

Fig. 5: Request and response in OpenFaaS framework.

loads. As Figure 4(c) illustrates, for the single host networking, the bridge mode and host mode networking are close to that without network setup (None) while the container mode networking is only half of it. This is due to the cheaper startup connecting to a proxy container instead of booting a new container. The container networking on multiple hosts is even more complicated. Compared to the host mode network, the overlay or routing solution, which involves additional registration and initialization, takes up to 23× longer startup time. The above examples clearly show the impact of cold start on the performance of containerized applications in different languages and configurations, and motivate us the adoption of hot container runtime to mitigate the cold start issue.

III. Analysis of Cold Start Overhead

A. Summary of Findings

Here we further studied the cold start latency in OpenFaaS [3], an event-driven functions and microservices platform which can be deployed both in the cloud and at edges. Other FaaS frameworks such as Knative, Kubeless, Fn, Openwhisk, etc., have similar architecture and mechanisms. Figure 5 depicts the key components and processing pipeline in OpenFaaS. The clients send requests to the gateway, which acts as an entry to the backends. Gateway works as a proxy forwarding requests to the corresponding functions and can be scaled to multiple instances. Each backend function consists of a container running two kinds of processes. The watchdog is a tiny Golang HTTP server that connects the executable containerized function in the local image with API Gateway. The watchdog puts a layer of HTTP shell on the function, writes to the `stdin` of the function process, and receives the response data from the function process `stdout`. The function process executes the user-defined handler and performs the application logic. The return value will be finally forwarded to the Gateway and received by the clients.

To precisely understand the startup overhead, we make a quantitative analysis of cold start incurred in processing serverless functions on the OpenFaaS platform. Specifically, we added timestamps in `MakeQueuedProxy` of Gateway, `main` and `pipeRequest` of watchdog to record the request latency at the gateway, watchdog, and function processes, respectively. We record six moments during the workflow path: ①: the request packet arrives the gateway; ②: the request packet reaches the watchdog; ③: the function process starts; ④: the function process stops; ⑤: the response packet sent out from watchdog; ⑥: the clients receive a response packet from the gateway. Then, we send HTTP requests to one function which generates a random number and uses *tcpdump* to capture the target packets. Compared to the function execution time and network forwarding, function initiation time (②→③) dominates the total latency. In addition, we also evaluated OpenFaaS on edge platforms such as Raspberry Pi and Nvidia Jetson TX2, and the results are much similar to the above.

B. Industry Practices

During the past few years, researchers in industry have proposed solutions to mitigate cold start in serverless applications. Engineers from Alibaba cloud found that containerized applications have to be downloaded from the warehouse and decompressed from the images before they are used and deployed. The performance is significantly affected by hardware such as network, and sudden access burst might bring network congestion and service not responding. To mitigate the cold start and provide large-scale container service ability, they proposed several optimizations including a new image format that does not need to fully download and an efficient compress algorithm. Besides, to reduce the network congestion on a single node, the Alibaba team also proposed to use a P2P network for data and image distribution. However, their design only focused on the overhead of image pulling and code download. Many other factors such as resource allocation and networking configuration can still contribute to the cold start of serverless applications.

Another growing interest in addressing cold start is to optimize the underlying infrastructure. To reduce the delay of cold start, Tencent engineers optimized their architecture in two aspects. First, they deployed their functions based on lightweight virtualization. Compared to the traditional virtual machine or containers, the cold start of applications can be reduced to 200 milliseconds in a new architecture. Second, Tencent proposed new scheduling policies for active functions. They designed a real-time autoscale system that can expand or contract in second-level based on the system metrics and monitoring data. For function prediction and preprocessing, they used periodic data analysis and machine learning to improve the accuracy. When the concurrent growth of the function is monitored, the number of instances that need to be expanded will be calculated immediately, and the scheduler quickly deploys instances to meet subsequent concurrent growth.

AWS adopts a fixed keep-alive policy that retains the resources in memory for minutes after function execution [1]. Many other open-source architectures, such as OpenWhisk, also use a similar design to keep functions warm for minutes. Although such a policy is simple and practical, it disregards actual invocation frequency and patterns of the functions, and also wastes lots of resources. As the FaaS platform usually charges based on the length of requests, regular warmup might also introduce unnecessary fees. For instance, applications with long initialization time, such as loading deep learning models, incur additional costs when cold starts happen frequently. Researchers in Microsoft Azure [27] recently proposed using different keep-alive values for workloads according to their actual invocation frequency and patterns. They allowed service providers in many cases to pre-warm a function execution just before its invocation happens. Different from their solutions controlling pre-warming and keep-alive

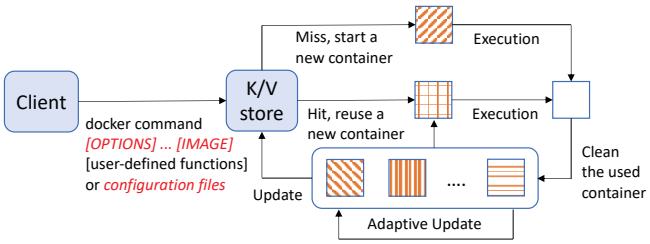

Fig. 6: HotC Architecture.

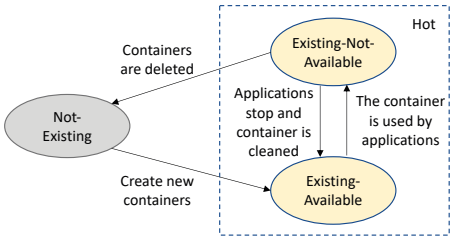

Fig. 7: Container state transition.

windows, our design adopts a time series history and a Markov model to mitigate potential cold starts of different types of container runtimes.

IV. HotC Design & Implementation

In this section, we present HotC, an approach to addressing the cold start and long latency in a serverless architecture. The key insight of HotC is to leverage lightweight containers and an efficient reuse mechanism to prevent the unnecessary cold start, excessive resource allocation, and reclaim.

A. HotC Overview

Figure 6 shows the architecture of HotC. It acts as a middleware between clients and backend servers. When new requests arrive, HotC always attempts to execute the user code in an existing and free container. If it cannot find an available container, HotC just starts a new one as usual. After the container finishes execution, it returns the results back to the client side and then HotC will clean up the container and prepare for the next request. HotC maintains a live container runtime pool and adaptively updates the pool over time. Such a design has many benefits: First, it is simple and straightforward, which does not involve disruptive changes to the existing architecture. Second, as the resource consumption mainly comes from the application execution instead of the container itself, it is lightweight to maintain a group of live containers without introducing too much overhead. Lastly, reusing the same container runtime can also offer hot cache and less translation lookaside buffer (TLB) flushing, which can significantly improve resource utilization as well as application performance.

B. HotC Key Components & Operations

Parameter Analysis. The first step of HotC is to analyze the user command or configuration file to figure out the parameter setting of the container runtime. The parameter includes container images, network configuration, UTS (UNIX Time Sharing) settings, IPC (Inter Process Communication) settings, execution options, etc. HotC treats containers with identical parameter configurations as the same type of runtime environment. HotC maintains a key value store to track the available containers. The key is the formatted parameter configurations for each container and the value is a list with container ID and state of the container. We define three states for the container: *Not-Existing* (marked as -1), *Existing-Not-Available* (marked as 0) and *Existing-Available* (marked as 1) and state transition is illustrated as Figure 7. In order to gain benefits from an existing container, it is ideal to run the application in the same runtime environment. As Algorithm 1 shows, HotC resolves the user input and looks for containers with the same runtime as candidates to reuse. If such a container exists and is available to run, we load user code into that candidate container and execute the program. Normally it might exist multiple containers with the same runtime if a large number of requests arrive. If that happens, the client just reuses the first available container. However, if the required container does not exist or all candidate containers are occupied, HotC just starts a container with that runtime setting.

Algorithm 1 Start container runtimes based on configurations

```
1: /* num_avail[key]: the available number of specific type of container runtimes */
2: function START_FUNCTION_RUNTIME(key)
3:     if runtime not existing or existing but not available then
4:         Start a new container and run the function
5:     else if existing and available then
6:         Reuse one container and run the function
7:         num_avail[key] − −
8:     end if
9: end function
```

Container Runtime Pool. To keep live containers, HotC maintains a container pool inside the memory. Take the alpine Linux container as an example, it only takes hundreds of KB for each live container. Therefore, the memory footprint of HotC is negligible through precise control of the number of containers and available system resources. In our current design, we set the maximum number of live containers to 500 and the memory usage threshold as 80% in the host. We used a heuristic method to identify the memory pressure through monitoring `used_mem` and `used_swap` in the kernel. If there exist too many containers or fewer resources, the oldest live container is forcibly terminated and releases the resources. The further overhead analysis will be discussed in Section V.

Used Container Cleanup. In order to efficiently reuse the containers, it is important to keep the used container clean. The CPU and memory resources are automatically reclaimed by the OS when the containerized applications terminate. The disk read also does not involve modifications to the Docker file system. Therefore, we only focus on the write to containers.

To keep the container clean and enable data writing, HotC assigns volume, which persists data generated and used by applications, to each container when they are created. Each live container has its unique directory to save its own data. Volume can efficiently bypass the container file system and share data between the host server and the containers. Based on that, the cleanup of to used container includes two steps: First, it deletes all files and directories in the old volumes. Second, HotC mounts new volumes to the containers for future use. To avoid resource waste and zombie files, the corresponding volumes are deleted once the containers stop execution.

C. Adaptive Live Container Management

In order to control the resource consumption and maintain high performance, we design an online prediction algorithm to adaptively control the live containers in the pool, as shown in Algorithm 3. The key idea is to combine exponential smoothing and a Markov chain method to exactly allocate resources and prepare the runtime in advance. The problem of short-term request prediction is a dynamic random process, and its trend changes frequently over time. The exponential smoothing method is adopted to predict the trend of runtime data based on the recent records while the Markov chain overcomes the data volatility. The combination of the two methods can achieve high accuracy of prediction results under dynamic workloads.

Algorithm 2 Clean up used containers

```
/* num_avail[key]: the available number of specific type of container runtimes */
function CLEAN_CONTAINERS()
    Clean the used containers and add them to pools
    num_avail[key]++
end function
```

1) Container Runtime Prediction. Whether the container runtime is reused or not is affected by many factors, such as application workload, number of users, network traffic, service location, etc. The relationship between each factor and whether the container runtime would be reused is difficult to accurately quantify with models, and the changes of each factor themselves are also relatively fuzzy. Exponential smoothing is suitable for predicting data that has no obvious trend, which calculates the exponential history value and cooperates with a certain time series model to predict future results. In HotC, we used historical data of certain types of containers to estimate their predicted number in the pool, which assigns different weights to the data in historical periods. The more recent data will be assigned with a higher weight. The number of specific type of container runtimes $e_{k_i,t}$ with configuration k_i at time t is estimated as follow:

$$\alpha * history[k_i][t] + (1-\alpha)e_{k_i,t-1} \tag{1}$$

Here, $history[k_i][1]$, $history[k_i][2]$, ..., $history[k_i][n]$ are time series data of how many specific type of container runtime

Algorithm 3 Adaptively update containers in the pool

```
/* num[key]: total number of specific type of containers;
e_{k_i,t}: expected number of specific type of container runtimes with configuration k_i at time t under exponential smoothing */
function UPDATE_POOL()
    for each key k_i do
        for 1 ≤ j < n do
            history[k_i][j-1] = history[k_i][j]
        end for
        history[k_i][n] = num[k_i] - num_avail[k_i]
        e_{k_i,t} = α Σ_{i=0}^{n-1} (1-α)^i history[k_i][t-i] + (1-α)^n e_{k_i,0}
        Calculate the predicted number E_{k_i,t} based on current state transition probability matrix e_{k_i,t} + (R_1i + R_2i)/2
        if E_{k_i,t} > num_avail[k_i] then
            Start E_{k_i} - num_avail[k_i] function containers
        end if
    end for
end function
```

is in the pool. The α represents the exponential smoothing coefficient and its range is $(0,1)$. The predict value $e_{k_i,t}$ is a weighted average of historical observations $history[k_i][t]$, and the weighted coefficient is a set of values that decay by geometric series $\alpha(1-\alpha)^i$, where the more recent one has a greater weight. The sum of weights of entire historical data $\sum_{i=0}^{n-1}\alpha(1-\alpha)^i + \alpha(1-\alpha)^n$ is 100%.

2) Determine the Parameter. The selection of the smoothing coefficient α is critical for the prediction accuracy. The smaller adoption of coefficient α is, the less influence of recent historical data will be and vice versa. According to our evaluation, when the original series shows a relatively stable horizontal trend, the coefficient α should be smaller, generally between 0.1 and 0.3. When the data series fluctuates significantly, a larger α should be used to increase the sensitivity of the model so that the prediction results can quickly keep up with the changes in historical data. In this research, we choose α as 0.8. For the initial value $history[k_i][1]$, due to multiple periods of smoothing, especially when the execution period is long, its influence is quite small. When the number of time series is more than 20, the influence of the initial value on predicted results is negligible, and the observation value of the first period can be used. When the number of time series is less than 20, the initial value has a certain influence, and the average value of the first five historical data can be used instead. Here we adopt the average of historical data as smoothed initial value.

3) Markov Prediction Modification. The exponential smoothing method is mainly used for short-term forecasting problems with fewer data and volatility. However, for serverless workload with significant random volatility, as shown in Figure 1 and 11, the prediction result is not accurate and with large relative error. To address the above, we adopt the

Markov chain, which predicts the results through the transition probability between states and can better compensate for limitations in the prediction process of exponential smoothing.

For predicted data $e_{k_i,t}$, we divide them into n region states $R_i = [R_{1i}, R_{2i}]$, where $i = 1, 2, ..., n$. The interval can be determined based on historical data $history[k_i][i]$. The state R_i will change dynamically over the time. In the Markov chain, the state transition from state R_i to state R_j can be expressed as $P_{ij}(k) = T_{ij}(k)/T_i$, $i = 1, 2, ..., n$, where T_i represented the historical data when the state is R_i and $T_{ij}(k)$ represented as the number of original data samples from R_i to R_j after k steps. Then the k-step state transition probability matrix can be expressed as follows:

$$P(k) = \begin{bmatrix} P_{11}(k) & P_{12}(k) & \dots & P_{1N}(k) \\ P_{21}(k) & P_{22}(k) & \dots & P_{2N}(k) \\ \vdots & \vdots & \ddots & \vdots \\ P_{N1}(k) & P_{N2}(k) & \dots & P_{NN}(k) \end{bmatrix} \quad (2)$$

The future state can be determined by one step on the state transition probability matrix. After the matrix is determined, the predicted state at the next moment can be inferred. Assuming the current state is R_i, then the predicted value can be calculated as the average value of the interval under the new state $e_{k_i,t+1} + (R_{1i} + R_{2i})/2$. For forecasting with large random volatility, the use of Markov chains can better revise the limitation in the exponential smoothing process. Since the prediction of the number of available containers is a non-stationary random process, the exponential smoothing method can fit the available container data to find out its changing trend, which can rectify the limitations of the Markov chain prediction process. Therefore, the combination of the two can better improve prediction accuracy and control system resource usage of live containers.

V. Evaluation

A. Experimental Settings

Our experiments were performed on a DELL PowerEdge T430 server, which was equipped with dual ten-core Intel Xeon E5-2640 2.6GHz processors, 64GB memory, Gigabit Network, and a 2TB 7200RPM hard drive. We used Ubuntu 16.04 and Linux kernel version 4.4.20 as the host OS. We also evaluated HotC on a Raspberry Pi 3, which was equipped with Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB memory, and 32GB storage. The OS was adopted by Linux Raspberrypi 4.14. We used *Docker* 1.17 for containers and the images were stored locally. We used *OpenFaaS* 0.8.5 to build serverless functions with Docker. The evaluation setting and application details of the individual case study were slightly different and further discussed in the respective sections.

B. Effectiveness of Reusing Container Runtimes

Startup and Execution Time. We first evaluated the startup time of two image recognition applications with HotC. One was implemented in *Python* and built on Google inception-v3 [33] model, which trained 1000 categories on the ImageNet

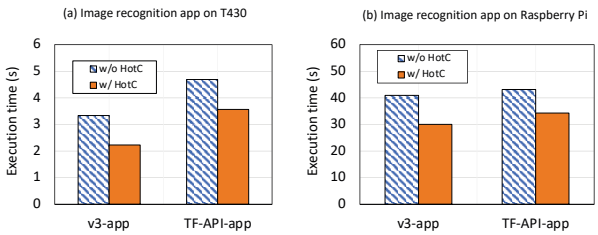

Fig. 8: The image recognition application execution time w/o and w/ HotC.

dataset (denoted as *v3-app*). Another was implemented in *Go* through Tensorflow APIs to perform image recognition (denoted as *TF-API-app*). The version of Tensorflow is 1.13. All the applications were executed inside Docker containers. The results shown were the average of ten runs. First, we evaluated the application execution time on the PowerEdge T430 server. As Figure 8(a) shows, the execution time of v3-app and TF-API-app reduced by 33.2% and 23.9% respectively compared to that without HotC. The performance improvement is due to the efficient reuse of the existing container runtime. Similarly, we also evaluated the performance of Raspberry Pi. Compared to the physical servers, Raspberry Pi has more resource constraints and is sensitive to the overhead. Here we executed the image recognition applications in overlay network containers. Compared to the physical servers, the normal execution time of the same application prolongs more than 10 times inside edge devices and makes the cold start impact less significant among the total execution time. However, as depicted in Figure 8(b), HotC still helped to reduce the execution time of v3-app and TF-API-app by 26.6% and 20.6%, respectively.

Web Application Latency. We also evaluated the application latency with HotC. We used *OpenFaaS* to build a serverless application that transferred the user input URL into QR code. The applications were implemented in different languages including *Python*, *Go*, *Node.js*, etc. For simplicity, all backend containers were configured and connected with network address translation (NAT). The clients sent requests using random configurations to the backends. As shown in Figure 9(a), the service latency is always high every time the backend involves a new container runtime setup and application execution. Based on our measurement, the URL transition only took around 60ms while the majority of time was spent on the resource allocation and container runtime setup. In comparison, Figure 9(b) depicts the service latency with HotC. The first few requests perform almost the same latency as in Figure 9(a) and it is inevitable as no existing runtime reuse is available at the beginning. However, as more requests were handled, the probability of the same type of request goes up and the following service latency, as well as the average latency, dropped dramatically. Such an improvement came from the efficient reuse of the available idle container runtime.

C. Adaptive Live Container Control

Impact on Prediction Strategy. As shown in Figure 10(a), we measure the live number of specific type containers

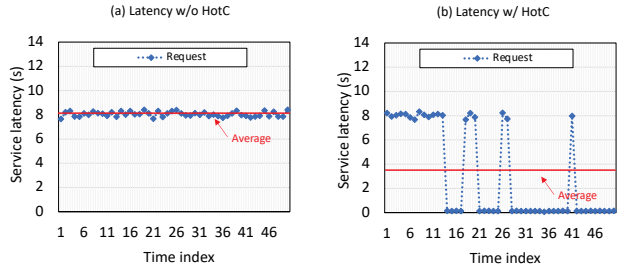

Fig. 9: The web application request latency w/o and w/ HotC.

achieved by HotC with different prediction strategies. The measurements are collected at different time intervals as the clients send requests. Compared to the real required number, the exponential smoothing method can simulate and predict the trends well. As a result, we can avoid cold container startup latency and unnecessary resource consumption at the same time. For instance, at time index from 7^{th} to 10^{th}, the relative error of prediction drops from 29% to 10% as the real request number of container jumps from 8 to 19. However, its drawback is also obvious, which forecast is relatively lagging and cannot handle large jittering. Figure 10(a) also illustrates that applying the exponential smoothing and Markov chain strategies together improves the overall prediction accuracy and matches closer with the real data. The improvement of prediction, as well as corresponding resource saving, is due to that the Markov model revises preliminary results to overcome the data fluctuation.

Sensitivity Analysis of HotC Parameters. Next, we measured the prediction result achieved by HotC over the real required number of a specific type of container and evaluated its performance for various values of parameters in Equation 1. First, we compared the small and large value of smoothing coefficient α. As depicted in Figure 10(b), the larger adoption of coefficient α is selected, the more impact the recent data has on the predicted results, even though the Markov chain amends the results. Another observation is a selection of initial values for prediction. However, too large of α will make data offset in prediction. For the early prediction, the first few predicted results are more accurate if the initial value is selected with the historical data. This is due to the fact that HotC's initial prediction is influenced significantly by the first few samples while such the impact becomes negligible with more data calculated inside the model.

D. Analysis of Request Patterns

Next, we study the benefits of HotC under different request flows. Figure 11 illustrates a collection of Youtube request from Umass Campus [4], [39]. The researchers measured the request at the campus gateway and counted statistics throughout the day. We find three representative request patterns based on the collected data. First, there exists a burst from 20 requests to 300 requests at T_{710}. Second, the request keeps decreasing in the afternoon from T_{800} to T_{1200}. Last, the throughput increases from T_{1200} to T_{1400} at night. We can

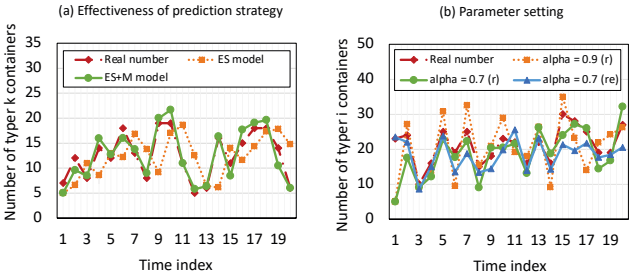

Fig. 10: Effectiveness of prediction strategy and sensitivity analysis of HotC design parameters. Here (r), (re) denote the initial value selects the random or real historical data, respectively. 'ES' and 'M' denote exponential smoothing and Markov prediction, respectively.

also observe such patterns from other cloud or edge services. Then, we evaluate the cold start issue with HotC with these request patterns. We mimic the distribution characteristics of the production workflow by running Multi-Generator [2] with smaller data size.

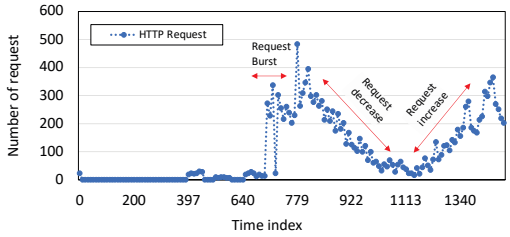

Fig. 11: Youtube request statistics of Umass. X-axis is the minute index through a day (1440 minutes).

Serial & Parallel Request. Figure 12(a) shows the serial latency achieved with and without HotC, respectively. The client is a single thread application sending the same request to the backends every 30 seconds. The experiment setting and configuration are the same as above. As depicted in the figure, every request has to start a new container runtime by default and encounters cold start latency. In comparison, after the very first request, HotC can reuse the container runtime of the previous request as all requests have the same configuration, which significantly reduces the request latency. Figure 12(b) depicts the average latency under parallel requests. Ten threads at the client keep sending requests to the backend and each thread has its own runtime configuration. The performance is unstable due to dynamic network traffic congestion and resource consumption. All latencies are pretty high due to a cold start in the default case. In comparison, after the very few requests, HotC consistently achieves better performance than that with cold starts. The average latency with HotC is only 9% of the default case.

Linear Increasing & Decreasing. Figure 13 plots the request latency of clients increasing or decreasing linearly over time. For the linear request increasing case, the clients sent two requests to the backend at the beginning, and every 30 seconds, the requests increased by two and the throughput at time index

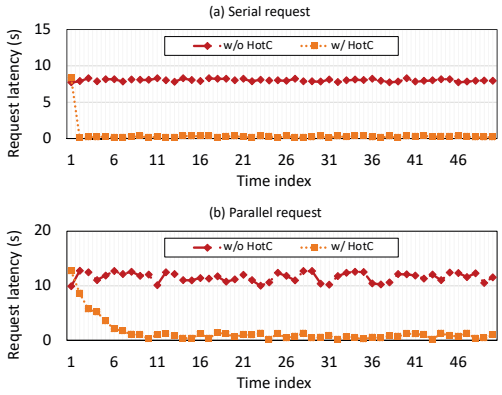

Fig. 12: Serial and parallel request latency w/o and w/ HotC.

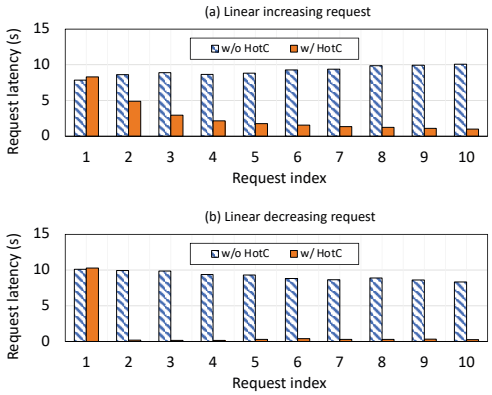

Fig. 13: Linear network throughput w/o and w/ HotC.

i is $2i$ requests per second. As shown in Figure 13(a), the latency without HotC is positive correlated with the number of requests. This is due to the fact of cold start as well as increasing network traffic congestion and resource competition. In comparison, the latency with HotC was reduced significantly. For instance, the latency is only 21% under index 5 compared to the default case. For individual requests, we observed that only part of them have a cold start each time and the rest can fully reuse the existing runtimes last time under HotC. Figure 13(b) plots the request latency under decreasing throughput over time. At the beginning, the throughput is 20 requests per second and it is reduced by two in the next round. Different from the increasing throughput case, there is always a container available if the requests keep decreasing. As a result, the request latency is always low under HotC except for the very first round of requests.

Exponential Increasing & Decreasing. We now evaluated HotC with workloads of exponential increasing or decreasing requests. As shown in Figure 14(a), we changed the number of requests to 2^i at round i to see how HotC's performance is affected by such the network flows. The latency with and without HotC both increased significantly due to the high volume of network traffic. However, we observed that different from the default case starting runtimes for each request, at least half of the requests in HotC can directly use the existing instances of the previous wave of requests, while the rest of the requests needs to wait for a new container runtime. For requests with exponential decreasing, similar to that in linear decreasing case, all following requests have hot container runtime available in the pool which will reduce the request latency dramatically.

Request Burst. Lastly, we also evaluated the performance of HotC under request burst. The client keeps sending eight requests each time and increases the throughput by 10× at the 4_{th}, 8_{th}, 12_{th}, 16_{th} round. As Figure 14(b) depicts, at the first burst, HotC can reduce the latency by around 9% through efficiently reusing the previously available containers. As time goes on, the latency can be reduced by up to 73% in the following bursts. The improvements here come from two aspects. First, there are more same types of containers available after the previous burst. Second, the Markov chain method can mitigate the random volatility of data and improves prediction accuracy. The following latency does not perform significant fluctuations and the network congestion and resource competition contribute to a slight spike of latency.

E. Overhead and Discussion

We also analyzed the overhead of HotC on resource usage. Figure 15 plots the CPU and memory usage monitoring on Raspberry Pi and physical server. First, we varied the number of live containers and measured the resource consumption. As shown in Figure 15(a), the number of live containers does not have an obvious impact on the available resource. The CPU usage increased by less than 1% (ten live containers) compared to that without containers. Similarly, the memory footprint due to a different number of live containers is also insignificant. For instance, the memory usage increased by 0.7MB for each individual live container. As discussed in Section II, the majority of resource consumption comes from the applications instead of the container itself, which left immense potential to keep live containerized runtime to address the cold start latency. We also measured the resource change during a containerized application lifecycle. As shown in Figure 15(b), we started a *Cassandra* database in one container at 6_{th} second to handle some user requests and then stopped it at 13_{th} second while keeping the container still live. *Cassandra* database is a heavy workload that executes the database on the Java virtual machine. Compared to the application resource consumption, the cold start overhead cannot be neglected during execution. Also, another observation is that the OS will automatically recycle the unused resources (i.e., memory) quickly and we did not need to worry too often about memory swapping for live containers when the memory resources are sufficient.

VI. Related Work

In this section, we review the most relevant work with regard to container startup and serverless cold start, as well as its corresponding solutions in today's cloud or edge platforms.

Accelerating Container Startup. Many researchers have revealed that the poor performance of containerized applications

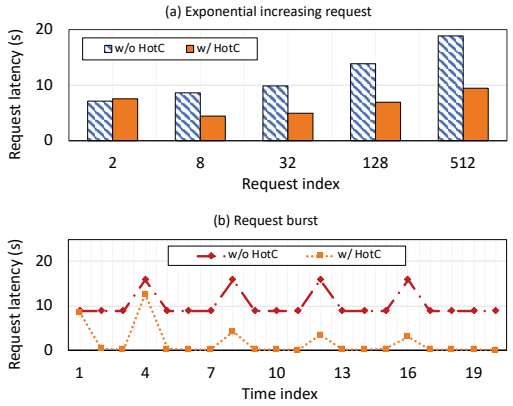

Fig. 14: Exponential and burst request w/o and w/ HotC.

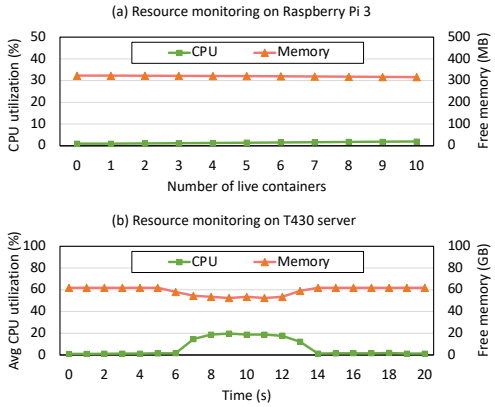

Fig. 15: The resource consumption of live containers.

suffered from the inefficiency and complexity of container startup. Harter et al. [12] reported pulling packages accounted most of container start time and proposed an optimized Docker storage driver by using backend clones lazily fetching. Akkus et al. [6] found that high startup delays of serverless applications came from isolated and separate containers and thus proposed a new serverless computing system that provides fine-grained application sandboxing as well as hierarchical message queuing and storage mechanism. Oakes et al. [25] proposed SOCK, a container system optimized in kernel scalability bottlenecks to provide speedup of the application and container initialization. Du et al. [10] modified Linux kernel based on the secure container and optimized the cold start time of the application to sub-millisecond levels. In industries, there also exist many manual solutions including periodically booting the containers, reducing artifact program size, prefetching the hot data, etc. Different from above works and inspired by JVM warm-up [20], this paper focuses on mitigating cold start through reusing the container runtime.

Resource Management and Allocation. There is a large body of work dedicated to elastic resource management [15], [17], [22], [26], [27], [29], [32], [34]. Mohan et al. [22] proposed optimizing cold start through pre-allocating virtual network interfaces that are later bound to new function containers. Wang et al. [34] proposed Replayable Execution, which uses checkpointing and sharing of memory among containers to speed up the startup times of a JVM-based FaaS system. Mohammad et al. [27] proposed reducing the number of cold starts and resource usage by predicting function invocations. Many other works proposed resource prediction and dynamic allocation for performance optimization. For instance, Kesidis et al. [17] proposed to use prediction of the demands of functions to allow providers to allocation resources for functions on containers. EMARS [26] predicts the right amount of memory for each function by tracking the function execution history.

Optimizing Container Architecture. Much effort [8], [9], [18], [19], [21], [23], [28], [38] has been dedicated to analyzing factors that affect container performance and proposing effective solutions. FreeFlow [19] presents a RDMA virtualization framework to speedup the network performance of containerized services. Zhu et al. [38] designed and implemented Slim, a low-overhead container overlay network in which packets inside only traverse the network stack exactly one time. Khalid et al. [18] reported that a container with heavy network traffic can decrease the compute resource available to its neighbors on the same server, and thus proposed a scheme, named Iron, to precisely account the consumed CPU time and enforce fair resource allocation. Other works, including the virtual routing, resource management [14], redistribution and reassignment [37], hardware offloading or bypassing the inefficient parts inside kernel [36], focus on optimizing the data path and improving container network processing. These studies are orthogonal and complementary to our work.

VII. Conclusion

This paper presented HotC, a container-based runtime management framework that leverages the efficient reuse of lightweight containers to mitigate the cold start and long latency issue of serverless applications. HotC is a simple and straightforward solution and does not need application modification. Our evaluation results showed that HotC can efficiently improve the performance of various applications with different network patterns in the cloud servers as well as edge devices. Our future direction is to evaluate the effectiveness of HotC in more complicated scenarios, such as cloudlet or high concurrency applications in a multi-cloud backend environment. For instance, in a distributed system, a few containers are extremely popular and are invoked a lot while others may not be used often. Some host machines might become overloaded and we need to consider load balancing when reusing the hot runtime. Next, small differences in the configuration file or some settings would lead to the lookup failure. We will explore adopting a subset of the available parameters as the key, and evaluate the performance which reuses an existing available or idle container with a similar configuration and applies the changes to execute the function. Lastly, we also plan to extend HotC into a more reliable architecture, e.g., adopting a distributed key-value store, to handle complex workloads.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. This work was supported by U.S. NSF grant CNS-2103459.

REFERENCES

[1] *Cold Starts in AWS Lambda*. https://mikhail.io/serverless/coldstarts/aws/.
[2] *Multi-Generator*. https://github.com/USNavalResearchLaboratory/mgen.
[3] *OpenFaaS*. https://www.openfaas.com/.
[4] *YouTube Traces from the Campus Network*. http://traces.cs.umass.edu/index.php/Network/Network.
[5] *How one second could cost amazon 1.6 billion in sales*. http://bit.ly/1Beu9Ah, 2012.
[6] 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 *2018 USENIX Annual Technical Conference (ATC)*, 2018.
[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
[8] D. Bermbach, A.-S. Karakaya, and S. Buchholz. Using application knowledge to reduce cold starts in faas services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 134–143, 2020.
[9] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.
[10] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
[11] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.
[12] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *Proceedings of the. 14th USENIX Conference on. File and Storage Technologies (FAST)*, 2016.
[13] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing(HotCloud)*, 2016.
[14] Y. Hu, M. Song, and T. Li. Towards full containerization in containerized network function virtualization. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
[15] H. Huang, J. Rao, S. Wu, H. Jin, K. Suo, and X. Wu. Adaptive resource views for containers. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.
[16] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving user perceived page load times using gaze. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
[17] G. Kesidis. Temporal overbooking of lambda functions in the cloud. *arXiv preprint arXiv:1901.09842*, 2019.
[18] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating network-based cpu in container environments. In *Proceedings of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
[19] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.
[20] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
[21] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.
[22] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.
[23] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 1–6, 2019.
[24] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
[25] 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 *Proceedings of USENIX Annual Technical Conference (ATC)*, 2018.
[26] A. Saha and S. Jindal. Emars: efficient management and allocation of resources in serverless. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.
[27] 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. *arXiv preprint arXiv:2003.03423*, 2020.
[28] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
[29] K. Suo, J. Rao, H. Jiang, and W. Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.
[30] K. Suo, Y. Zhao, W. Chen, and J. Rao. An analysis and empirical study of container networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2018.
[31] K. Suo, Y. Zhao, W. Chen, and J. Rao. vnettracer: Efficient and programmable packet tracing in virtualized networks. In *Proceedings of IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.
[32] K. Suo, Y. Zhao, J. Rao, L. Cheng, X. Zhou, and F. C. Lau. Preserving i/o prioritization in virtualized oses. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2017.
[33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
[34] K.-T. A. Wang, R. Ho, and P. Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys)*, 2019.
[35] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2018.
[36] J. Weerasinghe and F. Abel. On the cost of tunnel endpoint processing in overlay virtual networks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, 2014.
[37] Y. Zhang, Y. Li, K. Xu, D. Wang, M. Li, X. Cao, and Q. Liang. A communication-aware container re-distribution approach for high performance vnfs. In *Proceedings of IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.
[38] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim:{OS} kernel support for a low-overhead container overlay network. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.
[39] M. Zink, K. Suh, Y. Gu, and J. Kurose. Watch global, cache local: Youtube network traffic at a campus network: measurements and implications. In *Multimedia Computing and Networking*. International Society for Optics and Photonics, 2008.