Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing

Shijun QIN*, Heng WU, Yuanjia XU, Yuewen WU, Wenbo ZHANG
Institute of Software
Chinese Academy of Sciences
Beijing, China
{ qinshijun16, wuheng, xuyuanjia2017, wuyuewen, zhangwenbo}@otcaix.iscas.ac.cn

***Abstract*—Serverless computing is becoming one of the mainstream trends in cloud computing due to its advantages of simplified programming and cost saving. However, existing serverless platforms still adopt Docker container as its execution engine, which has the cold start problem and causes high common-case invocation latency. In this work, we analyze the lifecycle of common-case serverless invocation on existing serverless platforms and find that current container startup and pulling remote images are the two main reasons causing cold start so slow. Based on the study, we implement Nuka, a generic engine with millisecond initialization for serverless computing. Nuka is fully compatible with Docker interface and can smoothly replace Docker as the execution engine of existing serverless platforms. Through the isolation pool that reuses Linux's isolation configurations, Nuka avoids the high cost of container startup's scalability bottleneck, which reduces container's startup time with high concurrency scale. Nuka also avoids pulling remote images through dynamically resolving and importing required software packages from local package caching. A self-adaptive container reuse strategy dynamically controls container's pause time and replica numbers, which effectively reduces the frequency of cold start. Compared with Docker, Nuka can get a millisecond initialization with high concurrency and significantly reduces average time cost of cold startup by 6× on existing serverless platforms.**

Keywords—Serverless Computing, Cloud Computing, Container, Performance Optimization, Performance Analysis

I. INTRODUCTION

Serverless computing is a programming paradigm and execution model of cloud computing. In serverless computing, cloud providers are fully responsible for managing VMs, containers and software environments, and cloud users only focus on writing their codes. Serverless computing bills based on the actual resource usage consumed by invocation of applications. Therefore, it has the advantages of simplifying programming and cost saving compared to traditional cloud computing. Due to these advantages, serverless computing has emerged rapidly in the past several years. Now there have been a number of commercial serverless platforms (e.g, AWS Lambda[21], IBM Cloud Functions[9], Microsoft Azure Functions[8], and Google Cloud Functions[10]) and open source serverless platforms (e.g, OpenFaaS[17], Apache OpenWhisk[19], OpenLambda[4], and Knative[20]). These existing serverless platforms have gradually been able to support a variety of different workloads.

The minimal computation unit of serverless computing is called a function, serverless platform allocates resources and prepares software environment for the initialization of the associated function's execution environment when a request is received. After the function finishes executing, serverless platform will destroy the software environment and release the allocated resources. Compared to the traditional approach that handles requests with long-occupied resources and long-running applications, we call this approach with real-time initialization "cold start" handling.

Since the first serverless platform AWS Lambda came out just in 2015, the architecture and software ecosystem of existing serverless platforms generally continue to adopt existing approaches in traditional cloud computing. But due to the huge changes on "cold start" handling, serverless platforms are suffering non-negligible time cost on cold start while based on current architecture and software. Docker[23] is the standard fact of current Linux container, it provides applications with isolated environments. Existing serverless platforms generally adopt Docker as the execution engine of functions[18][19][20]. To execute the associated function, serverless platform need to start container and initial software environment previously. These operations usually cost hundreds of milliseconds in existing serverless platforms, which makes request's response time far longer than the function's real execution time. The problem of cold start has become one of the biggest challenges in the development and evolution of serverless computing.

From the perspective of cloud users, if cold start becomes faster and the frequency of its occurrence becomes lower, request's response time can lower latency so that cloud users will have a better user experience on serverless platforms. From the perspective of cloud providers, reducing cost on cold start enables cloud providers to recycle system resources faster[13], thereby effectively increasing resource utilization in production datacenters[7].

To reduce cost on cold start, we first analyze the life cycle of common-case serverless invocation, and finds the main reasons causing cold start so slow and some other motivations driving our design, and then introduce the design and implementation of Nuka based on these studies. Nuka reduces time cost on cold start through three techniques: (1) The isolation pool reuses namespace[28] and cgroups[29] through lightweight pause containers in the pool. It carefully avoids the bottleneck of creating and destroying Docker container with high concurrency scale. (2) The local package caching avoids pulling images through dynamically resolving and importing required software packages from local software package caching. (3) A self-adaptive container reuse strategy dynamically controls container's pause time and replica numbers, which effectively reduces the frequency of cold start. Compared with Docker, Compared with Docker, Nuka can get

a millisecond initialization with high concurrency and significantly reduces average time cost of cold startup by 6× on existing serverless platforms.

The rest of this paper is structured as follows. We first introduce the background and the motivations driving our design in Section II, and then introduce the design and implementation of Nuka in Section III and Section IV respectively. We finally evaluate the performance of Nuka in Section V, discuss related work in Section VI, and conclude in Section VII.

II. Background And Motivation

In this section, we study the reasons that causes slow initialization (cold start) in existing serverless platforms with Docker container and other motivations driving out design. Our studies are based on open source serverless platforms, but we think they can reflect commercial platforms with a variety of common characteristics.

A. Frequent Cold Start with Docker

We survey the execution model of four most popular and representative open source serverless platforms. Table I shows following conclusions: (1) **Docker is the execution engine of existing open source serverless platforms**. Docker has the advantages of reliability and comprehensive function, and has been validated in production-grade container-based platforms (eg. Kubernetes[36], Mesos[37]). (2) Serverless platforms execute a function with an individual container for better isolation (**one to one mapping**). (3) A container starts and initializes only when the associated request is received, and it destroys when finishing executing a function (**cold start handling**). (4) A container will not execute multiple functions at the same time (**without concurrent execution**).

TABLE I. Execution Model of Four Serverless Platforms

Name	Execution Engine	Container Function Mapping	Cold Start	Concurrent Execution
OpenWhisk	Docker	one to one	yes	no
Knative	Docker	one to one	yes	no
OpenFaaS	Docker	one to one	yes	no
OpenLambda	Docker / SOCK	one to one	yes	no

These above conclusions give us following implications: (1) Container startup frequency will be increased due to the features of new execution model, and decomposing of applications[2] will further amplify this problem. (2) Before starting container, container images need to be pulled from remote image registry[25] if it does not exist locally. If serverless scenario makes images hard to be cached, pulling remote images may be frequent and cost.

B. What Causes Cold Start So Slow?

Based on the implications in Section II.A, we benchmark the Nodeinfo[14] application in OpenFaaS and traditional web server respectively. We conducted the experiments on an Intel(R) Xeon(R) CPU E5-2682 v4 (2.50GHz), 32 GB RAM, SSD machine with the 4.13.0-37 Linux kernel. To better validate the influence of cold start, we close OpenFaaS's container remaining option, and the request concurrency is ensured within the maximum availability of the machine resources. As shown in Table II, compared with web server, invocation latency on OpenFaaS increases proportionally with request concurrency scale. Production-grade online services are usually latency-sensitive. For example, most Gmail remote procedure calls complete under 100 ms[11]. Such high invocation latency makes the quality of service unacceptable, which motivates our study on what causes cold start so slow with high request concurrency.

TABLE II. Latency and Throughput of OpenFaaS and Web Server with Different Request Concurrency

Request Concurrency	Latency (ms)		Throughput (reqs/s)	
	OpenFaaS	*Web server*	*OpenFaaS*	*Web server*
32	193.8	12.7	161.3	2078.1
64	315.0	14.8	182.7	2745.4
128	498.6	18.1	143.5	3049.2
256	1153.5	27.1	99.8	2804.8

Bottleneck of concurrent container startup. The first implication in Section II.A shows startup and destruction of containers will be very frequent in serverless platforms. For this scenario, we stress test concurrent startup and destruction of empty Docker containers respectively and find concurrent startup is a main bottleneck. Experiment results show that the maximum throughput of creating empty Docker containers is about 200~230 containers per second with the machine. With the concurrency of 256, average startup time reaches up to 1s and latency increases proportionally with the concurrency scale. While destruction of containers is fast relatively, which have a maximum throughput of 2500~3000 ops/s (operations per second). Staring a Docker container mainly includes loading image file, setting namespace and cgroups. To further study the reason causes the bottleneck of concurrent startup, we benchmark the six subsystems of Linux namespace (IPC, network, mount, PID, user, UTS), cgroups subsystem and image loading operation. Experiment result shows concurrent creation of network namespace is the key bottleneck, its maximum throughput is only 200~250 ops/s and latency increases proportionally with concurrency scale. Besides, Cgroups involves the control of multiple hardware resources, which causes its maximum throughput is only 350~400 ops/s and makes cgroups become the second bottleneck. Mount namespace is another bottleneck with a maximum throughput of 800~1000 ops/s. The throughput of other namespaces and loading image is about 2000~3000 ops/s, whose latency does not increase significantly with concurrency scale.

Bottleneck of pulling images. We collect and analyze a set of sample function's images provided by OpenFaaS. We find the size of 80% images is 1~10MB, but it still takes hundreds of milliseconds to pull these images in a 10 Gigabit/s production-grade bandwidth. Our further study measures the time cost on each procedures of pulling a remote image. Experiment result shows downloading the compressed files of required image layers actually takes very little time in modern datacenters (eg. an image with 100MB size at most takes 10 ms to download its compressed file), while the procedures of resolving required layers, compressing, decompressing and validating images will cost at least hundreds of milliseconds. Our study shows the layer number of images is generally 3~5, and the latency of these procedures increases with the image layer numbers. Fortunately, if the required image has already existed on the machine, pulling operation will not be performed. So the actual effect on cold start of pulling images depends on a machine's ability of caching images.

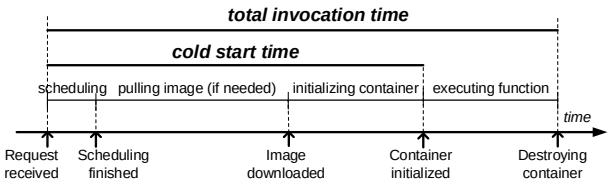

Fig. 1. Lifecycle of common-case invocation on existing serverless platforms

In conclusion, as shown in Fig. 1, **pulling image and starting container with high concurrency are the two main bottlenecks of cold start in existing serverless platforms**.

C. Caching Images May Be Hard

In order to know the actual effect of pulling images mentioned in Section II.B, we explore the potential total image size that might be cached on a machine in serverless scenario. Serverless computing requires cloud users decompose the original application to multiple single-logic functions for more fine-grained scheduling and billing. For example, we can decompose a microservice by its RESTful API and make one API correspond to one function. With this specification, we rebuild Docker images for each function on five Python applications and compare the total image size after rebuilding with the original image size. As shown in Table 3, we find magnification of image size increases proportionally with the increasing of decomposing number, which may magnify image size by tens to hundreds of times. Now production-grade machines may cache tens to hundreds GB images of microservices to reduce pulling images[32]. With the popularity of hardware resources disaggregation[33], modern datacenter machines generally provide at most 1TB SSD storage size, which may make it hard to cache most of the images locally in serverless scenario.

Through further inspecting the procedure of building Docker container, we find the application dependencies are built into the top layer of the image, which makes some same dependency repeatedly loaded by different images. But if we build all dependencies into the read-only common underlying layer, those uncommon-used dependencies will increase the size of all images. This gives us the implication that a new alternative approach should be used to avoid the original image building-pushing-pulling workflow.

TABLE III. IMAGE SIZE COMPARISON OF ORIGINAL APPLICATIONS AND SERVERLESS APPLICATIONS

Name	Decom-posing number	Python package number	Original size (MB)	Rebuild total size (GB)	Magni-fication
piggy	34	11	66.5	1.69	25.4
delta	51	12	70.3	2.89	41.1
nebul	72	14	82.1	4.78	58.2
sahoo	93	14	77.6	5.35	69.0
conroy	117	16	89.5	6.74	75.3

D. Pausing/Unpausing Container Is Fast

In order to explore further potentiality on reducing cold start cost with Docker container, we benchmark the six operations in the lifecycle of Docker container. Experiment result shows pausing and unpausing a container can be completed within hundreds of microseconds, which is 100× or more faster than creating a new container or staring a stopped container. This is because paused container only switches out its CPU context and still occupy the memory when memory resource is sufficient. Thus paused container can void the slow container initialization (mainly including container startup and function runtime initialization). But if a paused container keeps idle for a long period of time, memory resource will be wasted and frequent swap between memory and disk will occur when memory is insufficient. This will be the technique support of reusing container for multiple requests to reduce cold start frequency we discussed in Section III.C.

III. NUKA SYSTEM DESIGN

In this section, we describe our three key techniques on the design of Nuka. We first present the design of isolation pool that reduces container startup time and increases throughput of container startup with high concurrency. We then present the design of local package caching to void pulling image. We finally introduce the design of container reuse strategy to reduce cold start frequency.

A. Isolation Pool

The key idea of isolation pool is to void the high cost of current network namespace, cgroups and mount namespace creation discussed in Section II.B through pooling and reusing them. As shown in Fig. 2, isolation pool mainly contains a local message queue to store messages of container startup and destruction, an agent to manage the lightweight pause containers in the pool as well as add/destroy the process group recorded in the messages.

Local message queue. In order to smoothly replace Docker as the execution engine, we consider Nuka use the same interfaces and specifications with Docker for container lifecycle management. Containerd[31] is the key component of Docker that is responsible for lifecycle management of container and image, and more and more serverless platforms or other container management platforms have taken bare containerd as the container runtime interface. Thus we use a stub-based approach to take over the container startup and destruction requests from containerd (like docker run and docker stop command), these requests are originally transmitted to the underlying libcontainer[30] to start and destroy containers with the original startup and destruction approach used by Docker. The requests are resolved and reformatted to a message based on our specification and then stored in the local message queue. The local message queue is a FIFO queue and has the advantages of high reliability and scalability, which can ensure that request is received and executed by the pool agent and the agent work normally with high request concurrency respectively.

Lightweight pause container. The original approach of creating containers with libcontainer uses the *unshare*() system call, which will create a new namespace for the process. We have validated this approach has bottleneck with high concurrency. Now we add the process group to an existing namespace with *setns*() system call, and then propose two questions: (1) How to maintain these existing namespace configurations and how does it cost? (2) Are there residual configurations or files in the existing namespaces and will they disturb the process group to be added? We solve this problem with a lightweight pause container and validate our questions. Inside a pause container, there is only a lightweight daemon checker that checks and recovers configurations and files when new process group is added to the namespace of a pause container. For example, for PID namespace, the checker

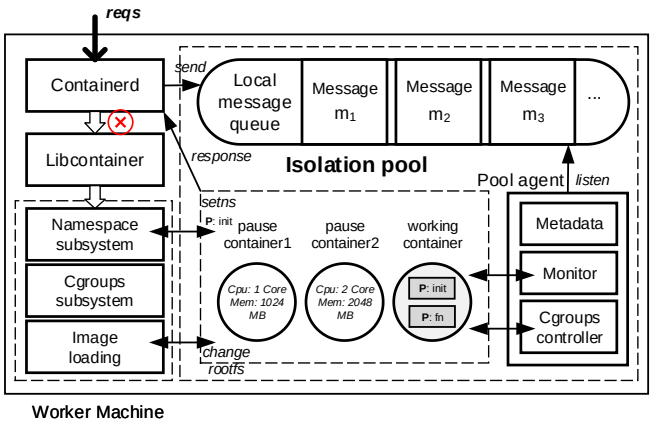

Fig. 2. The design of isolation pool

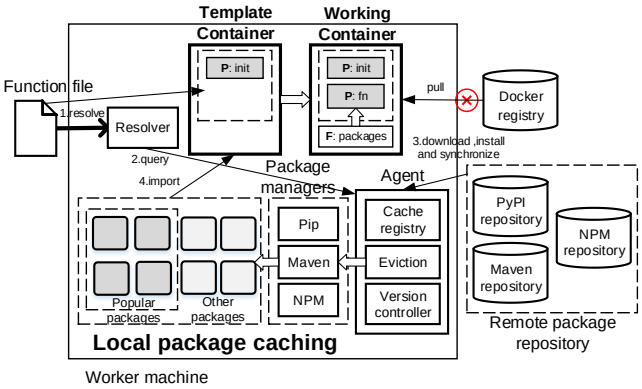

Fig. 3. The design of local package caching

will check whether there is only the check process itself in the container, and other process will be killed if existed. For mount namespace, all additional mount points will be unmounted if existed (except the directory of local package caching in Section III.B). For network namespace, if the checkers detects that the network configuration has been modified (eg. adding NIC and bridge), checker replaces them with original configuration files and make it effective. These operations are measured very fast and the average time cost is just several milliseconds. Thus the checker ensures the clean namespace environment, and process group will not be disturbed. These lightweight pause containers are managed by the pool agent, pool agent will activate the checker add the process group associated with the function to an idle pause container, and then function will be executed with isolation.

Dynamic cgroups configuration. Production-grade serverless platforms require adjustable resource limitations. For example, cloud users can manually specify memory quota of a function in AWS Lambda. Thus, we design a cgoups controller in the pool agent to modify cgroups configurations of these pause containers. When pool agent receives a startup message, the cgroups controller modifies the selected pause container's cgroups configuration files based on the new resource limitation in the message, and then move the process group to the cgroups. We benchmark this new approach and the original approach of creating/deleting cgroups, experiment result shows that this new approach is 2~5 times faster than the original approach with different concurrency scale.

B. Local Package Caching

The main purpose of local package caching is to void pulling image from remote Docker registry as well as reduces downloading and installing of software packages. The core idea of our local package caching mainly consists of multiple languages support, caching strategy for popular and unpopular packages, cache registry three parts. It abandons the original layered image storage mechanism of Docker to build required dependencies to the top layer of the image and provides a shared storage area on every worker machine, then a template container will mount the storage area and import its required packages through the resolver and the agent.

Multiple languages support. Production-grade serverless platforms are required to run applications written by multiple mainstream programming languages. For example, AWS Lambda supports Python, JavaScript, Java, Golang and etc. Programming languages can be divided into two types through the way of building and running. The first type is interpreted language (eg. Python, JavaScript, Java). They run their applications by interpreter (or language runtime) and will not build dependencies into a binary file, language runtime will import these dependencies through the local package index (eg. Java's .m2 directory). The other type is static language (eg. Golang). They build their applications and dependencies into a binary file that can be directly executed by operation system, which makes their software environment preparation relatively simple. We just need to store and get the binaries.

We make a high-level abstraction on package management for the complex interpreted languages, and provide an unified management for them. Our core idea is to reuse the ability of existing package managers (eg. Maven[35], Pip[24], NPM[17]) with maximization. We find there are downloading, installing, uninstalling and importing four basic operations. We abstract these operations of different languages into five unified primitives (download, install, uninstall, import and query), which blocks the differences between package management of multiple different languages. As shown in Fig. 3, the resolver will resolve the import statements in the submitted function code and then generate corresponding primitives, then the agent transfers the primitives to the package manager, the selected package manager performs the basic package management operation finally.

Caching strategy. In a Python package study by Oakes et al.[1], 36% of import statements are to just 0.02% of packages, which implicates most functions actually depend on a few part of packages. Thus we define the top 100 popular packages in each supported language as popular packages (eg. the top 100 Python packages occupy up to 70% import statements), these popular packages are cached on every worker machines of a serverless platform. We have tested the total storage size and installing conflict of these popular packages, and find it just takes several GBs local storage and no installing conflict occurs with the newest stable language interpreter or runtimes. Based on the result, we distribute these popular packages on every worker machine and then install them permanently. For the other packages, worker machines try maximum effort to cache them while two main problems might occur. The first is when machine does not have enough storage to cache a new package, we design a cache registry in the agent to record the cached packages and its recent importing operations, then agent will evict some packages based on LFU or LRU. The second problem is installing conflicts might occur when installing a new package based on a lot of installed packages, the agent will uninstall conflict packages from top to bottom. The cache registry will remove the item of the packages after they are uninstalled.

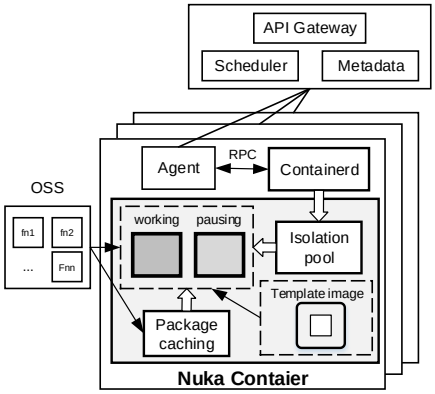

Fig. 4. The high-level architecture of Nuka in a serverless platform

C. Container Reuse

To reduce the frequency of cold start, a common practice is to reuse launched containers by remaining them warm for a period of time. Similar approach has already been used in AWS Lambda. Based on the finding in Section II.D, we introduce a more efficient container reuse strategy that both reduces latency and increases memory utilization. We divide the type of request load into stable loads and flash crowds these two categories and take different handling strategy for the two kind of loads for a more efficient container reuse.

Handling stable load. Stable load has the characteristic of approximate average load size in each short period of times. Based on this characteristic, we remain a launched container pause for a period of time without destroying it after it finishes executing a function. When the next request is received, we just unpause an existing associated pausing container to handle the request, thus cold start will not be performed. The setting of container pause time bases on function execution time, load size of the request and number of containers to handle the request. Our core idea is to make the maximum handling ability of a container larger than load size received by it. As shown in (1), t_f and t_p are the average time of function execution and real pause time respectively, Q represents load size of the kind of request (requests per second) and N represents the number of containers to handle the request, so the right part of (1) means the average number of requests received by one container in a second, the left part of (1) means the number of requests one container can handle. To minimize the memory occupation by idle containers as well as the slight uncertainty of load size, we just need to set maximum container pause time slightly larger than the threshold of t_p computed by (1).

$$1 / (t_f + t_p) > Q / N \tag{1}$$

Handling flash crowd. Flash crowd is a sudden and large surge in traffic of some workloads, which is hard to be handled by the same approach of handling stable load. Because the load size may suddenly become very large, then the number of pausing containers is not enough to handle the load size, which makes requests waiting and increases invocation latency. When the load size becomes very small, most of the associated paused will be idle and memory resource will be wasted. To better handle the flash crowd, we use an automatic container scaling strategy that bases on the changing load size. When the load size increases and requests occur waiting, we quickly expand the container replicas until no requests waiting, and when load size decreases, we contract container replicas to the minimum number that ensures no requests waiting. For horizontal container expansion, we have a further optimization, if the container to be expanded already started on the machine, we will use *fork*() and *clone*() to create the new container, which reduces the cost of function's runtime initialization.

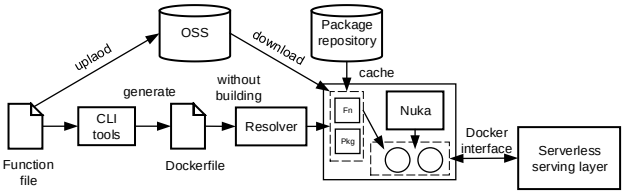

Fig. 5. The new workflow of deploying a function with Nuka

IV. Implementation

In this section, we describe the implementation of Nuka container system. We first introduce detailed implementation techniques, its interaction with existing serverless platforms, and then describe how a function is deployed in the new workflow with Nuka.

A. Overview

We implement the complete Nuka container system with all components described in Section III based on Docker. Nuka relies on Docker's containerd component to manage the lifecycle of containers and images. Besides, Nuka is fully compatible with the specifications and interfaces of containerd and can port containers previously running on Docker or bare containerd to it smoothly. As shown in Fig. 4, Nuka is deployed and running on every worker machine in a serverless platform, and we require containerd is previously running on the machine. The requests from agent of serverless platform to containerd will be redirected to Nuka's isolation pool to manage the containers on the machine. A 237 KB template image that only contains the checker application is provided to create lightweight pause containers in the isolation pool. Function files submitted by cloud users are stored in the OSS (object storage service). Cadvisor[34] and our container reuse strategy is integrated in the isolation pool to monitor resources utilization and control container's pause time replica number respectively. In addition, we implement isolation pool (about 2300 lines code) and local package caching (about 900 lines code) mainly by Python and shell scripts.

B. Function Deployment

We now describe our new workflow to deploy the original functions and interactions with serverless serving layer to handle requests. In the original workflow, the client tool of serverless platform resolves the function file and its associated parameters and then generates a Dockerfile of the function, then builds a image through the Dockerfile and push it to the remote Docker registry. When a request is received, the worker machine will pull the image from Docker registry for the first time and then initial the container (through docker run) based on the image to handle the request. In our new workflow, as shown in Fig. 5, a resolver will resolve the RUN commands in the Dockerfile, then downloads and installs dependent packages that is not cached on the machine. The submitted function files will be uploaded to a OSS, and then worker machine will download function file from OSS and Nuka adds the process group associated with the function to a pause

container in the isolation pool and then imports required packages to handle the request.

V. Evaluation

In this section, we evaluate the effectiveness of Nuka. In our evaluation, we mainly study the following questions:

1. How effective does the three designs of Nuka work compared to previous unoptimized scenario?

2. How well does Nuka meet the latency, scalability, throughput and resource utilization compared with Docker and other existing related works?

3. How well does open source serverless platform working with Nuka work compared to production-grade commercial serverless platform with real workload case?

All experiments were run on one Intel(R) Xeon(R) CPU E5-2682 v4(2.50GHz), 32 GB RAM, 256 GB SSD machine with the 4.13.0-37 Linux kernel.

A. Evaluation of Isolation Pool

In order to validate the scalability of isolation pool, we conduct the experiment of concurrent container startup and destruction with isolation pool of different pause container number. As shown in Fig. 6, with a pool of 256 pause containers, the container startup latency does not increase proportionally as we vary the concurrency under 256. The result of a pool with 128 pause containers further illustrates its effectiveness. When concurrency reaches up to 128, latency begins to increase proportionally with the pool of 128 pause containers. This is because only concurrency is under the number of pause containers in the pool, all startups can reuse the existing isolation configurations (we show in Section II.B that some operations such as creating network namespace rely on a global lock and have bottleneck with concurrency), and excess startups will be performed by the original approach. As shown in Fig. 7, the throughput also correspondingly increases with sufficient pause containers. These have shown isolation pool can effectively reduce cost of creating containers with high concurrency through avoiding the previous scalability bottleneck of creating containers.

B. Evaluation of Local Package Caching

In order to validate the effectiveness of local package caching, we measure the average latency of pulling image and dynamical importing packages respectively with OpenFaaS's sample functions. We build the images through the Dockerfiles provided by OpenFaaS and push them to a remote Docker registry with a 10 Gigabit/s production-grade network bandwidth. These images are all built with the size between 10~100 MB and dependencies occupy the main size of images. Through resolving the import statements of function files by the cache agent, the required packages of these functions are all popular packages and have been cached on the machine with our caching strategy. As shown in Fig. 8, the latency is reduced from hundreds of milliseconds to about 20 milliseconds in average because it only needs to resolve dependencies by the cache agent and then starts the interpreter or runtime. Considering some functions may rely on some unpopular packages, we measure the downloading and installing time of 100 unpopular Python packages and find the average time is about tens of milliseconds in our environment, and a function usually requires less than 3 unpopular packages, which is not enough to cause much latency compared to pulling container.

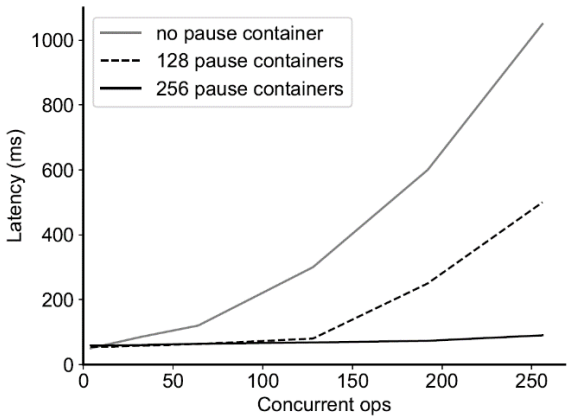

Fig. 6. The average latency of concurrent container startup/destruction with isolation pool of different pause container number

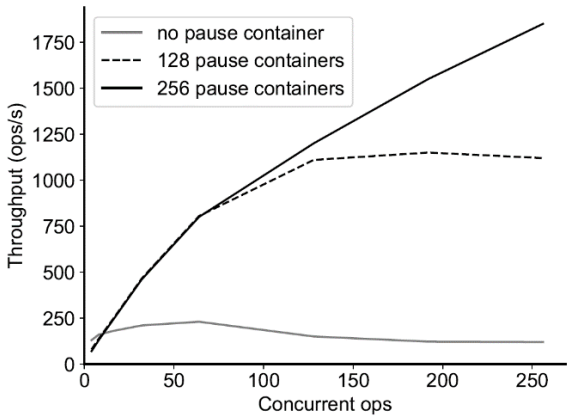

Fig. 7. The throughput of concurrent container startup/destruction with isolation pool of different pause container number

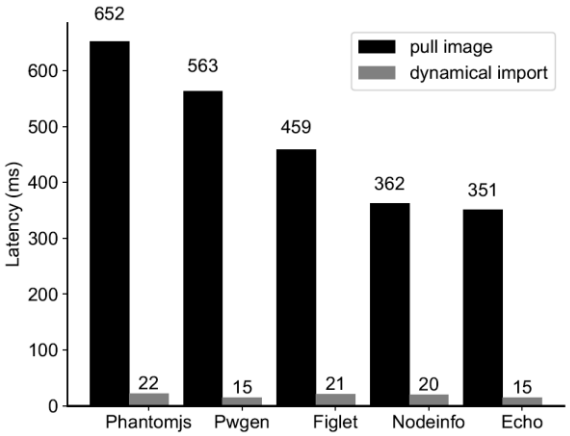

Fig. 8. The latency of pulling image and dynamical importing packages with OpenFaaS's sample functions

C. Evaluation of Container Reuse

In order to validate the effectiveness of our container reuse strategy, we stress test the Nodeinfo application with stable load and flash crowds respectively. As shown in Fig. 9, after using the container reuse strategy, the average latency has been reduced closely to the real function execution time compared with none optimization and only isolation pool for stable load. We observe there is almost no occurrence of new container initialization after container replica number and container pause time become stable, which illustrates the container reuse strategy is effective for stable load. Compared

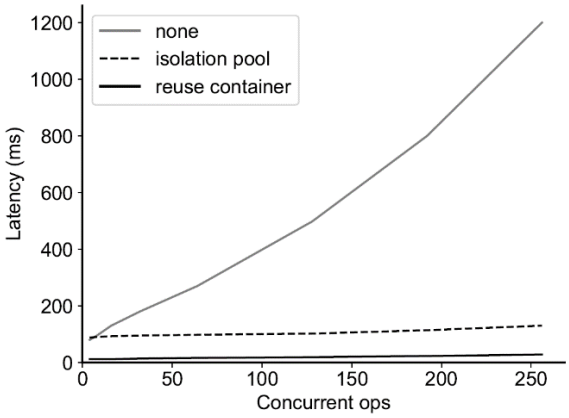

Fig. 9. The average latency of Nodeinfo application with none optimization, isolation pool and container reuse

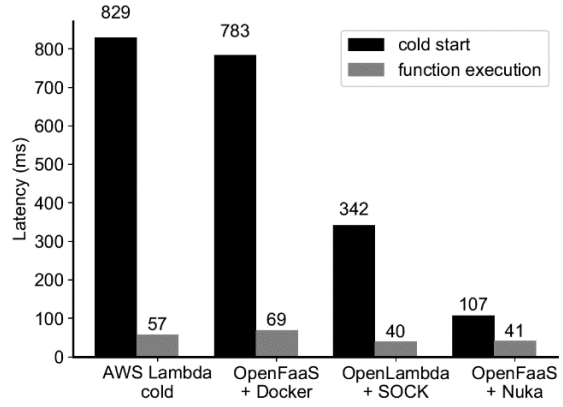

Fig. 10. The latency of handling an Image-Resizing application for the first time on the worker machine with different platforms

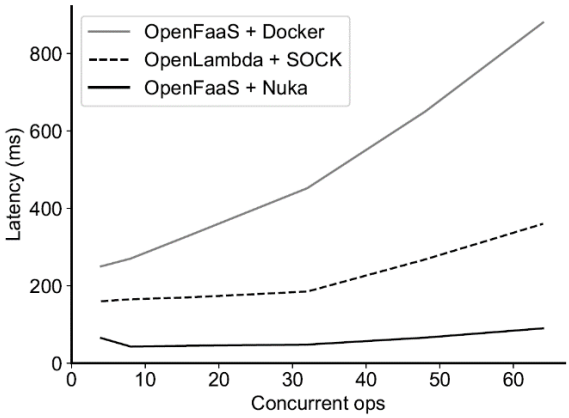

Fig. 11. The average latency of handling Image-Resizing application with different concurrency with Nuka, Dokcer and SOCK

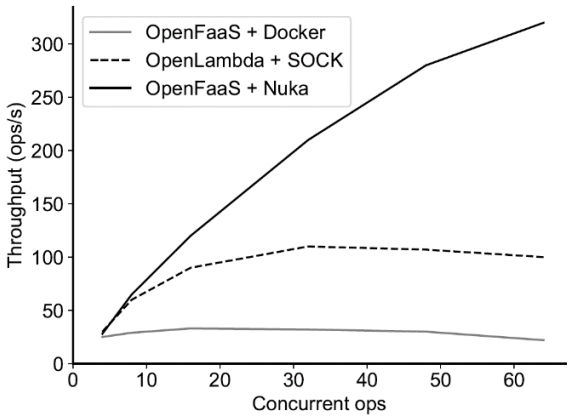

Fig. 12. The throughput of handling Image-Resizing application with different concurrency with Nuka, Dokcer and SOCK

with none optimization, the CPU utilization increases by 20 ~ 40%. In addition, the throughput increases by 5× and 50× compared with none optimization and only isolation pool respectively. For flash crowd, it can reduce the container initialization latency within 50ms under the concurrency of 256, which ensures the quality of service in this scenario.

D. Evaluation of Nuka

In order to better evaluate the effectiveness of Nuka in the real serverless scenario, we use the sample Image-Resizing application from AWS Lambda and port it to serverless platforms that we will benchmark. In the Image-Resizing application, the function reads an image from AWS S3, uses several packages to resize it, and writes the output back to AWS S3[15]. We instrument handler code to separate compute and S3 latencies from platform latency. For this experiment, we compare AWS Lambda with cold start, OpenFaaS with Docker, OpenLambda with SOCK, and OpenFaaS with Docker. As shown in Fig. 10, for the first time of handling Image-Resizing application, Nuka reduces cold start latency by 6× compared with Docker in the same OpenFaaS platform and hardware resources. In addition, OpenFaaS with Nuka also outperforms the state-of-the-art commercial serverless platform AWS Lambda and latest related research work SOCK. The experiment result of latency and throughput with different concurrency are shown in Fig. 11 and Fig. 12 respectively. Compared with Docker, Nuka gets a lower latency by 5~10× and a higher throughput by 3~20× with the concurrency in the experiment. Compared with SOCK, Nuka gets a lower latency by 3~5× and a maximum throughput by 3×. For most of the sample functions provided by OpenFaaS, Nuka can reduce cold start latency under 100ms with affordable concurrency.

VI. Related Works

Serverless computing now has already been used in various scenario such as IoT and edge computing[22], big data processing[6], video analysis[26], machine learning[16], and enhancement of system security[27]. It is expected that serverless is going to be one of the mainstream trends of cloud computing in the future[2]. There are mainly container-based and architecture-based two solutions to reduce high invocation latency and increase resource utilization in existing related works.

Container-based optimization. Oakes et al.[1] proposed SOCK to reduce time cost on cold start, SOCK uses more streamlined isolation mechanism to replace the original isolation of Docker and uses Zygotes provisioning to cache Python interpreter and necessary libraries in the memory to avoid scalability bottlenecks and accelerate application initialization. Thalheim et al.[11] proposed the lightweight container Cntr, it splits unused content of production environment into another container image, and the split images collaborate through remote communication, which reduces the time of pulling remote image in production environment and does not affect the use of containers in the development environment.

Architecture-based optimization. Akkus et al.[3] proposed SAND serverless platform that uses application-level sandboxing and hierarchical message bus two key techniques to execute multiple associated functions in the same container, which reduces code start latency get a better resource utilization in multiple associated functions invocation scenario. McGrath et al.[12] proposed an accelerating approach with workers maintaining warm queues and cold queues, where containers can be reused and new containers can be created respectively. Boucher et al.[5] proposed a novel serverless platform design that makes cold start latency in microseconds based on language-based isolation and microsecond-scale preemption.

VII. Conclusion

Serverless computing provides simplified programming and cost saving for cloud users and is becoming one of the mainstream trends in cloud computing. However, existing serverless platforms still adopt the traditional Docker container as execution engine, which has the cold start problem and causes high common-case latency on serverless platforms. In this work, we analyze the life cycle of common-case serverless invocation and find the main reasons causing cold start so slow. Based on our study, we design and implement Nuka, a generic engine with millisecond initialization for serverless computing to replace Docker. Nuka avoids the high cost of container startup scalability bottleneck through the isolation pool. Nuka also reduces the time cost on pulling image by dynamically resolving and importing required packages from local package caching. Nuka further provides an self-adaptive container reuse strategy that dynamically controls container's pause time and replica numbers, which effectively reduces the frequency of cold start. Our hope is that this work can minimize cold start cost in existing serverless platforms, which will emerge the popularity of various applications deployed on serverless computing.

Acknowledgment

This work was supported by funding from national key research and development plan 2018YFB103602.

References

[1] Oakes E, Yang L, Zhou D, et al. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers[C]//2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018: 57-70.

[2] Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing[J]. arXiv preprint arXiv:1902.03383, 2019.

[3] Akkus I E, Chen R, Rimac I, et al. SAND: Towards High-Performance Serverless Computing[C]//2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018: 923-935.

[4] Hendrickson S, Sturdevant S, Harter T, et al. Serverless Computation with OpenLambda[C]//8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). 2016.

[5] Boucher S, Kalia A, Andersen D G, et al. Putting the "Micro" Back in Microservice[C]//2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018: 645-650.

[6] Jonas E, Pu Q, Venkataraman S, et al. Occupy the Cloud: Distributed Computing for the 99%[C]//Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017: 445-451.

[7] Liu Q, Yu Z. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: a View from Alibaba Trace[C]//Proceedings of the ACM Symposium on Cloud Computing. ACM, 2018: 347-360.

[8] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/ functions/. 2019.

[9] IBM Cloud Functions. https://www.ibm.com/cloud/ functions/. 2019.

[10] Google Cloud Function. https://cloud.google.com/functions/. 2019.

[11] Thalheim J, Bhatotia P, Fonseca P, et al. Cntr: Lightweight OS Containers[C]//2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018: 199-212.

[12] McGrath G, Brenner P R. Serverless Computing: Design, Implementation, and Performance[C]//2017 IEEE 37th International Conference on Distributed Computing Systems Workshops. IEEE, 2017: 405-410.

[13] Wang L, Li M, Zhang Y, et al. Peeking Behind the Curtains of Serverless Platforms[C]//2018 Annual Technical Conference. 2018: 133-146.

[14] Nodeinfo. https://github.com/openfaas/faas/tree/master/sample-functions/NodeInfo. 2019.

[15] Image-Resizing. https://aws.amazon.com/cn/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/. 2019.

[16] Zhang C, Yu M, Wang W, et al. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving[C]//2019 USENIX Annual Technical Conference (USENIX ATC 19). 2019.

[17] NPM. https://www.npmjs.com/. 2019.

[18] OpenFaaS. https://www.openfaas.com/. 2019.

[19] Apache OpenWhisk. https://openwhisk.apache.org/. 2019.

[20] Knative. https://cloud.google.com/knative/. 2019.

[21] AWS Lambda. https://aws.amazon.com/lambda/. 2019.

[22] de Lara E, Gomes C S, Langridge S, et al. Hierarchical Serverless Computing for the Mobile Edge[C]//2016 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 2016: 109-110.

[23] Docker. https://www.docker.com/. 2019.

[24] Pip. https://pypi.org/project/pip/. 2019.

[25] Docker registry. https://github.com/docker/distribution. 2019.

[26] Fouladi S, Wahby R S, Shacklett B, et al. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads[C]//14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). 2017: 363-376.

[27] Bila N, Dettori P, Kanso A, et al. Leveraging the Serverless Architecture for Securing Linux Containers[C]//2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017: 401-404.

[28] Linux namespace. http://man7.org/linux/man-pages/man7/namespaces.7.html. 2019.

[29] Linux cgroups. http://man7.org/linux/man-pages/man7/cgroups.7.html. 2019.

[30] Libcontainer. https://github.com/opencontainers/runc. 2019.

[31] Containerd. https://github.com/containerd/containerd. 2019.

[32] Anwar A, Mohamed M, Tarasov V, et al. Improving Docker Registry Design Based on Production Workload Analysis[C]//16th USENIX Conference on File and Storage Technologies (FAST 18). 2018: 265-278.

[33] Shan Y, Huang Y, Chen Y, et al. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 69-87.

[34] Cadvisor. https://github.com/google/cadvisor. 2019.

[35] Maven. https://maven.apache.org/. 2019.

[36] Kubernetes. https://kubernetes.io/. 2019.

[37] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center[C]//NSDI. 2011, 11(2011): 22-22.