Scorpius: Proactive Code Preparation to Accelerate Function Startup

Heng Yu[1], Junxian Shen[1], Han Zhang[1,2], Jilong Wang[1,2,3], Congcong Miao[4], Mingwei Xu[1,2,3]
[1]Institute for Network Sciences and Cyberspace, Tsinghua University
[2]Beijing National Research Center for Information Science and Technology (BNRist)
[3]Peng Cheng Laboratory (PCL), [4]Tencent
{yuheng20}@mails.tsinghua.edu.cn, {zhhan, xumw}@tsinghua.edu.cn,{wjl}@cernet.edu.cn

2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS) | 978-1-6654-6824-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/IWQoS54832.2022.9812868

***Abstract*—Massive enterprises deploy their applications on public clouds to relieve infrastructure management burden. However, applications are faced with highly fluctuating workloads, while clouds provision exclusive resources at coarse time granularity, resulting in severely low resource efficiency. Function-as-a-Service (FaaS) platform enables fine-grained resource multiplexing, which has the potential to improve efficiency. However, FaaS platforms could consume several seconds to start functions and the long startup latency can severely hurt the performance of applications. In this paper, we measure the FaaS platforms and find that most startup latency is occupied by code preparation. To reduce the code preparation latency with little resource overhead, we propose Scorpius, a FaaS platform that proactively prepares code based on the historical data of functions. It combines two optimization categories: (1) To reduce the code size, Scorpius proposes to proactively prepare partial libraries over servers and run functions on the server with most library sharing. (2) To advance the start time, Scorpius proposes to predict the function overload with a simple model and proactively scale code to more servers. We have implemented a prototype of Scorpius and conducted extensive experiments. Evaluation results demonstrate that compared with state-of-the-art methods, Scorpius can reduce the code preparation latency by 87.6% with only 9.3% storage overhead.**

I. Introduction

Nowadays, massive enterprises deploy their applications on public clouds to relieve infrastructure management burden [1]. Running applications on such Infrastructure-as-a-Service (IaaS) platforms requires exclusively monopolizing hardware resources (e.g., CPUs, memory) at a coarse time granularity (minutes or hours). Meanwhile, the request rate of real-world applications keeps drastically fluctuating all the time [2]. However, over-provision resources to meet peak workload demands could introduce severely low resource efficiency [3]. To enable statistically multiplexed resources, a thriving model named *FaaS*, also known as *serverless computing*, is emerging as an essential evolution in cloud computing [4]. In this model, users only need to upload their applications as a set of functions, and the FaaS platform runs these functions anywhere with enough resources when receiving requests [5]. Thus, users only pay for what they use at a fine time and resource granularity [6].

However, the benefits of the FaaS platform come with overhead. To handle requests, the platform could consume several seconds to start functions and this process might hurt the performance of applications [7]. The startup workflow mainly includes three phases: (1) Prepare code (e.g., download codes and install libraries); (2) Boot sandbox (e.g., start VMs or containers); (3) Launch application (e.g., initialize runtime and load codes). Up till now, many researches have explored the optimization of sandbox booting [8], [9] and application launching [10], [11]. These researches can effectively reduce startup latency, but they *ignore* the code preparation, while this phase can occupy more than 66% latency according to our measurement (See §II). Orthogonal to above researches, this paper focuses on code preparation and mainly makes the following **contributions**:

Firstly, we take the OpenFaaS platform [12] as an example and find that even a simple function can experience 3000ms startup latency. We carefully analyze the workflow of function startup and identify code preparation as the main overhead. We measure real-world applications, and find that both imported libraries and function invocation follow the *heavy-tailed distribution*, i.e., few public libraries occupy dominant sizes of most functions and most user requests invoke few hot functions (See §II). Recent researches for code preparation optimization can be divided into two categories: (1) reducing code size, which optimizes from the space dimension. However, the package-based methods (e.g., REG [13]) *reactively* prepare libraries and can still incur high startup latency. The repository-based methods (e.g., SOCK [7]) adopt the strategy of *trade-time-by-space* and can lead to vast storage overhead; (2) advancing start time, which optimizes from the time dimension. However, the histogram-based methods (e.g., Azure [14]) only focus on the function cold start and *ignore* the function scale out for most requests. The ML-based methods (e.g., Sinan [15]) rely on *complex prediction model* and could consume massive computing resources. Therefore, none of them are good solutions for code preparation in practice.

Secondly, we propose Scorpius, a FaaS platform that *proactively prepares code* based on the historical data of functions. It combines the two optimization categories to accelerate function startup (See §III). To reduce the code size without much storage overhead, Scorpius proposes to *proactively* prepare partial libraries over servers, and run functions on the server with most library sharing. To advance the start

978-1-6654-6824-4/22/$31.00 ©2022 IEEE

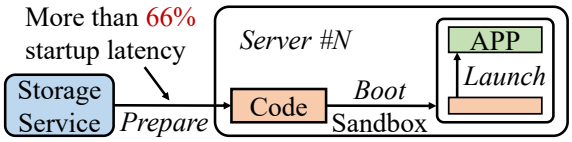


Fig. 1: Workflow of function startup

time for function scale out without much computing overhead, Scorpius proposes to predict the function overload with a simple model, and *proactively* scale code to more servers. However, applying this design to FaaS platforms can face two challenges: (1) how to proactively prepare and share public libraries over servers. Since the libraries occupy most sizes of codes, the proactive preparation should promote library sharing. (2) how to proactively scale code to more servers with a simple model. Since most functions are invoked very infrequently, the resource prediction model should be simple enough. To overcome these challenges, Scorpius proposes two designs, i.e., proactive library preparation algorithm and code scaling mechanism. The library preparation algorithm utilizes the historical data of user functions to guide the library preparation over all servers. Besides, Scorpius exploits the import frequency and clustering feature of libraries to improve the usage efficiency of reserved storage. The code scaling mechanism utilizes a simple threshold model to predict the overload of functions. Furthermore, Scorpius carefully sets the initial thresholds and dynamically adjusts them to match the actual situation of individual function.

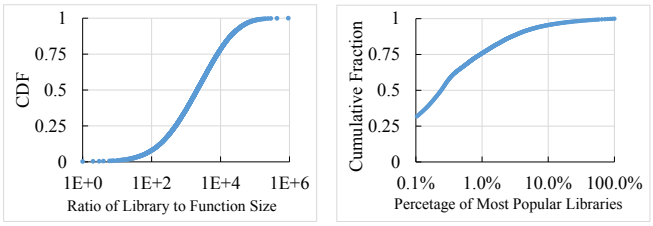
Thirdly, we implement a prototype of Scorpius and conduct extensive experiments through the real systems as well as trace-driven simulations across different methods and scenarios (See §IV). Evaluation results demonstrate that compared with state-of-the-art methods (e.g., REG [13]), Scorpius can reduce the code preparation latency by 87.6% with only 9.3% storage overhead. To the best of our knowledge, we are the first to thoroughly study the problem and propose a systematic solution for code preparation.

II. Background and Motivation

In this section, we first introduce the startup workflow of functions. Then, we describe our measurements of real-world functions and traces. Finally, we show the limitations of existing FaaS platforms on function startup.

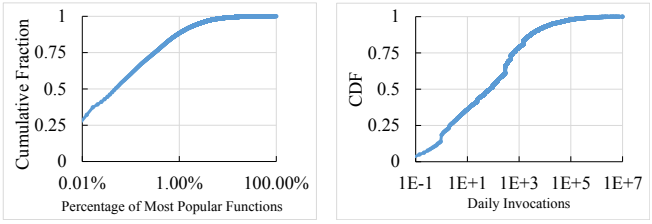
A. FaaS overhead

The functions startup overhead can hurt application QoS. Microservice architecture is a popular software engineering approach that enables large applications to evolve rapidly [16]. It is widely adopted by large companies such as Amazon to build their applications [17]. Microservices communicate with each other through remote procedure calls (RPC) and functions in FaaS platforms provide a native way to implement these RPC handlers [18]. However, FaaS platforms have cold start overhead, which can jeopardize the performance of microservices. According to our measurement, even a simple function (e.g., "Hello World") in OpenFaaS platform [12] can experience about 3000ms startup latency.



(a) Ratio of library to function size (b) Library cumulative fraction

Fig. 2: A real-world library import measurement



(a) Invocation cumulative fraction (b) Function daily invocations

Fig. 3: A real-world function invocation measurement

In reality, an application might contain tens to hundreds of microservices, and *the startup overhead in FaaS platforms can certainly hurt the QoS of application*.

The prepare code phase can occupy most startup latency. Fig. 1 shows the workflow of function startup, which mainly includes three phases: (1) Prepare Code phase. To run a function on a specified server, the FaaS platform needs to download user code and install public libraries (e.g., flask); (2) Boot Sandbox phase. To ensure isolation in the public cloud, each function needs to run in a separate sandbox (e.g., docker); (3) Launch Application phase. User function needs to run in a specialized runtime (e.g., python interpreter) and load required libraries. Although most FaaS platforms (e.g., OpenFaaS [12]) could compress the user code and required libraries into a sandbox image (e.g., docker image), the prepare code phase still occupies most startup latency, mainly due to the *dominant library overhead*. Our measurement on a simple python function (e.g., online service) also proves this, where the three phases consume about 2000ms, 600ms and 400ms respectively, and the prepare code phase covers more than 66% of the total startup latency.

B. Function measurements

Observation 1: few public libraries occupy dominant sizes for most functions. Developing high-quality functions with high-level programming language from scratch could be daunting even for professional developers, while importing public libraries can make this task much easier. We measure 64K open-source projects from GitHub [19] and Fig. 2 shows the statistics results. From Fig. 2(a), we can see that the size of public libraries imported by functions is much larger than the source code written by developers, e.g., their size ratio can even be more than 500 for over 75% functions. Fig. 2(b) implies that the minority of libraries are excessively hot, e.g., 13.8% of the most popular libraries are imported by more than 96.6% of all user codes. This also agrees with our practical

Category of Code Preparation		Existing Works	Effectiveness	Practicality	Scalability	Cold Start	Scale Out
Code size	Package-based	REG [13]	✗	✓	✓	✓	✓
	Repository-based	SOCK [7]	✓	✗	✓	✓	✓
Start time	Histogram-based	Azure [14]	✓	✓	✓	✓	✗
	ML-based	Sinan [15]	✓	✓	✗	✗	✓
Hybrid		**Scorpius**	✓	✓	✓	✓	✓

TABLE I: A comprehensive comparison between Scorpius and state-of-the-art code preparation works

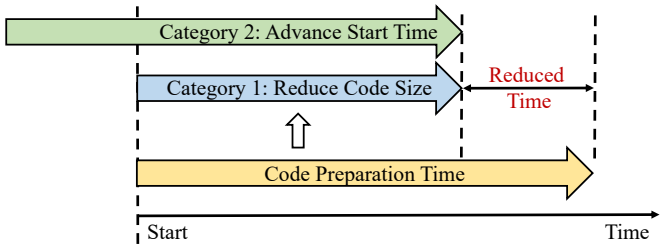


Fig. 4: Existing optimization categories on code preparation

experience, for example, the public library `flask` is more frequently used than the `pydeps`.

Observation 2: most user requests invoke few hot functions. In reality, the frequency of function invocation can vary significantly. Our measurement of real-world traces [14] shown in Fig. 3 illustrates the invocation feature of user functions. As shown in Fig. 3(a), we can see that few user functions are extremely hot and frequently invoked, e.g., 18.6% of the most popular functions are invoked by 99.6% of all invocations. Fig. 3(b) demonstrates that massive user functions are quite cold and rarely invoked, e.g., 81% of the functions are invoked once per minute or less on average. Therefore, a small number of hot functions can be invoked by most user requests and occupy the majority of hardware resources.

C. Motivating Scorpius

The function startup is too long and this could limit its large-scale deployment in reality. Indeed, many researches have attempted to reduce the startup overhead of FaaS platforms. For example, gVisor [8] and FireCracker [9] optimize the hypervisor as well as kernel to reduce the sandbox booting latency. Replayable Execution [10] and Catalyzer [11] utilize the snapshot technique [20] to mitigate the application launching overhead. These methods can reduce startup latency to some extent, but they all *ignore* the prepare code phase that occupies most of the startup overhead.

Up till now, some recent researches have noticed this phase, and we can divide them into two categories. As Fig. 4 shows, the first category optimizes from the space dimension, it targets at reducing the code size and therefore slashing the code preparation time. The second category optimizes from the time dimension, it focuses on advancing the start time to reduce the waiting time of code preparation. TABLE I shows the comparison of state-of-the-art code preparation works. The package-based methods (e.g., REG [13]) usually prepare public libraries with package management software, and functions that run on the same server can share these libraries. However, the

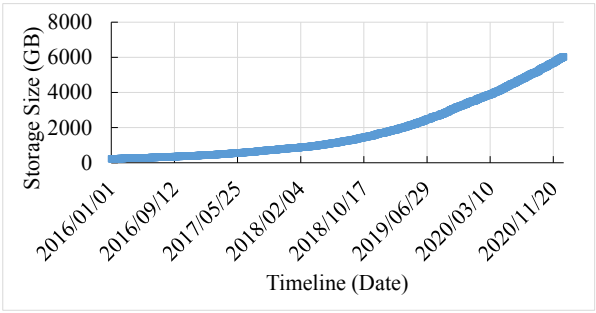


Fig. 5: Storage overhead of library repository over time

library sharing is opportunistic and cannot effectively reduce the preparation latency for most functions (evaluated in §IV-E). The repository-based methods (e.g., SOCK [7]) usually pre-prepare all public libraries on local disks as a repository and can skip all library preparation when starting functions. However, it merely trades storage for latency and each server rather than cluster requires to hold a local repository. We analyze the upload time of various libraries and show the storage size of library repository over time in Fig. 5. As we can see, the storage overhead of public libraries increases rapidly in recent years. Pre-preparing all libraries on a local disk may be possible at the time of SOCK but not now, and this solution is not practical in reality. The histogram-based methods (e.g., Azure [14]) rely on statistical histograms to predict the request arrival. However, it only focuses on the optimization of cold start and ignores the scale out for hot functions which handle most requests (See Fig. 3(a)). The ML-based methods (e.g., Sinan [15]) utilize machine learning models to predict resource requirements for microservice. However, the machine learning model can consume massive computing resources and is not scalable for the FaaS platform since most functions are rarely invoked (See Fig. 3(b)). Therefore, none of above methods are good solutions and this also raises a problem: *Can we design a method that significantly reduces the code preparation latency with little resource overhead?*

Our approach. The goal of Scorpius is effectively reducing the latency of code preparation. Besides, it should only occupy little storage and computing resources, and is also able to handle the function scale out.

III. Scorpius Design

In this section, we first introduce the overview of Scorpius. Then, we describe the specific designs of Scorpius. Main notations can be found in TABLE II.

Variables	Explanation
L	library set that is available in public repository
F	all functions collected from history data
K	server set that is used for FaaS platform
$\|A\|$	element number of set A
s_i	storage size of library i
r_i	import frequency of library i
t_i	number of times that library i is chosen
W_k	storage capacity of server k
α_k	storage proportion of server k for library
U_k	library set for server k
V_k	storage size of the library set for server k
G_f	libraries that are imported by function f
f_i	whether library i is imported by function f
x_{ik}	whether library i is prepared on server k
ϕ_{fk}	shared library size for function f on server k
$\epsilon(\phi)$	keep the maximal value in each row for matrix ϕ
C_{ij}	closeness degree between library i and j
$\delta_n(C)$	the Nth largest number of array C
D_{ik}	closeness degree between library i and server k
M_{ik}	value of raw i column j in server-library matrix
T_0	time that required to prepare codes on server
P_0	probability that prepared codes are sufficient
a	resource usage of a function
b	next resource demand that requires another code
$P_t(r\|a)$	probability that resource usage of a function is a, and its resource demand is r after time t

TABLE II: Explanation of relevant variables

A. Scorpius overview

To effectively reduce the latency of code preparation, Scorpius *proactively prepares the code based on the historical data of functions*. Besides, Scorpius is a *hybrid* method. It optimizes code preparation from both the space and time dimensions, i.e., it not only reduces the code size but also advances the start time. Furthermore, Scorpius is well designed to overcome the shortcomings of the two optimization categories in TABLE I. For the code size optimization, the package-based methods (e.g., REG [13]) can be regarded as *reactive* methods, i.e., they prepare most libraries after the demands arrive. On the contrary, the repository-based methods (e.g., SOCK [7]) advocate to *proactively* prepare all libraries, which only simply trade the storage overhead for preparation latency. Based on this, we propose to *proactively* prepare *partial* libraries over servers and run functions on the server with most library sharing to reduce required code. For the start time optimization, the histogram-based methods (e.g., Azure [14]) only focus on the function *cold start* while the ML-based methods (e.g., Sinan [15]) rely on the machine learning model, which is *too heavy* for sporadic functions. Based on this, we mainly focus on the function scale out in this paper. In practice, Scorpius is orthogonal to the histogram-based methods and we can combine them to optimize the start time. Specifically, we propose to predict the function overload with a *simple* model and *proactively* scale code

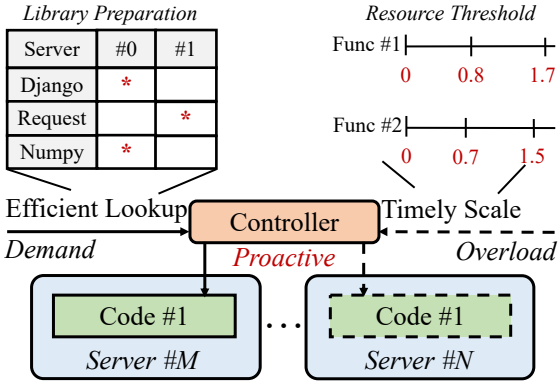


Fig. 6: Overview of Scorpius

to more servers to reduce waiting time. Fig. 6 shows the overview of Scorpius, it proactively prepares partial libraries on specific servers. When a function demand arrives, Scorpius can efficiently lookup a low latency server with most library sharing. When a function is going to suffer from overload, Scorpius proactively scales the code to more servers. To identify the trade-off between latency and resources introduced above, we would like to address the following two challenges:

Challenge 1: how to proactively prepare and share public libraries over servers. According to our measurement in Fig. 2(a), public libraries occupy the most sizes of codes. To reduce the latency of code preparation, we should *proactively prepare partial public libraries over servers* to reduce required codes. However, as Fig. 2(b) shows, public libraries have different import frequencies, and how to prepare and share these public libraries over servers according to the import frequency and clustering feature is a big challenge.

Challenge 2: how to proactively scale code to more servers with a simple model. According to our measurement in Fig. 3, most requests are handled by hot functions. When a function is going to become hot, we should *proactively scale the code of the function to more servers* to provide elastic resources. However, accurate prediction model can consume massive resources, and how to scale code to more servers with a simple but effective model is a big problem.

To overcome these challenges, as Fig. 6 shows, Scorpius proposes two designs, i.e., proactive library preparation algorithm and code scaling mechanism. The library preparation algorithm utilizes the historical data of user functions to guide the library preparation over all servers. Besides, Scorpius exploits the import frequency and clustering feature of libraries to improve the usage efficiency of reserved storage. The code scaling mechanism utilizes a simple threshold model to predict the overload of functions. Furthermore, Scorpius carefully sets the initial thresholds and dynamically adjusts them to match the actual situation of individual function.

B. Proactive library preparation

1) Library preparation algorithm: The difficulty of the problem lies in how to reduce most preparation latency with little storage overhead. Randomly placing libraries onto servers is simple but can be far from ideal. To address the problem,

different from existing works [13], [7], we propose to learn the library preparation from the history data of user functions. Our assumption is that functions have similar import patterns on libraries, i.e., some libraries are frequently imported. Next, we first formulate the problem with an NLIP model. Then, we propose a heuristic algorithm for this problem.

Problem formulation. Let L denote the library set that is available in the public repository. For a library $i \subset L$, its storage size is s_i. Let F present all functions collected from history trace. Given a function $f \subset F$, whether library i is imported by this function is denoted as f_i. We assume the server set that is used for FaaS platform is K. Given a server $k \subset K$, the storage capacity of the server is W_k, and the storage proportion that can be used for library is α_k. We set a binary variable x_{ik} to indicate whether library i is prepared on server k. When function f is placed on server k, the library size it can share is defined as ϕ_{fk}. Besides, we always place functions on the server with most library sharing. We define a matrix operation $\epsilon(\phi)$, which keeps the maximal value in each row and sets the value of other locations to 0. For an arbitrary set A, $|A|$ represents the element number of the set. Our objective is to maximize the aggregate library size across all functions when placed on these servers, which is formulated as,

$$\max \sum_{f \in F} \max_{k \in K} \{\phi_{fk}\} \tag{1}$$

where

$$\phi_{fk} = \sum_{i \in L} s_i \cdot f_i \cdot x_{ik}, \quad \forall f \in F, k \in K \tag{2}$$

s.t.

$$\sum_{f \in F} \epsilon(\phi_{fk}) \leq W_k, \quad \forall k \in K \tag{3}$$

$$\sum_{i \in L} s_i \cdot x_{ik} \leq \alpha_k \cdot W_k, \quad \forall k \in K \tag{4}$$

More specifically, Eqn. (2) describes the relationship between library preparation strategy and shared library size. Constraint (3) and (4) specifies the storage capacity for function placement and library preparation in each server.

To prepare libraries on servers with most library sharing, a naive method is brute-force search, but it can be expensive since its time complexity is $O(2^{|L| \cdot |K|})$, where $|L|$ is the total number of libraries (e.g., 64k) and $|K|$ is the total number of servers (e.g., 64). Besides, the Multidimensional Assignment Problem (MAP), which is known to be NP-hard [21], [22], can be regarded as a special case of our problem. It is hard to find a polynomial-time algorithm to derive the optimal solution. Therefore, we propose a heuristic algorithm to find a reasonable solution for the problem.

Heuristic algorithm. To guide the design of our algorithm, we analyze the relationship between functions and libraries, and highlight two observations as well as design principles: (1) the import frequency of libraries varies significantly. As Fig. 2(b) shown, few well-designed libraries (e.g., `flask`) are excessively hot and frequently imported. Thus, we should first prepare these hot libraries. Besides, as constraint (3) described, the number of functions that can be placed onto each server is limited, some excessively hot libraries should be prepared in multiple copies. (2) the usage of libraries shows clustering feature. For example, the `request` and `json` libraries are often used together to support HTTP. Thus, we should try to place these libraries on the same server.

Based on these observations, we propose a heuristic algorithm with greedy idea to simplify the complexity. At each step, the algorithm chooses a library and places it on a server, until the storage for libraries in all servers is occupied. Given a library $i \subset L$, r_i denotes its import frequency. Given a server $k \subset K$, U_k is the library set for server k, V_k is the storage size of U_k. Given a function $f \subset F$, the libraries that are imported by the function are denoted as G_f.

In response to the import frequency, we define an *enthusiasm degree* for each library and choose the library with the highest degree at each step. First, the degree should be positively correlated with the import frequency to prioritize hot libraries. Second, once a library is chosen, its degree should decrease quickly to handle the extremely skewed import frequency. Let t_i denote the number of times that library i is chosen, the enthusiasm degree of library i can be defined as:

$$E_i = \frac{r_i}{t_i^2} \tag{5}$$

To satisfy the clustering feature, we define a *closeness degree* to quantify the appeal between libraries. Let C_{ij} denote the closeness degree between library i and j, and it should be positively correlated with the number of times the two libraries are used together. The closeness degree between library and server is the sum of the closeness degrees between the library and these libraries on the server. However, we only sum the most N (e.g., 9) closest libraries to prevent the weakly correlated libraries on the server from influencing the degree. At each step, we calculate the closeness degree between the library and each server, and place the library on the server with the highest degree. Let $\delta_n(C)$ denote the Nth largest number of array C, the closeness degree between library i and server k can be defined as:

$$D_{ik} = \sum_{n=1}^{N} \delta_n(C_{ij}), \quad \forall j \in U_k \tag{6}$$

We now give our heuristic library preparation algorithm, which is shown in Algorithm 1. Firstly, it derives the import frequency of libraries and closeness degree between libraries with all function data (Line 1-2). Then, it initializes the enthusiasm degree for libraries, and places each server with a frequently imported but different library (Line 3-5). It loops to prepare libraries (Line 6-14), until the storage for libraries in all servers is occupied. In each iteration, it first finds the library with the highest enthusiasm degree (Line 7-8), and the server with the highest closeness degree for this library

(Line 9-10). Then, it puts the library on the server and updates corresponding library information for this server (Line 11). If all storage for libraries in the server is occupied, it records prepared libraries in the server-library matrix, and gets this server out of subsequent iteration (Line 12-14). Finally, the algorithm can return a server-library matrix (i.e., M), which contains the library preparation of each server.

Algorithm 1: Heuristic Library Preparation Algorithm

Input: Notations shown in TABLE II;
Output: M_{ik};

1 $r_i = \sum_{i \in G_f} |f|, \forall f \in F, i \in L$;
2 $C_{ij} = \sum_{i \cup j \in G_f} |f|, \forall f \in F, i \in L, j \in L, i \neq j$;
3 $t_i = 1, E_i = r_i, \forall i \in L$
4 $U = U \cup \{\arg_{i \in L} \delta_n(r_i)\}, \forall n \leq |K|$
5 $V = V \cup s_{\arg_{i \in L} \delta_n(r_i)}, \forall n \leq |K|$
6 **while** $U \neq \emptyset$ **do**
7 $\quad p = \arg_{i \in L} \max E_i$;
8 $\quad t_p = t_p + 1, E_p = \frac{r_p}{(t_p+1)^2}$;
9 $\quad D_{pk} = \sum_{n=1}^{N} \delta_n(C_{pj}), \forall k \in K, j \in U_k$
10 $\quad q = \arg_{k \in K} \max D_{pk}$;
11 $\quad U_q = U_q \cup p, V_q = V_q + s_p$;
12 $\quad$ **if** $V_q > \alpha_q \cdot W_q$ **then**
13 $\quad\quad M_{iq} = s_i, \forall i \in U_q$
14 $\quad\quad U = U \setminus U_q, V = V \setminus V_q$
15 **return** M_{ik};

We consider an example of three libraries (`numpy`, `request` and `django`) and three servers (#0, #1 and #2) to illustrate our algorithm. Assume the size of the three libraries is 18.5MB, 3.3MB, 41.5MB, respectively. We further consider there are three functions in history and their library imports are [`numpy`, `request`], [`django`] and [`numpy`], respectively. At first, we place each server with a different library. Then, we choose the library with the highest enthusiasm degree (e.g., `numpy`) and place it on the server with the highest closeness degree (e.g., server #1). Fig. 7 shows the server-library matrix after this step. The specific number in the matrix represents the storage size of corresponding library.

2) Efficient server lookup: After the preparation of libraries, we should find the execution locations with the least startup latency for functions. We aim to develop an efficient server lookup method with fast query speed.

Our core idea is to derive the function execution location with a matrix operation. Users usually upload source codes to FaaS platform. Therefore, we can obtain library import information directly from user codes. Besides, users can actively provide library names in a separate file. Once a function is uploaded, imported libraries are known. A function $\mathbf{f}$ can be denoted as $[f_1, ..., f_i, ...]$, where $f_i \in \{0, 1\}$ denotes whether library i is imported ($f_i = 1$) or not ($f_i = 0$) by this function. With the server-library mapping table M, our server lookup method that tries to find the location can be as:

Server Matrix

Library \ Server	#0	#1	#2
Django	0	0	41.5
Request	0	3.3	0
Numpy	18.5	18.5	0

Function Array

Func #1
0
1
1

Fig. 7: Library representation for servers and functions

$$q = \arg_{k \in K} \max \sum_{i \in L} M_{ik} \times f_i \tag{7}$$

Fig. 7 shows an example of our server lookup method. We optionally prepare three libraries (`django`, `request` and `numpy`) on three servers (#0, #1 and #2). The function array represents the function imports the `request` and `numpy`. When the function demand arrives, we can utilize a matrix multiplication of the server matrix and function array, to find that this function should be executed on server #1 since it can save most library storage.

Hot functions can be executed on multiple servers at the same time. For the subsequent codes of the same hot function, we can skip the already used servers and repeat Eqn. (7) to find the execution locations. Therefore, the execution location and preparation time of each code can be known in advance. Besides, with the deployment of user functions, the libraries on each server can vary accordingly. However, we ignore their effects on the placement of functions and leave this optimization as our future work.

C. Proactive code scaling

1) Simple threshold model: The difficulty of the problem lies in how to build a simple but effective model for code scaling. To satisfy the demand, we propose a threshold model to dynamically scale code. Specifically, we set a threshold array for each function, and scale code when its resource usage reaches a certain threshold. For example, we can set an array [0.7, 1.8, ...] for `func1`. When the resource usage of `func1` reaches 0.7 or 1.8, we start to prepare the second or third code. Essentially, this model is predicting the requirement of code scaling based on current resource usage. Compared with existing optimizations on the start time [14], [15], our threshold model can timely scale the code to more servers with little computing overhead.

2) Effective threshold optimization: The key factor is how to set thresholds for a newly deployed function to achieve timely code scaling. A naive method is manually setting an empirical threshold array. However, this method is far from ideal. First, the feature (e.g., preparation time) and demand (e.g., service quality) of functions can be diversified, and a single threshold array cannot satisfy all functions. Second, setting an individual threshold array for each function can be time-consuming. Besides, if these thresholds are set too low or too high, they can waste excessive storage resources, or cannot prepare code before actual usage.

To satisfy diversified properties of functions, we exploit to automatically set thresholds for individual functions. Thus, we

formulate the problem with our model to guide the setting of thresholds. The diversified properties we considered here are preparation time and expectation probability. The first one is the time that we prepare the codes of a function on a specific server and is defined as T_0. The second one represents the probability that we expect the prepared codes to be sufficient for user requests and is denoted by P_0. The threshold indicates that when resource usage of a function reaches this value, the probability that prepared codes are sufficient in the next period of time (T_0) is less than user expectation (P_0), which means code scaling is required. We assume that the resource usage of a function is a, and the next resource demand that requires another code is b. Then, the threshold is the first value for a that satisfies the following inequation:

$$\prod_{t=0}^{T_0} \sum_{r=0}^{b} P_t(r|a) < P_0 \tag{8}$$

where $P_t(r|a)$ is the probability that resource usage of a function is a, and its resource demand is r after time t. Since T_0 and P_0 are known for functions, b represents all integers, the key factor for threshold calculation is $P_t(r|a)$.

The intuitive meaning of $P_t(r|a)$ is predicting the resource demand of functions in the next period of time based on current resource usage. However, resource prediction is difficult especially for new deployed functions, since we lack the historical data of their resource usage. To address the challenge, we propose to learn the variation pattern from the invocation traces of other functions to set initial thresholds. Our assumption is that functions have similar variation patterns on resource usage, especially for the same category (e.g., event handler). Besides, we dynamically adjust these thresholds to better match the actual usage situation of this function.

Initial thresholds. To reduce the overhead of computing resources, we propose to approximate the variation pattern with statistics from real-world traces [14]. Specifically, we record the resource usage of functions at different times and count appeared frequency of different resource demands after a period of time. Then, we *treat appeared frequency as possible probability*. To improve the accuracy of prediction, we further classify these functions according to the category of function requests (e.g., event or queue) and utilize statistics from the same category to calculate the thresholds.

Dynamic adjustment. The behaviors of thresholds under actual situations can be classified into three categories: (1) `late`, which means code is not ready when required; (2) `early`, which means code is prepared before required; and (3) `exactly`, which is best but happens rarely under the time granularity of seconds. Both `late` and `early` are harmful since `late` can increase latency while `early` can waste storage. Thus, we need to adjust these thresholds to better match the feature of individual functions. Our observation is that `late` is related to user experience and more unacceptable than `early`. Therefore, inspired by the congestion control algorithm in TCP, we propose to *decrease all and increase finite* for threshold adjustment. When `late` happens, we decrease all subsequent thresholds within its range by one unit. If `early` happens, we only increase finite subsequent thresholds (e.g., 3) within its range by one unit.

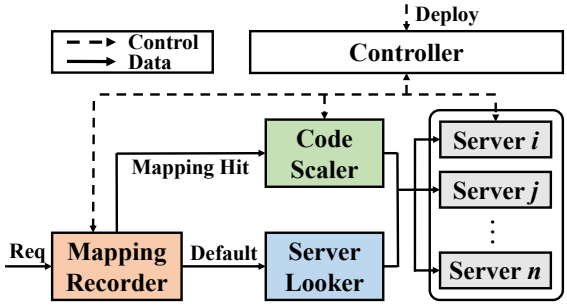


Fig. 8: Architecture and workflow of Scorpius

IV. Architecture and Evaluation

A. Scorpius architecture and workflow

Fig. IV-A shows the architecture and workflow of Scorpius. Colorful blocks shown in the architecture are responsible for request delivery and are the main modules introduced into existing FaaS platforms. Next, we describe these modules and their corresponding workflow in detail.

Mapping Recorder. Mapping Recorder is responsible for maintaining function-servers mappings for functions. It utilizes a hash table to store these mappings and each function can be mapped to a server list. Each arrival request requires lookup this table with its piggyback function information. If the function is alive, a server list that has prepared corresponding function code can be obtained.

Server Looker. Server Looker is responsible for finding a low latency server for functions according to our lookup method. During the initialization of FaaS platform, Scorpius requires learning the library preparation from existing functions based on Algorithm 1 and storing the status of library preparation as a server matrix of Fig. 7 into Server Looker. On receiving a demand for function startup, it extracts the library information from the function and only requires a matrix multiplication to identify the targeted server.

Code Scaler. Code Scaler is responsible for monitoring the status of functions and dynamically scaling codes for functions. During the initialization of FaaS platform, Scorpius requires learning the initial function threshold from existing traces and storing the thresholds about functions as a two-dimensional table into Code Scaler. On receiving a request for a function, it compares current request rate with defined thresholds and actively triggers the code scaling once the request rate exceeds these thresholds.

Therefore, Scorpius works as follows: On receiving a function deployment requirement, *Controller* extracts corresponding library information from the function. When the first request of the function arrives, it is delivered to the *Server Looker* and utilizes library information to choose the execution location. Once the request rate exceeds its defined thresholds during subsequent periods, the code of this function is scaled to multiple servers. Function-servers mappings are

dynamically added to *Mapping Recorder* and all requests for alive function are continuously monitored by *Code Scaler*.

B. Simulator and workloads

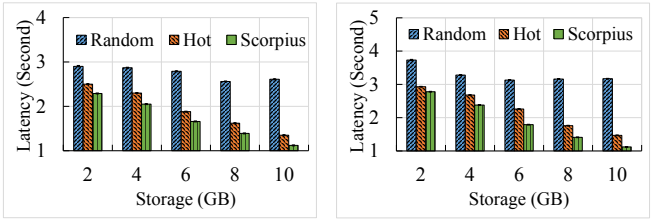
To evaluate the optimization of our designs for function startup, the most effective way is to utilize user functions to conduct large-scale experiments in real environments. However, it is hard to satisfy even for large cloud provider [14]. First, code preparation involves the management of massive servers (e.g., 64) and it is not cost-effective to reserve so many resources. Second, user functions are private for providers and it is almost impossible to expose such confidential information. Therefore, similar to Microsoft [14], we show the benefits of Scorpius with both real system and simulator. First, we utilize real system to evaluate the resource overhead of server lookup and code scaling. Second, we build a simulator, which follows the framework of most open-source FaaS platforms [23], [12], and utilize real-world workloads as the input [14], [7], to evaluate the effects of our designs.

Workloads. As the input to our simulator, all data is extracted from real workloads. To approximate user codes in FaaS platforms, we scrape 64k python projects with more than 50 stars from the GitHub and treat open-source projects as possible user functions [7]. We extract imported libraries from these projects and filter them in the public Python Package Index (PyPI) repository. Besides, we build a local GitLab and repository to collect storage size and preparation time (including network latency) of user codes and public libraries. The traces we utilize are from real users in a large cloud provider [14]. Since these traces are recorded at the granularity of minutes, we complete them to the granularity of seconds with linear interpolation. Finally, we equally split these functions and traces, treat half of them as training data and utilize the remaining half for evaluation.

Simulator. The modules of our simulator are basically consistent with the architecture of Scorpius. For library preparation, it first learns initial library distribution with our heuristic algorithm from existing functions. Then, it places the remaining functions on multiple servers with the lookup method, and records the placement and corresponding storage size of all servers. For code scaling, it first learns initial threshold setting with our statistical algorithm from existing traces. Then, it dynamically scales codes according to resource usage extracted from the remaining traces and records the time of code scaling and actual requirements.

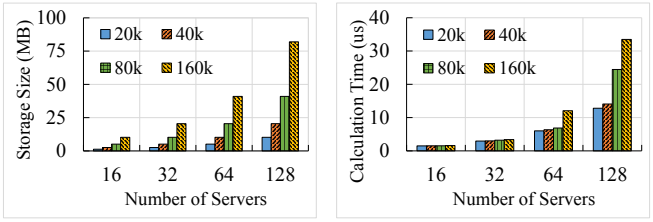
C. Library preparation

Optimization on preparation latency and storage overhead. To reduce preparation latency with little storage overhead, Scorpius carefully considers the import frequency and clustering feature in the design of library preparation. To demonstrate their effects, we place the same function set on a server cluster with different library preparation methods. Fig 9 shows the evaluation results with different server numbers. First, different server numbers show similar results for these preparation methods. Second, compared with the randomly prepared libraries (Random), the algorithm that only considers import frequency (Hot) can reduce the preparation latency by 18.2%∼53.6% with the same storage overhead. Third, Scorpius further combines the clustering feature to reduce the preparation latency by 5.1%∼23.8%, and can outperform the randomly prepared libraries by 25.4%∼64.6%.



(a) Comparison with 32 servers (b) Comparison with 64 servers

Fig. 9: Optimization on preparation latency and storage overhead for library preparation



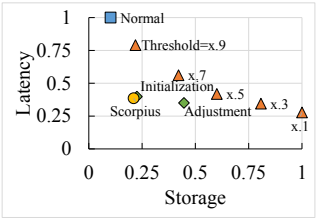
(a) Storage size with servers (b) Calculation time with servers

Fig. 10: Overhead of server lookup method

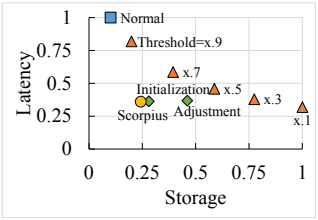
Overhead of server lookup method. To demonstrate the advantage of Scorpius on server lookup, we evaluate the storage and computing overhead of the server lookup method. Storage overhead mainly comes from the matrix that records library preparation status. Fig. 10(a) shows the storage size of different library numbers with increasing servers. As we can see, the storage size increases linearly with the number of servers and libraries. However, even with 128 servers and 160k libraries, the total storage size is only 81.9MB. Computing overhead mainly comes from the matrix multiplication that identifies the targeted server for functions. Fig. 10(b) shows the calculation time of different library numbers with increasing servers. As we can see, the calculation time increases linearly with the number of servers. Besides, even with 128 servers, the average calculation time is merely 33.5μs.

D. Code scaling

Optimization on preparation latency and storage overhead. Regular code scaling (e.g., on demand) absents design for preparation latency, Scorpius considers the optimization on preparation latency and storage overhead with well-set thresholds. To demonstrate their effects, we scale code to satisfy the same real-world traces with different scaling methods. Since preparation latency is directly related to user experience, we utilize the sum of the squares of time difference to quantify preparation latency. Besides, storage overhead can be extremely long (e.g., redundant code), we utilize the sum of the logarithms of time difference to quantify storage overhead.

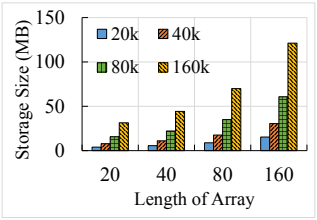


(a) Comparison for `event` traces

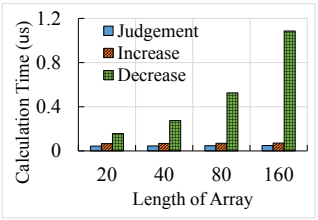


(b) Comparison for `queue` traces

Fig. 11: Optimization on preparation latency and storage overhead for code scaling



(a) Storage size with array length



(b) Calculation time with array length

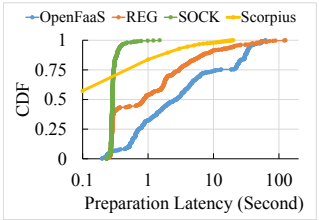
Fig. 12: Overhead of code scaling mechanism

Fig 11 shows the normalized results with different request categories. First, different request categories have little influence on scaling methods. Second, compared with on-demand scaling (Normal), our scaling mechanism that utilizes thresholds (Threshold) effectively trades storage for latency with different threshold settings. Third, compared with the constant threshold (e.g., x.5), careful threshold initialization (Initialization) can reduce the preparation latency and storage overhead by 52.1% and 20.8%, while dynamic threshold adjustment (Adjustment) can reduce them by 21.6% and 19.8%. Besides, Scorpius combines the advantage of careful initialization and dynamic adjustment to achieve the best behavior, and can reduce the two aspects by 58.6% and 21.5%.

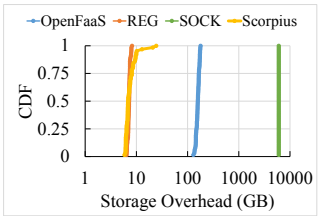
Overhead of code scaling mechanism. To demonstrate the benefit of Scorpius compared with the complicated prediction model, we evaluate the storage and computing overhead of code scaling mechanism. Storage overhead mainly comes from the statistics on resource usages and threshold arrays. Fig. 12(a) shows the storage size of different function numbers with increasing array length. As we can see, the storage size increases linearly with the length of array and number of functions. However, even with 160 thresholds and 160k functions, the total storage is only 121.2MB. Computing overhead mainly comes from the threshold judgement and dynamic adjustment. Fig. 12(b) shows the calculation time of different threshold operations with increasing array length. As we can see, the calculation time of threshold judgement and increase is almost constant, and the longest calculation time of threshold decrease is still no more than 1.1μs.

E. Overall improvement

Preparation latency and storage overhead for different methods. Existing methods either reactively prepare codes (e.g., REG [13]) or pre-prepare all libraries on each server



(a) CDF of code preparation latency



(b) CDF of code storage overhead

Fig. 13: Overall comparison on preparation latency and storage overhead with different methods

(e.g., SOCK [7]), Scorpius prepares partial libraries over servers and runs function on the server with most library sharing. Besides, Scorpius optimizes the start time for function scale out and is complementary to the methods designed for function cold start (e.g., Azure [14]). To demonstrate the overall improvement of Scorpius, we compare the preparation latency and storage overhead of different methods (computing overhead has been evaluated above). Since the request traces and scraped functions are separate, we randomly combine trace and function to form a complete record. Besides, we choose moderate parameters for Scorpius (e.g., 64 for server number) and replay 14-days traces (from Azure [14]) to evaluate these methods. Fig. 13(a) shows the CDF of preparation latency with different methods. Compared with the package-based method (REG), Scorpius can reduce the average latency from 4.58s to 0.57s with about 87.6% improvement. Fig. 13(b) shows the CDF of storage overhead with different methods. Compared with the package-based method (REG), Scorpius could increase the total storage from 460GB to 503GB with only 9.3% overhead. Compared with the repository-based method (SOCK), which prepares all libraries on each server in advance, Scorpius can significantly reduce the storage overhead by up to three orders of magnitude at the cost of only doubling the preparation latency.

V. Related Works

A. FaaS platform

Existing researches on FaaS platform can be roughly classified into two categories. The first category focuses on leveraging FaaS platform to support various applications [24], [25]. For example, Fouladi et al. [24] design a mechanism to run video processing tasks on AWS Lambda. Jonas et al. [26] design a framework to support distributed computing for the majority of users. gg [25] designs a framework to execute everyday applications on function service. The second category targets at improving FaaS platforms for various purposes [27], [6]. For example, Pocket [27] designs elastic ephemeral storage for analytics. Boucher et al. [3] adopts language-based isolation to reduce process launching time. SAND [6] designs a specific FaaS platform to support complex applications. However, none of them are designed to accelerate code preparation. Scorpius is the first to identify code preparation as the main overhead of function startup and proactively prepare code to mitigate corresponding overhead.

B. Function placement

Many researches have exploited the placement to optimize performance. Some focus on placing multiple services on the same server to relieve communication overhead [28], [29]. For example, FAASM [29] aggregates all functions into a process to support memory sharing. NetVM [28] consolidates multiple network functions into a server to speedup packet delivery. Some propose to place computing tasks into storage nodes to mitigate access overhead [30], [31]. For example, Splinter [30] ships computing to storage to avoid round trips and data movement. Malacology [31] extends the storage system to support flexible computing tasks. However, none of existing researches utilizes placement to accelerate code preparation. Scorpius fully leverages the library import information of functions and place functions into the server with most library sharing to slash startup overhead.

VI. Conclusion

We have presented Scorpius, a FaaS platform designed for code preparation. Compared with existing works, Scorpius proactively prepares the code based on the historical data of functions, and combines two optimization categories to accelerate function startup. First, it proposes a library preparation algorithm to reduce the code size. Then, it designs a code scaling mechanism to advance the start time. Evaluation results demonstrate that compared with state-of-the-art methods, Scorpius can reduce the code preparation latency by 87.6% with only 9.3% storage overhead.

Acknowledgement

We thank anonymous IWQoS reviewers for their valuable comments. This research is supported by Joint Research on IPv6 Network Governance: Research, Development and Demonstration (2020YFE0200500) and the Postdoctoral Science Foundation of China (2021 M690338). Han Zhang is the corresponding author.

References

[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009, pp. 51–62.

[2] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *SIGCOMM on Applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 199–210.

[3] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the" micro" back in microservice," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 645–650.

[4] S. Hendrickson, S. Sturdevant, T. Harter *et al.*, "Serverless computation with {OpenLambda}," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[5] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, pp. 44–54, 2019.

[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 *Usenix Annual Technical Conference*, 2018, pp. 923–935.

[7] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 57–70.

[8] Google. Google gvisor. https://github.com/google/gvisor.

[9] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[10] 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 2019*, 2019, pp. 1–16.

[11] D. Du, T. Yu, Y. Xia *et al.*, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 467–481.

[12] OpenFaas. (2019) Serverless functions made simple. https://github.com/openfaas/faas.

[13] W. Wang, L. Zhang, D. Guo, S. Wu, H. Cui, and F. Bi, "Reg: An ultra-lightweight container that maximizes memory sharing and minimizes the runtime environment," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 2019, pp. 76–82.

[14] M. Shahrad, R. Fonseca *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference*, 2020, pp. 205–218.

[15] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[16] Y. Gan, Y. Zhang, D. Cheng *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[17] (Jan, 2021) 4 microservices examples: Amazon, netflix, uber, and etsy. https://blog.dreamfactory.com/microservices-examples/.

[18] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[19] Github. Github. https://github.com/.

[20] CRIU. (2020) Checkpoint/restore in userspace. https://criu.org/Main_Page.

[21] Y. Jiang, X. Shen, C. Jie, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2008, pp. 220–229.

[22] M. Cieliebak, S. J. Eidenbenz, A. Pagourtzis, and K. Schlude, "On the complexity of variations of equal sum subsets." *Nord. J. Comput.*, vol. 14, no. 3, pp. 151–172, 2008.

[23] knative. (2020) Kubernetes-based platform to deploy and manage modern serverless workloads. https://knative.dev/.

[24] S. Fouladi, R. S. Wahby, B. Shacklett *et al.*, "Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 363–376.

[25] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[26] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 445–451.

[27] A. Klimovic, Y. Wang, P. Stuedi *et al.*, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 427–444.

[28] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[29] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 419–433.

[30] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, "Splinter:{Bare-Metal} extensions for {Multi-Tenant}{Low-Latency} storage," in *13th USENIX OSDI*, 2018, pp. 627–643.

[31] M. A. Sevilla, N. Watkins, I. Jimenez *et al.*, "Malacology: A programmable storage system," in *Proceedings of the European Conference on Computer Systems*, 2017, pp. 175–190.