

FaaSBatch: Boosting Serverless Efficiency With In-Container Parallelism and Resource Multiplexing

Zhaorui Wu , Yuhui Deng , Yi Zhou , Jie Li , Shujie Pang , and Xiao Qin 

Abstract—With high scalability and flexibility, serverless computing is becoming the most promising computing model. Existing serverless computing platforms initiate a container for each function invocation, which leads to a huge waste of computing resources. Our examinations reveal that (i) executing invocations concurrently within a single container can provide comparable performance to that provided by multiple containers (i.e., traditional approaches); (ii) redundant resources generated within a container result in memory resource waste, which prolongs the execution time of function invocations. Motivated by these insightful observations, we propose *FaaSBatch* - a serverless framework that reduces invocation latency and saves scarce computing resources. In particular, *FaaSBatch* first classifies concurrent function requests into different function groups according to the invocation information. Next, *FaaSBatch* batches the invocations of each group, aiming to minimize resource utilization. Then, *FaaSBatch* utilizes an inline parallel policy to map each group of batched invocations into a single container. Finally, *FaaSBatch* expands and executes invocations of containers in parallel. To further reduce invocation latency and resource utilization, within each container, *FaaSBatch* reuses redundant resources created during function execution. We conduct extensive experiments based on Azure traces to evaluate the effectiveness and performance of *FaaSBatch*. We compare *FaaSBatch* with three state-of-the-art schedulers *Vanilla*, *SFS*, and *Kraken*. Our experimental results show that *FaaSBatch* effectively and remarkably slashes invocation latency and resource overhead. For instance, when executing I/O functions, *FaaSBatch* cuts back the invocation latency of *Vanilla*, *SFS*, and *Kraken* by up to 72.58%, 74.10%, and 72.62%, respectively; *FaaSBatch* also slashes the resource overhead of *Vanilla*, *SFS*, and *Kraken* by 70.2% to 98.40%, 67.74% to 98.12%, and 43.01% to 78.90%, respectively.

Index Terms—Cloud computing, serverless computing, batching request, resource management.

I. INTRODUCTION

SERVERLESS computing has been widely adopted by various enterprises due to its rapid deployment, scalability, and low cost. With serverless computing in place, developers only need to upload the code of one or several functions and set trigger events to invoke these functions without concern about underlying infrastructures [1]. And after the functions are registered in a serverless computing platform (i.e., AWS lambda [2], Azure Function [3], OpenWhisk [4], OpenFaaS [5], etc.), they can be invoked by the customers.

In serverless computing, each function is initialized and executed inside a container (or Virtual Machine). If there are no keep-alive containers that can service subsequent invocations (i.e., *warm start*), a new container will be started from scratch (i.e., *cold start*). Because a newly started container in a cold start first creates and starts a runtime environment and then fetches and imports necessary dependencies, a cold start incurs significant overhead [6], [7]. As thousands of function invocations are sent to a serverless computing platform, high concurrency workloads are inevitable [8]. Since conventional serverless platforms launch a specific container for each incoming function invocation [9], [10], [11], serverless computing platforms involve provisioning numerous container instances, which exacerbates resource pressure on infrastructure servers.

A common strategy used by serverless computing providers to alleviate the above-mentioned issue is to resourcefully queue functions by introducing the notion of slack, thereby reducing the number of launched containers [12], [13], [14], [15], [16]. Unfortunately, this strategy introduces a waiting overhead for queued invocations, which results in a latency penalty. That is, the batching strategy sacrifices invocation latency in exchange for resource relaxation. Besides, our evaluation (detailed in Sec. II) shows that executing invocations concurrently within a single container can provide comparable performance to that provided by multiple containers. Therefore, we believe there is an opportunity to execute invocations concurrently within a single container at a low provisioning cost. Moreover, we found that the pattern of container reuse in serverless platforms leads to the creation of redundant function resources. Specifically, due to the nature of stateless, serverless functions make use of provided socket clients to communicate with external storage (i.e., Amazon S3 [17] or Azure Blob storage [18]). And these client instances are created repeatedly when a platform handles multiple function executions. The overhead of these instances

Manuscript received 4 September 2023; revised 8 December 2023; accepted 4 January 2024. Date of publication 11 January 2024; date of current version 13 March 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62072214 and in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2021B1515120048. Recommended for acceptance by J. Zhai. (Corresponding author: Yuhui Deng.)

Zhaorui Wu, Yuhui Deng, Jie Li, and Shujie Pang are with the Department of Computer Science, Jinan University, Guangzhou, Guangdong 510632, China (e-mail: diom_wu@163.com; tyhdeng@jnu.edu.cn; lijieqmd11@163.com; p_shujie@163.com).

Yi Zhou is with TSYS School of Computer Science, Columbus State University, Columbus, GA 31097 USA (e-mail: zhou_yi@columbusstate.edu).

Xiao Qin is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849-5347 USA (e-mail: xqin@auburn.edu).

Digital Object Identifier 10.1109/TC.2024.3352834

created repeatedly is considerably high, especially in cases where function execution times are generally short.

These insightful observations motivate us to devise *FaaSBatch*¹ - a novel serverless framework that tackles invocation latency and resource utilization through request batching, inline parallelism, and a resource multiplexing strategy. Particularly, FaaSBatch is proposed to tackle the following two dilemmas: (i) increasing concurrency achieves the lowest execution latency when spawning containers to service incoming requests for functions, however, this comes at the cost of starting numerous containers; (ii) batching strategy, on the other hand, confines multiple invocations into containers, trading invocation latency for memory resources. To achieve these goals, we introduce three modules, namely, *invocation mapper*, *inline-parallel producer*, and *resource multiplexer* in FaaSBatch. Specifically, we design a module - *invocation mapper* - embedded in a serverless computing platform, which performs batching of concurrent invocations based on conditions to reduce the number of provisioned containers. In this study, we categorize function types into CPU-intensive and I/O functions. When performing CPU-intensive functions, the invocation mapper places all invocations received within a specific time-window interval into a single container instead of batches them in multiple containers. When it comes to I/O functions, the invocation mapper forwards function invocations by identifying the requests and works in conjunction with the resource multiplexer to enhance the efficiency of function execution. Executing all concurrent invocations inside a single container avoids expensive container creation, thereby lowering invocation latency and resource utilization. Second, we develop an *inline-parallel producer* module based on the mapping between invocations and containers so that all batched invocations can be executed in parallel inside containers (detailed in Sec. II). Last, within a container, we propose a resource multiplexing module called *resource multiplexer* to monitor the generation of redundant resources, which facilitates their reuse. Also, an LRU strategy is leveraged to reduce the memory footprint of the redundant resource. In doing so, there is no need to create any additional redundant resources, which further suppresses resource pressure and invocation delay inside a container. The contributions of this study are following.

- We examine the parallel execution of functions and reveal that (i) function invocations concurrently executed within a single container and across multiple containers can deliver similar performance; (ii) within a container, redundant resources cause heavy overheads in terms of invocation latency and resource utilization.
- We design a novel serverless framework called *FaaSBatch* to improve both invocation latency and resource utilization. FaaSBatch integrates three indispensable modules, namely, an *invocation mapper* module, an *inline-parallel producer* module, and a *resource multiplexer* module. Invocation mapper and inline-parallel producer focus on reducing resource utilization by minimizing provisioned

containers, while resource multiplexer is concentrated on improving invocation latency and resource utilization by multiplexing and scheduling redundant resources.

- We conduct extensive experiments to evaluate our FaaSBatch. Experimental results show that FaaSBatch remarkably reduces invocation latency and resource overhead compared with other state-of-the-art approaches.

II. MOTIVATION

FaaSBatch is conducive to packing multiple function invocations into a single container and carrying out invocation executions in parallel inside the container. In this section, we first provide a motivational study of performance measurements to highlight the feasibility of executing invocations in parallel within a container (Sec. II-A). Secondly, we present the necessity of multiplexing redundant resources during function execution (Sec. II-B). Then, in light of the multi-tenant scenarios in serverless computing, we propose the necessity of cache scheduling and invocation mapping strategy specific for I/O functions (Sec. II-C).

A. Sharing Container for Concurrent Invocations

Serverless computing platforms pride themselves on their high concurrency and function processing capability. Conventional serverless platforms launch an individual container for each incoming function invocation. To satisfy consumers' desire for a superior application experience, serverless providers reduce invocation latency at the expense of increased concurrency of function execution. Unfortunately, Modern serverless applications generate massive amounts of concurrency (e.g., 1500 requests per second during peak performance [8]). This burst of invocations pushes traditional serverless platforms into critical cold starts and resource-intensive disasters [1]. Specifically, cold-start overheads affect invocation latency, and the number of started containers exacerbates the memory overhead. To address this issue, we stuff all concurrent invocations into a single container and expand them (i.e., execute them in parallel). In doing so, our proposed FaaSBatch reduces infrastructure resources while maintaining low latency.

To evaluate the effectiveness of our proposal, we conducted a concurrency performance measurement on a single 32-core server with two different mapping manners. Fig. 1 depicts the experimental results of the concurrency measurement, where "Sharing" represents our proposed strategy (i.e., stuffing all invocations into a single container), whereas "Monopoly" represents the traditional scheme that maps each function invocation into a container. To fairly compare the execution times of Sharing and Monopoly, we warm up the container(s) before firing function invocations; we set the degree of concurrency from 10 to 640 for the same CPU-intensive function - computing the Fibonacci series with $N = 30$. Fig. 1 shows that our Sharing strategy delivers similar performance compared to Monopoly. However, our Sharing strategy harvests huge resource savings by launching only one container. Furthermore, since our strategy shares a container for concurrent invocations, it spares memory resources without jeopardizing execution

¹A preliminary short version of this paper [19] appears in Proceedings of the 43rd IEEE International Conference on Distributed Computing Systems (ICDCS 23). We comprehensively investigate semantic of multi-tenant scenario and add extensive experiments in the current paper.

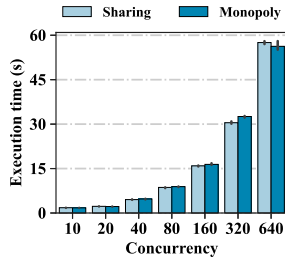


Fig. 1. Execution latency.

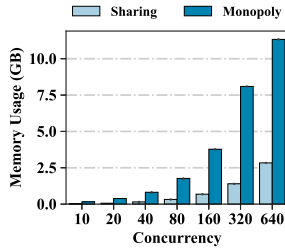


Fig. 2. Memory overhead.

performance. Moreover, Fig. 2 illustrates how memory usage trends vary with the increasing concurrency in both strategies. It can be observed that the Sharing strategy exhibits progressively significant memory efficiency as concurrency increases compared to the Monopoly strategy. This arises from the fact that the Sharing strategy merely entails the initiation of an additional set of threads within a container to handle concurrent invocations. In contrast, the Monopoly strategy requires the creation of a corresponding number of containers. The process of initializing a container is associated with a significantly greater memory overhead than that involved in the initiation of threads. The result substantiates the effectiveness of batching function invocations for concurrency execution within a single container instance.

A well-efficiency sharing scheme also depends on the temporal continuity of function invocations. For some rarely invoked functions (e.g., 1 request per hour), our proposed strategy may fail short of demonstrating the required resource reduction. Fortunately, serverless platforms feature mostly hot invocations. Specifically, more than 99% of overall invocations are occupied by 20% of the popular functions [9]. Digging a bit deeper into the execution characteristics, we analyze the Azure Function trace [11] and plot the experimental result in Fig. 3. Fig. 3 shows the execution pattern of fifteen representative functions (each invoked more than 1000 times by the same user) over a full day. It indicates that the invocation patterns exhibit bursty and tight temporal locality during execution. Specifically, within a 10-milliseconds interval, the number of function invocations received ranges from 2 to 15. Fig. 4 displays the Cumulative Distribution Function (CDF) distribution of time intervals for all function invocations in the Azure Function trace. Globally, it shows that over 40% of function invocations occur within an interval of less than 100 milliseconds, and an additional 40% have intervals ranging from 100 milliseconds to one second.

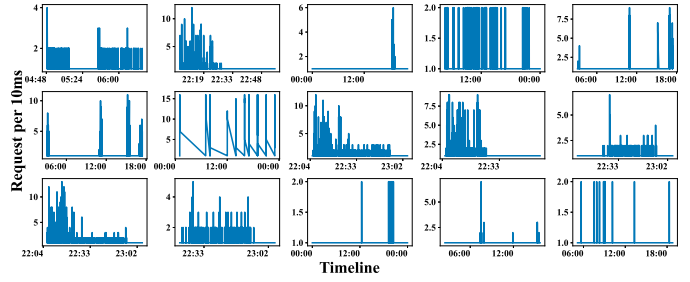


Fig. 3. The temporal locality pattern of different functions in Azure trace.

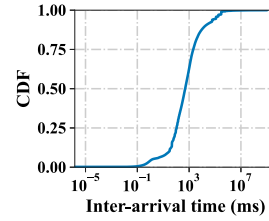


Fig. 4. CDF of requests' IaT.

Such results confirm the feasibility and effectiveness of our proposed strategy since compared with traditional strategies, our proposed strategy can achieve comparable performance with extreme savings in memory resources by launching a single container.

1) *Implication 1: High concurrency of invocations introduces significant overheads due to cold starts. However, there is an opportunity to leverage the serverless computing model (by populating concurrent invocations and expanding inside a single container) to handle bursty and time-localized invocation patterns.*

B. Resource Redundancy During Function Execution

Batching concurrent invocations and executing them inside a single container is a promising approach to relieve the pressure on the underlying infrastructure's memory resources. However, this strategy (as well as the conventional approaches) may lead to resource duplication. Due to the stateless nature, serverless functions have to maintain their generated intermediate data via access to cloud object storage, e.g., AWS S3 [17] or Azure Blob storage [18]. The cloud object storage service allows developers to create socket clients using the provided SDK (Software Development Kit) and API (Application Programming Interface) to perform further operations, such as creating, updating, reading or deleting objects (or CRUD operations) [20]. For example, Azure Blob storage and AWS S3 provide APIs (please see Listing 1) for accessing cloud storage objects. Creating these programming language-level objects (in this case, the clients) requires computational and memory overheads. Although all invocations are executed concurrently in a single container in the form of threads (whose resources are shared inside the process), the programming language still cannot recognize these duplicate objects. Therefore, multiple instances of

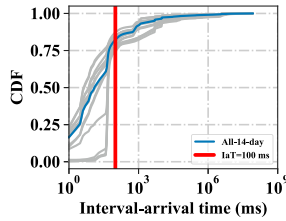


Fig. 5. CDF of blobs' IaT.

the same client reside in container-owned memory, resulting in wasted memory.

Code Listing 1. Creating cloud storage clients

```
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient
# Azure blob client
azure_client = BlobServiceClient(
    "ACCOUNT_URL",
    credential=DefaultAzureCredential()
)
# Do something with the azure_client...
-----
import boto3
# AWS S3 boto3 client
boto3_client = boto3.client("s3",
    aws_access_key_id="ACCESS_KEY",
    aws_secret_access_key="SECRET_KEY",
    aws_session_token="SESSION_TOKEN"
)
# Do something with the boto3_client...
```

To shed light on the accesses patterns of cloud storage objects, we investigate the Azure Blob trace [21], which contains 14 days of logs, including 33.1 million invocations with 44.3 million data accesses. The analysis results are plotted in Fig. 5, which shows the CDF of the inter-arrival time (IaT) for the blobs with more than 2 accesses (there is no IaT otherwise). Fig. 5 plots fifteen curves, with fourteen gray lines representing the CDF of IaT for each day (from day 1 to day 14) and the one highlighted in blue representing the CDF of blob access patterns after consolidating the fourteen days' logs. We can observe from Fig. 5 that nearly 80% of the objects are repeatedly accessed within 100 ms, while the remaining 10% are revisited ranging from 100 ms to 1000 ms. These observations reveal that most blob accesses are bursty rather than smooth, which is consistent with the report from Microsoft Azure [21]. It is noteworthy that when servicing bursty accesses, a large number of function invocations will be executed in parallel within a container. As a result, there will inevitably be duplicate instances of created socket clients. And these repeated creations of socket clients lead to significant overhead. To quantitatively examine this overhead, we plot the overheads of repeated creation of Amazon S3 socket clients within a single container in Fig. 6 and Fig. 7. Fig. 6 shows that an increase in concurrency (from 1 to 10) leads to a performance penalty. For example, the time to create S3 clients increases by almost 50X when increasing the concurrency from 1 (66 ms) to 9 (3 s). Fig. 7 illustrates a rising trend of memory overhead with respect to the repeated creation of S3 socket clients. Specifically, as the concurrency degree increases, the memory consumption of a single container increases from 9 MB to 60 MB.

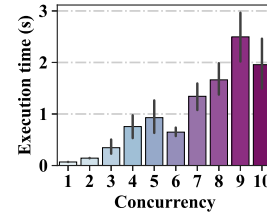


Fig. 6. Created time overhead.

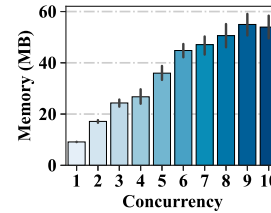


Fig. 7. Used memory overhead.

In a nutshell, though executing invocations in parallel within a container shortens waiting time, the redundant resource creation problem remains, and it may cause to additional overhead. Promisingly, parallel execution of requests in containers helps avoid the creation of redundant resources. The above analysis unveils an opportunity to optimize resource sharing within containers.

1) *Implication 2: Performing concurrent invocations in a fold-spread fashion accounts for resource savings outside containers. And there is still an opportunity to further mitigate redundant resource waste that resides inside containers. We argue that multiplexing the redundant resources created by function execution will significantly optimize execution overheads.*

C. Cache Scheduling Mechanism

Enabling resource sharing for I/O functions within containers can reduce redundant resource creation, especially when implemented through in-memory caching. However, this strategy poses a challenge in serverless computing environments where each identical socket client instance requires a dedicated cache slot, leading to significant memory overhead. To mitigate this, a cache scheduling strategy with a predefined cache size within each container becomes essential. It's worth noting that implementing a caching policy in serverless containers presents challenges similar to those in building distributed caching systems. A carefully designed mapper is essential for efficiently mapping function invocations to containers.

Furthermore, even invocations targeting the same function may contain different parameters, leading to the creation of different socket client instances. This variability can impact the efficacy of the caching mechanism. As shown in Fig. 8, a service provider P_A has registered three different functions with the serverless computing platform to offer a service to front-end users. In this context, the function invoker is P_A , instead of the front-end users. Therefore, if multiple front-end users access a

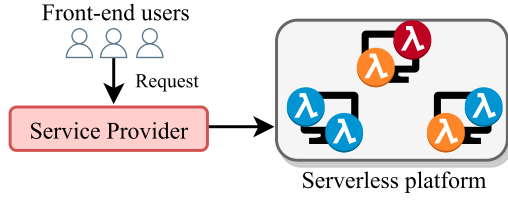


Fig. 8. Application overview of serverless paradigm.

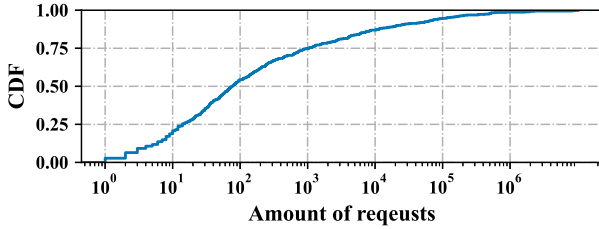


Fig. 9. Number of accesses to different blobs.

particular I/O function provided by P_A concurrently, these function invocations would be routed to the same container under the operation of FaaSBatch. Because these function requests come from different front-end users, it results in the presence of three distinct socket clients within the same container instance. In such scenarios, the invocation mapper should be smart enough to differentiate between these distinct socket client instances created within the same container due to different front-end user requests. This is essential for optimizing resource allocation and avoiding unnecessary memory overhead.

To illustrate the feasibility of cache scheduling in serverless computing environments, we analyzed the access frequency distribution of different blobs based on the Azure Blob trace. Fig. 9 demonstrates the number of accesses to different blobs in Azure Blob trace. Specifically, less than 25% of blobs were accessed fewer than ten times over a 14-days period, the remaining 70% of blobs experienced access frequencies ranging from ten times to several hundred times, with a very small portion of blobs being accessed over a thousand times. This indicates that the access patterns of blobs generally conform to a normal distribution. Hence, we believe that employing a cache scheduling strategy to enhance the access efficiency of hot blobs is feasible.

1) *Implication 3: Despite sharing and multiplexing the redundant resources of I/O functions cuts back the execution overheads, it fails to address the issue of unbounded memory consumption in multi-tenant scenarios. Caching scheduling offers a promising solution for optimizing resource utilization, and combined with smart invocation mapping strategies, we believe effective resource allocation can be achieved.*

III. DESIGN OF FAASBATCH

A. Architecture Overview

The aforementioned inspiring observations and analysis reveal that there exists a pressing demand behind a novel strategy

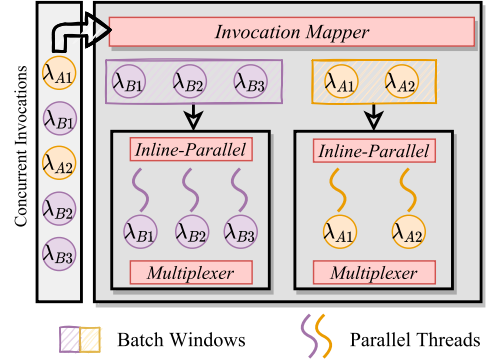


Fig. 10. Architecture overview of FaaSBatch.

to enhance the performance of serverless computing in high-concurrency scenarios. In this regard, we propose *FaaSBatch* that aims to optimize computing resource utilization and invocation latency. As shown in Fig. 10, FaaSBatch consists of three modules: *invocation mapper*, *inline-parallel producer* and *resource multiplexer*. Concurrent invocations issued by customers are first classified by the invocation mapper according to their function type. For example, in Fig. 10, λ_{A1} and λ_{A2} are grouped under the common batch window as they share the identical function λ_A . Next, a group of invocations is placed into a single container. Given the mappings between containers and function invocations, the inline-parallel producer expands the batched invocations of containers and executes them in parallel. Finally, the resource multiplexer is responsible for monitoring the execution course of the grouped functions, during which the resource multiplexer manages the reusable redundant resources using caching strategy.

B. Invocation Mapper

FaaSBatch endows developers with the capability to specify an annotation indicating whether a function is CPU-intensive or I/O-intensive. Upon receiving an invocation with this specific annotation, the invocation mapper transitions to a distinct mapping model, tailored to address the varying characteristics of these invocations. This annotation-based approach is not only highly flexible but also well-suited for application in both academic research and industrial contexts [6], [22].

1) *Mapping the CPU-Intensive Functions:* The overarching goal of the invocation mapper is to provision a fewer number of containers for saving resource consumption. As we described in Sec. II, the performance of stuffing concurrent invocations into a single container is similar to that of traditional approaches. Inspired by this observation, we batch all group-level function invocations into a single container for harvesting extreme resource savings. In doing so, these grouped invocations can be executed in parallel.

A function group is defined as the concurrent invocations received for an identical function over a period of time. To obtain function groups, we set a fixed time interval (default in 0.2 second) at the beginning for FaaSBatch to listen to the request queue. Thus, all invocation requests within this time interval can be approximated as concurrent invocations. It is

noteworthy that the interval is configurable, and we investigate the impacts of time interval value on FaaSBatch's performance in Sec. VII-A. For example, during a single time interval, assuming the platform in Fig. 10 receives two function λ_A (consisting λ_{A1} and λ_{A2}) and three function λ_B invocations (involving λ_{B1} to λ_{B3}). Our invocation mapper first divides these invocations into two function groups (see the purple and yellow rectangles in Fig. 10). Then, each group of invocations is batched together into a single container. Last, the two groups of invocations are executed in parallel to shorten invocation latency.

2) *Mapping the I/O Functions:* When handling I/O functions, serverless computing platforms will inevitably receive many requests for creating different types of socket clients, even the requests are invoked by the same user, as socket clients can access external storage objects located in various places by accepting different parameters. As we mentioned before, FaaSBatch utilizes the resource multiplexer module (embedded inside the container, detailed in Sec. III.D.1) to cache and reuse the socket client instances, aiming to avoid the redundant resource creation. Furthermore, the motivation of our study (see Sec. II-C) drives the necessity of employing a cache scheduling algorithm within containers to avoid the prolonged occupation of container memory by cold request.

However, the method of FaaSBatch to cache socket clients within containers, aimed at preventing the unnecessary recreation of these clients, is significantly influenced by its invocation routing strategy. While incorporating a consistent hashing algorithm would intuitively enhance cache locality by directing function invocations to specific containers, the dynamic nature of serverless computing—represented by frequent container starts and stops—challenges this approach. Traditional consistent hashing often results in an imbalance in the distribution of cached elements within containers [23], [24], as represented by most of these elements being hosted in a limited number of containers. This imbalance leads to frequent replacement of cache elements, particularly when a new invocation is dispatched to a container approaching its cache capacity, thereby triggering unnecessary cache replacements. In a high dynamic change in containers, such frequent cache replacements detrimentally impact the efficiency of managing redundant socket clients.

Routing function invocations. To address the above-mentioned issue, we introduce a globally content-aware invocation routing strategy in the invocation mapper. This strategy evaluates the cache status of each container post-invocation and optimizes routing for upcoming invocations.

In the initial state, the invocation mapper targets I/O functions and utilizes a consistent hashing algorithm to map the invocations to containers. For each completed function invocation, the container proactively syncs cache information to the invocation mapper module, enabling FaaSBatch's cache content awareness. For cache contents, the invocation mapper maintains two types of mappings: from cached key to container, and from container to cached keys. The first mapping is primarily for quickly allocating incoming invocations to containers with the relevant cache contents, while the second is for analyzing the cache status in each container. With the support of these two

mapping relationships, when the invocation mapper receives a group of concurrent invocations, it can analyze the parameters ('args' shown on Fig. 12) of these invocations and allocate containers based on the parameters, thereby enhancing the cache hit rate. The advantage of this content-aware approach is that it avoids unnecessary cache replacements caused by the mappings maintained by traditional consistent hashing in highly dynamic container scenarios.

For some request parameters not present in the container's cache, allocating a new container is necessary. As previously discussed, blindly using a conventional consistent hashing to assign invocations to containers can lead to frequent cache replacement in highly dynamic environments. This is because, after the dynamic changes of containers, consistent hashing adjusts the mapping relationships of the cache, which may lead to an imbalance in the number of cached elements in certain containers. Therefore, we maintain an additional layer of mapping on top of the hash mapping, which we call substitute nodes. In other word, for a specific parameter, we maintain not only the nodes derived from the hash calculation for the parameter but also substitute nodes. These substitute nodes may be the hash nodes corresponding to other parameters. Specifically, we select a container with the largest cache space from all containers to serve as the substitute node. Then, subsequent requests targeting that invocation parameter are redirected to this substitute node. After redirecting the requests, the invocation mapper still maintains the existing structure of the mapping relationships, and adds alternative mapping nodes for the requests, avoiding the serious overhead caused by frequent modifications to the hash ring. By doing so, only when the cache space of all containers reaches its maximum limit, FaaSBatch inevitably encounter cache replacement, rather than experiencing frequent cache replacements due to imbalanced distribution of cache elements, as is common with traditional consistent hashing. In this way, we achieve the goal of distributing cache items among dynamically adjusted container instances through a simple and effective method, thereby reducing cache replacements.

C. Inline-Parallel Producer

The inline-parallel producer is primarily in charge of carrying out invocation execution for each function group. More specifically, in collaboration with the invocation mapper, the inline-parallel producer fulfills its responsibility through the following three main steps (see Fig. 11):

Step 1: Analyzing function groups' information. The invocation mapper delivers to the inline-parallel producer a function group along with its information, such as the number of invocations, the function type, and resource limits (see ① in Fig. 11).

Step 2: Obtaining a container based on the received information. Next, the inline-parallel producer selects and obtains a container instance based on the received function type (see ② in Fig. 11), and the occurrence of a cold start depends on whether a keep-alive container exists on the current serverless platform. The resource limit information is mainly used to restrict the number of CPU cores available to the container

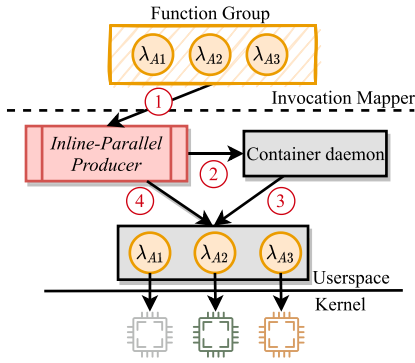


Fig. 11. Module overview of inline-parallel producer.

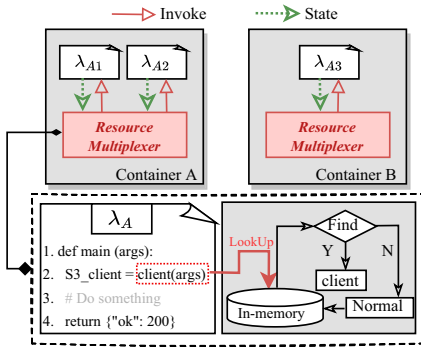


Fig. 12. Module overview of resource multiplexer.

(see ③ in Fig. 11) since a customer may specify the number of CPUs in the function execution environment. The available CPU cores for a docker container can be specified by setting the `cpu_count` or `cpuset_cpus` parameters. After a container instance is ready for serving the group of invocations, the inline-parallel producer delivers the group of invocations into the container.

Step 3: Executing the invocations. In this step, the inline-parallel producer transmits an HTTP request to the container selected in Step 2 to activate the execution of the invocations in the function group (see ④ in Fig. 11).

D. Resource Multiplexer

1) Resource Multiplexing Module: FaaSBatch utilizes the resource multiplexer module to avoid creating redundant resources during the execution of serverless invocations, thereby reducing invocation latency and resource overhead. In each container, we integrate a resource multiplexer to handle function invocation execution (see Fig. 12). In particular, when the inline-parallel producer expands function invocations in a container, the container executes the function invocations under the monitoring of its resource multiplexer. More specifically, the resource multiplexer maintains *resource-args-result* mappings in memory. When a socket client creation request is issued (for example, calling a `client(args)` function), the resource multiplexer intercepts the request and searches the cached mappings for a matched entry. If a matched entry is found, the resource multiplexer directly returns the corresponding result

to serve the creation request. Otherwise, the creation request will be served through the normal procedure. It is noteworthy that we employ a hashing technique to creation arguments to reduce memory overhead and speed up the matching process. In addition, since computing resources are reused at the container level, there is no need to consider hash collisions that occur with extremely low probability [25].

Now let us make use of the following example to illustrate the caching process orchestrated by our proposed resource multiplexer. Let `client(args)` denote an AWS S3 client creation, where `args` represents the creating arguments such as AWS access key, bucket name, etc. For the first request for creating a client, we associate the invoked function with its input arguments as a mapping entry `client → Hash(args)`. Next, the resource multiplexer stores this mapping entry and starts to build the client. Once the client building is completed, the resource multiplexer first sends the client instance (`S3_client` in Fig. 12) to the function caller. Then, function `client` and hashed arguments `Hash(args)` are associated with the client instance (`S3_client`), which is stored as `client → Hash(args) → S3_client` in the resource multiplexer. In doing so, if future requests querying the same creation are issued, the resource multiplexer will directly serve those requests with the cached results. For example, if a serverless platform receives λ_{A1} and λ_{A2} in the same time interval and λ_{A3} in another time interval (see Fig. 12). With the resource multiplexer in place, both container A and container B can maintain only one S3 client instance. Correspondingly, container A saves half of its resource creation overhead, and this saving increases with the increase of concurrency. Besides, the invocation latency is also reduced due to the saving on creating redundancy resources.

2) Cache Scheduling Mechanism: Under the semantic specificity of serverless computing for multiple users (described in Sec. II-C), FaaSBatch is capable of employing a cache scheduling strategy with a single container, thereby enhancing the efficiency of resource utilization.

Recall that the invocation mapper module batches invocations belonging to the identical function and executes them within a single container. With the diversity and high-density of access patterns, function invocations to the same function utilizes different socket client instances, inevitably, to accommodate the diverse users of the front-end. Despite the presence of the resource multiplexer module, various socket clients remain persistently cached in memory until the container is subsequently reclaimed by the serverless computing platform. This can lead to the accumulation of cold data, resulting in inefficient utilization of container memory.

To address the issue of resource waste in a general way, we leverage LRU as the eviction strategy in the resource multiplexer module to manage the cache of the instances. Note that the eviction strategy is implemented in the resource multiplexer for each container (as depicted on Fig. 12). Thus, with the support of the cache eviction strategy, the resource multiplexer module will selectively schedule cache instances to maximize the use of the temporal locality of function request (as demonstrated in Fig. 5). The reasons of selecting LRU as the eviction strategy are discussed in Sec. V-C.

IV. EVALUATION ENVIRONMENT AND METHODOLOGY

Setups and Baselines. This study focuses on the performance of FaaSBatch running on a single machine, rather than the efficiency of clustered servers. In this regard, we develop and evaluate FaaSBatch on our private cloud environment, which is comprised of two Virtual Machines (VMs): a small client VM with 8 vCPUs and 16 GB memory and a large worker VM with 32 vCPUs and 64 GB memory. In addition, we set the keep-alive time for containers in 5 minutes. We compare FaaSBatch with the following three state-of-the-art approaches.

Vanilla: The vanilla approach represents the invocation model adopted by the vast majority of serverless computing frameworks: launching an isolated environment (i.e., a container) for executing each function invocation.

Kraken [15]: Around the principle of batching, Kraken utilizes the notion of slack to allow invocations to be complete in advance of the provided SLOs (Service Level Objectives) while minimizing the number of provisioned containers. Specifically, Kraken provisions a specific number of containers based on the EWMA (Exponentially Weighted Moving Average) model, and then batches the invocations into the launched containers with parameters determined by the supplied SLOs and historical execution times.

SFS [26]: As an OS scheduling in a serverless computing platform, SFS delivers all function invocations to channels, each of which is bound to a specific CPU core. By dynamically perceiving IaT of requests and assigning an adaptive size of time slices to execution tasks, SFS constructs a priority-based strategy to reduce invocation latency in high-concurrency scenarios. In short, SFS transactions improve the performance of short functions at the expense of increasing the execution time of long functions.

Porting Kraken Strategy. In this study, there are some experimental modifications that differ from the original Kraken settings. Specifically, In the original Kraken's experimental setup, the SLOs for each function is fixed at 1000 ms, we argue that it is not well-suited for general scenarios because invocations are executed in widely varying time ranges. Therefore, for fair comparison purposes, we take the 98-percentile latency of each function obtained by the Vanilla strategy as the function SLO for the Kraken strategy. Moreover, since we are primarily interested in the batch size setting in the Kraken policy rather than the efficiency of the EWMA model, we set the accuracy of the predicted workload in Kraken to 100% by using the function invocation patterns collected in the Vanilla strategy.

Dispatch Intervals. The efficiency of the batching approach in FaaSBatch depends on the size of the batch window (i.e., the dispatching interval in our FaaSBatch prototype). To this end, we evaluate the batching efficiency by varying the window sizes from 0.01 s to 0.5 s. A window size of 0.01 s signifies that all function invocations received within this time interval are considered concurrent and uniformly scheduled by the serverless computing framework.

Evaluation Metrics. FaaSBatch aims to optimize *Invocation Latency* and *Resource Cost*. In this study, we define invocation latency as the processing time of each function invocation. That

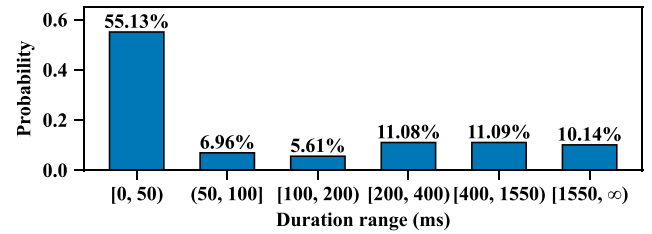


Fig. 13. Probability distribution of function duration.

is, the time from when an invocation is received by a serverless platform to when the final output is returned. The invocation latency is composed of several parts:

- (1) *Scheduling latency* represents the time when a serverless platform receives an invocation until it is sent to a selected container.
- (2) *Cold-start latency* defined as the time overhead to start the selected container, which is originally included in the scheduling latency. However, we gouge it apart from the scheduling latency for a better evaluation (please see details below).
- (3) *Queuing latency* refers to the duration that an invocation is queued in the selected container for execution.
- (4) *Execution latency* is defined as the time period starting from when the container begins the execution of the invocation to the moment the computed result is returned.

Please note that the original scheduling latency involves the cold-start procedure. However, for better evaluation, we remove the cold-start latency from the scheduling latency to evaluate the time overhead of different function scheduling strategies. Ultimately, holding all other latencies intact, we subtract the cold-start latency from the scheduling latency in our evaluation.

Resource costs, on the other hand, represent the utilization costs a serverless computing provider incurs to execute functions. Typical resource costs include the number of provisioned containers, memory consumption, and CPU utilization.

Benchmarks. To evaluate the performance, we generate serverless function workloads based on the Azure Functions [11] and Azure Blob [21] traces. The Azure trace contains 1,980,951 function execution times and invocation timestamps spanning 14 days. Motivated by the experimental settings in previous study [26], we analyze the execution time of function invocations. Fig. 13 plots the distribution of function execution times. It shows that the execution time durations roughly follow a skewed distribution, which is consistent with the one presented in the previous study [26]. Thus, we exploit this distribution to generate workloads for serverless functions by using CPU-intensive computational functions - computing the Fibonacci series (fib) with different input N-values. Due to the fact that fib with N between 20 and 26 completes in less than 45 ms, we program our benchmark to generate fib functions with an N between 20 and 26 with a probability of 55.13%. A more detailed mapping relationship between N-values and execution durations can be found in Table I in study [26]. Additionally, in conjunction with the Azure trace, we follow the process in Listing 1 to create AWS S3 clients (I/O functions) and

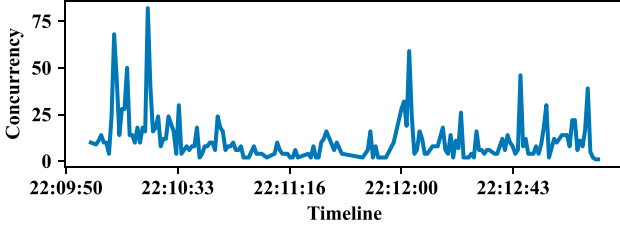


Fig. 14. Invocation pattern of the generated workload.

evaluate the resource multiplexer module in FaaSBatch. We replayed the total of 2,000 invocations made within 3 minute (from 22:10 to 22:13) of the Azure Day 13 trace. And the invocation patterns are shown in Fig. 14. We choose this representative function invocation pattern because it serves as a strong indicator of the burstiness of serverless functions. The Azure Function trace lacks detailed parameters for each invocation. This limitation poses challenges in generating parameters for I/O functions based solely on the Azure Function trace. In contrast, the Azure Blob trace, comprising 14 days' worth of logs encompassing 33.1 million invocations and 44.3 million data accesses, offers a broader spectrum of invocation parameters, including I/O identifiers, thereby enriching the data set. Consequently, our experimental methodology integrates both Azure Function and Azure Blob traces to yield production-level data, encompassing I/O identifiers. In the Azure Blob trace, these I/O identifiers are primarily represented by I/O parameters, specifically the `arg` as illustrated in Fig. 12.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the performance and effectiveness of FaaSBatch driven by Azure traces [11], [21]. We compare our FaaSBatch with three state-of-the-art schedulers, namely, Vanilla, SFS, and Kraken.

A. Evaluating Invocation Latency

We evaluate the performance of FaaSBatch in terms of the invocation latency during function execution. Recall from Sec. IV that the invocation latency in this study is composited of four parts. In Fig. 15, Fig. 16, and Fig. 17, we plot the invocation latency for CPU-intensive functions and I/O functions, respectively, under various strategies across different dispatch intervals. In the following, we demonstrate the experimental results and detailed analysis.

1) *Scheduling Latency*: A scheduling latency represents the duration from when a serverless platform receives a function invocation to when the invocation is sent to the container. When gauging scheduling latency, we take into account the cold-start latency introduced below. However, when it comes to the whole scheduling process, we subtract scheduling latency from cold-start latency for better evaluation.

CPU-intensive Functions: Fig. 15(a) depicts the scheduling latency incurred when executing CPU-intensive functions, comparing four distinct strategies across a spectrum of dispatch intervals. It shows that our FaaSBatch takes the shortest time

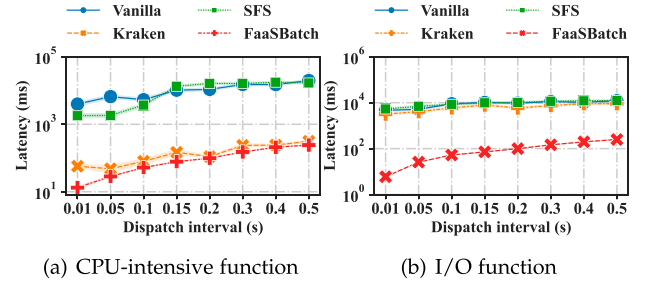


Fig. 15. Scheduling latency.

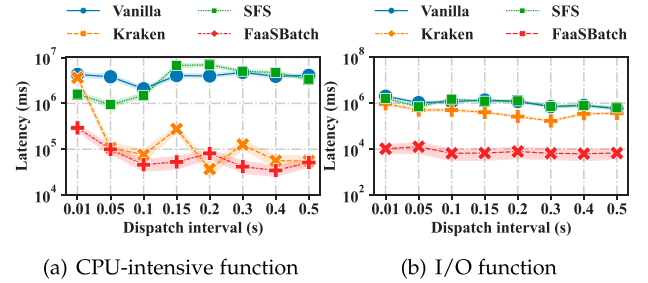


Fig. 16. Cold-start latency.

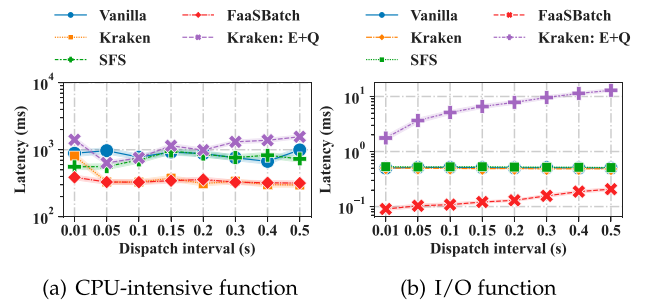


Fig. 17. Execution and queuing latency.

to dispatch function invocations to containers, which demonstrates the superiority of FaaSBatch over the Vanilla and SFS in terms of scheduling latency. On the other hand, the better-performing policy Kraken is comparable to FaaSBatch in terms of decision time. This is because, in our experimental settings, we set the prediction accuracy of Kraken to 100% (detailed in Sec. IV). Practically, Kraken would be burdened with additional time requirements for EWMA based forecasting. Overall, this enhanced performance is attributed to FaaSBatch's efficient approach of only launching a single container for each group of concurrent invocations, thus bypassing the complex process of determining the number of containers required.

I/O Functions: When it comes to the I/O functions, we can observe from Fig. 15(b) that FaaSBatch continues to offer a considerable advantage in decision-making time over Kraken, SFS, and Vanilla. For example, FaaSBatch achieves the reduction on scheduling latency over the other three strategy ranging from 57.2% to 82.98%. Digging a bit deeper into the scheduling latency, about 40% of the decision times in SFS and Vanilla

are less than one second, while more than half of their decision times are more than ten seconds. As for Kraken, nearly 90% of the decision times are less than one second. In the case of the best performer FaaSBatch, it delivers sub-second decisions for all invocations.

In short, SFS and Vanilla fail to distribute invocations to containers efficiently in high-concurrency scenarios. This is because they launch a single container for each incoming function invocation, which is a CPU-consuming task. Therefore, busy CPUs running in worker nodes can lead to amplified instruction execution times. Thanks to the ability of batching function invocations, Kraken is comparable to FaaSBatch when executing CPU-intensive functions. However, the queuing overhead incurred by Kraken impacts the extended invocation latency (detailed in Sec. V.A.3).

2) *Cold-Start Latency*: Due to the varying number of containers generated by different strategies during function execution, we have summed up the total container cold-start times under different dispatch intervals and presented them in Fig. 16(a) and Fig. 16(b).

CPU-intensive Functions: Fig. 16(a) shows the aggregated cold-start latency when executing CPU-intensive functions. This illustrates that among the four assessed strategies, FaaSBatch consistently exhibits the least cold start overhead across various dispatch intervals. This is because the invocation mapper module places concurrent invocations into a single container, thereby reducing the number of containers needed. Besides, the cold-start overhead produced in Kraken is quite similar to our FaaSBatch. In other words, Kraken provisions a similar number of containers as FaaSBatch. This effect is expected because Kraken batches invocations based on the SLO metrics of CPU-intensive functions. However, a similar amount of provisioned containers forces Kraken to impose a considerable queuing latency within a single container. The queuing overhead is described in the next evaluation section.

I/O Functions: Fig. 16(b) plots the distribution of the cold-start latency when executing I/O functions. It reveals that among the four evaluated strategies, FaaSBatch still maintains the lowest cold start overhead.

3) *Execution and Queuing Latency*: There are two types of latency measurements. The first one is execution latency, which is defined as the time period starting from when the container begins the execution of the invocation to the moment the computed result is returned. The second one is the sum of execution latency and queuing latency. Please note that only Kraken policy has queuing latency when executing concurrent invocations, which is labeled as *Kraken: E+Q* (see the purple line in Fig. 17(a) and Fig. 17(b)).

CPU-intensive Functions: Fig. 17(a) shows the execution latency of the four strategies when executing CPU-intensive functions with different dispatch intervals. It shows that our FaaSBatch outperforms significantly compared to Vanilla, SFS, and Kraken (with the queuing overhead). This is because FaaSBatch is capable of expanding the execution of concurrent invocations as multiple threads in the same container without incurring additional performance overhead. As for Kraken, its execution time is similar to that of our FaaSBatch. Unfortunately,

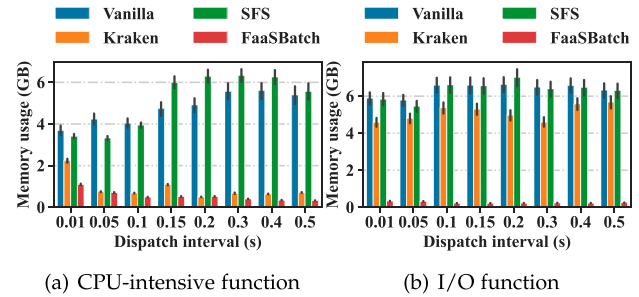


Fig. 18. Total memory consumption.

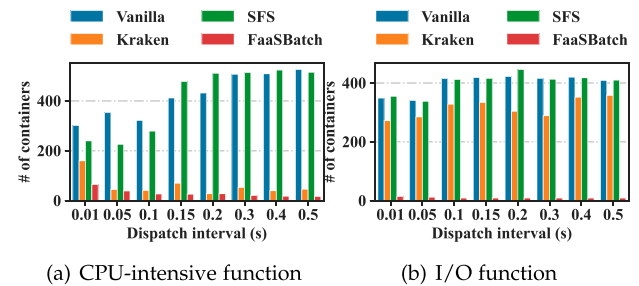


Fig. 19. Number of provisioned containers.

its processing time (marked with *Kraken: E+Q* in Fig. 17(a)) is much higher since Kraken generates huge queued invocations. On average, FaaSBatch achieves a $2.52\times$, $1.14\times$, and $2.19\times$ improvement in processing time over Vanilla, Kraken, and SFS, respectively.

I/O Functions: Fig. 17(b) depicts the execution latency of the four strategies when executing I/O functions. Fig. 17(b) demonstrates that our FaaSBatch significantly cuts back invocation latency. On average, FaaSBatch demonstrates a $3.67\times$, $3.79\times$, and $3.56\times$ enhancement execution latency over Vanilla, Kraken (with the queuing overhead) and SFS, respectively. The peak improvements vary between $204.57\times$ and $650.38\times$. The primary reason is that Vanilla, SFS, and Kraken generate redundant resources (i.e., repeated creation of AWS S3 clients) when executing the function invocations. These results illustrate that FaaSBatch enables redundant resource multiplexing at the granularity of a single container.

B. Evaluating the Resource Cost

In this group of experiments, we investigate the resource cost of the four strategies. We obtain the resource utilization (i.e., CPU and memory) in the host at a frequency of once per second. Fig. 18, Fig. 19 and Fig. 20 show the resource overheads of CPU-intensive and I/O functions under different experimental settings. Specifically, Fig. 18(a), Fig. 18(b) show the memory usage of the four strategies, Fig. 19(a) and Fig. 19(b) illustrate the number of container provisioned in different experiment settings, and Fig. 20(a) and Fig. 20(b) demonstrate the CPU overhead of CPU-intensive and I/O functions, respectively.

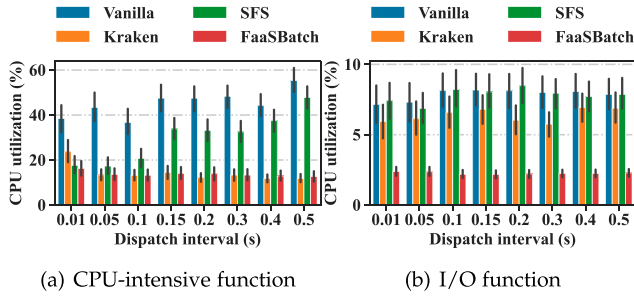


Fig. 20. CPU resource overhead.

1) *Memory Resource Overhead*: In this part of the study, we evaluate the memory resource overhead caused by CPU-intensive and I/O functions.

CPU-intensive Functions: Again, Fig. 18(a) shows the total system memory usage caused by the four compared strategies with respect to different dispatch intervals when executing CPU-intensive Functions. It reveals that FaaSBatch outperforms Vanilla and SFS significantly. For instance, FaaSBatch reduces the total system memory usage of Vanilla and SFS by 70.20% to 93.97% and 67.74% to 94.50%, respectively. On the other hand, FaaSBatch has also made progress in memory resource optimization compared to Kraken, achieving an average optimization of 34.10%, and up to 54.50% at its maximum. However, when dealing with I/O functions (see the section below), the optimization efficiency of Kraken decreases compared to our FaaSBatch. The reason is that the efficiency of Kraken's batch decisions varies with function invocation patterns, which is highly dynamic in serverless computing. In contrast, our FaaSBatch achieves a more stable performance improvement.

I/O Functions: Fig. 18(b) shows the total system memory consumption caused by the four compared strategies with respect to different dispatch intervals when executing I/O functions. We can observe from Fig. 18(b) that FaaSBatch still achieves the lowest memory consumption. And FaaSBatch exhibits higher system memory optimization when processing I/O functions than when executing CPU-intensive functions. For example, FaaSBatch reduces the memory consumption of Vanilla, Kraken, and SFS by 94.25% to 96.61%, 92.62% to 95.92%, and 94.08% to 96.79%, respectively.

The root cause is that FaaSBatch leverages its resource multiplexer module to eliminate resource redundancy. Moreover, Fig. 18(a) and Fig. 18(b) indicate that the optimization efficiency of Kraken decreases compare to our FaaSBatch. That is, FaaSBatch provides more stable optimization efficiency. This effect is realized because our FaaSBatch (i) batches all invocations received over a period of time and (ii) multiplexes the redundant resources generated in the procedure of function invocation execution.

2) *Number of Provisioned Containers*: The approach of mapping invocations into a single container slashes the number of provisioned containers during the execution.

CPU-intensive Functions: Fig. 19(a) depicts the number of containers provisioned caused by the policies with respect to

different dispatch intervals under the CPU-intensive function workload. It can be observed from Fig. 19(a) that FaaSBatch is superior to Vanilla, Kraken, and SFS in terms of spawned containers. For example, Vanilla, Kraken, and SFS averagely spawn more containers than FaaSBatch by 92.83%, 50.10%, and 92.66%, respectively. Some reasons can explain this effect. First, Vanilla and SFS spawn one container per invocation request. Second, even though Kraken strives to reduce the total number of containers spawned by batching invocation requests, Kraken fails to recognize the effectiveness of concurrently executing function invocations within a single container.

I/O Functions: Fig. 19(b) shows the number of containers provisioned caused by the policies with respect to different dispatch intervals under the I/O function workload. Again, FaaSBatch surpasses the other three policies. For example, FaaSBatch reduces the number of containers required by Vanilla and SFS by 97.49% and 97.50%, respectively. Please note that on average, 398.87 and 401 containers are provisioned to serve a total of 2,000 function invocations, and containers and invocations are mapped on a one-to-one basis. Kraken, a batch policy designated to optimize the number of provisioned containers, launches an average of 315.2 containers for 2,000 I/O function invocations, while FaaSBatch requires only 10 to achieve the same computing purpose. In other words, FaaSBatch optimizes the performance of Kraken by 96.82%. The following aspects can explain the superior performance delivered by FaaSBatch. To satisfies the executed SLOs, Kraken must start multiple containers for a set of concurrent invocations to reduce queuing overhead. FaaSBatch on average serves 20 (2,000/10) invocations with one container, while Vanilla, SFS, and Kraken must activate a container when receiving 5.01 (2,000/398.87), 4.98 (2,000/401) and 6.34 (2,000/315.25) invocations, respectively. In a word, serving the equivalent number of function invocations with fewer containers is key in FaaSBatch to reduce memory consumption. In addition to FaaSBatch's lowest number of containers provisioned, FaaSBatch also achieves more stable performance than Kraken's performance fluctuation. Specifically, the average number of containers generated by Kraken rises from 60.37 when processing CPU-intensive functions to 315.25 when processing I/O functions. Such a rising number indicates that Kraken's batch decision efficiency varies with the function invocation patterns, which are highly dynamic in serverless computing.

3) *CPU Resource Overhead*: Now we investigate the CPU resource overhead of the four policies. Fig. 20(a) and Fig. 20(b) depict the CPU utilization when executing the CPU and I/O functions, respectively. The main contributors to CPU resource overhead are the running serverless platform, containers, and the execution of functions submitted by consumers. When applying the SFS policy, its additional process scheduler also consumes CPU resources.

CPU-intensive Functions: Fig. 20(a) depicts the CPU utilization of the four policies with respect to different dispatch intervals under the CPU-intensive function workload. Unsurprisingly, FaaSBatch outpaces Vanilla, SFS, and Kraken. For example, FaaSBatch reduces the CPU utilization of Vanilla,

SFS, and Kraken by 47.04%, 45.55%, and 20.84%, respectively. There are two reasons for FaaSBatch's lowest CPU utilization: (1) Vanilla, SFS, and Kraken spend more time on scheduling function invocations, which pushes CPUs toward higher utilization (please see Fig. 15(a)); (2) Vanilla, SFS, and Kraken incur more cold starts, and starting containers take up more CPU resources (please see Fig. 16(a)).

I/O Functions: Fig. 20(b) shows the CPU utilization of the four policies with respect to different dispatch intervals under the I/O function workload. Similar to the CPU-intensive function workload case above, we can see that the CPU resources consumed by FaaSBatch are significantly lower than Vanilla, SFS, and Kraken. For instance, FaaSBatch reduces the CPU utilization of Vanilla, SFS, and Kraken by 66.69% to 73.36%, 65.54% to 73.89%, and 59.92% to 68.01%, respectively. Moreover, FaaSBatch saves a greater percentage of CPU resources when performing I/O functions than when performing CPU functions. This is because, in addition to reducing scheduling overhead and cold-start overhead, FaaSBatch's resource multiplexer module avoids resource overhead for repeated creation. In contrast, Vanilla, SFS, and Kraken must create more socket clients, which leads to high CPU utilization.

C. Evaluating the Cache Efficiency

This section discusses the resource multiplexer module's benefits for serverless computing, focusing on the need for an effective cache scheduling policy due to the drawbacks of unlimited caching, such as increased memory use from idle data.

We conducted separate experiments to assess cache efficiency, utilizing a synthetic dataset derived from Azure Function [11] and Azure blob [21] traces, featuring 10,000 function invocations and 5,060 unique blob requests. Traditional cache policies like LRU, LFU, Random, IdleCache [9], and GDSF [27] were compared with the default UnboundedCache policy, examining cache hit rates and memory usage across varied cache sizes (2 to 64). Fig. 21 illustrates the performance of these caching strategies. Notably, UnboundedCache's performance remained constant regardless of cache size, influenced instead by the variability of function invocations and container assignments. This variability affected the cache hit rates and memory consumption across different containers.

In conclusion, UnboundedCache, GDSF and IdleCache fail to yield a significant enhancement in cache hit rates, despite their considerable consumption of memory resources. In contrast, LRU caching strategy can provide a relatively high cache hit rate in most cases while maintaining a lower memory consumption. Specifically, LRU and LFU have similar memory consumption, but LRU improves the cache hit rate by 10% to 50% compared to LFU. Based on the above experimental data, we believe that LRU can provide a more versatile scheduling strategy for the resource multiplexer module.

D. Evaluating the Routing Strategy

The invocation mapper utilizes the content-aware invocation routing strategy to improve the efficiency of distributed

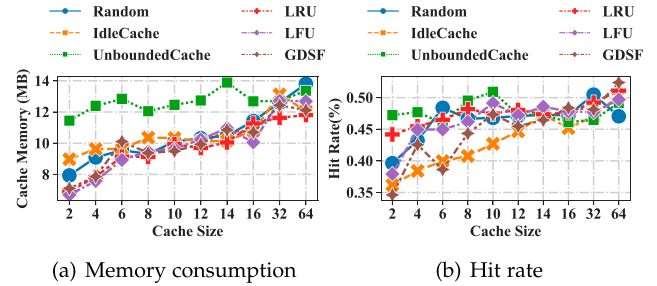


Fig. 21. Measurements of different caching strategies.

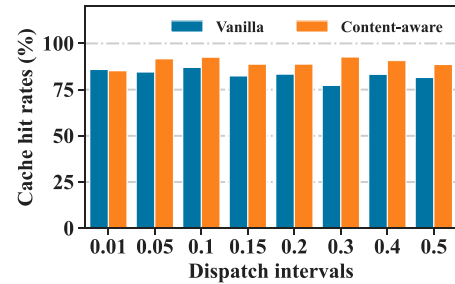


Fig. 22. Cache hit rates of different routing strategies.

caching in a dynamically changing container environment. As previously outlined in Section III.B.2, dynamic container provisioning in serverless architectures can give rise to cache imbalances. This imbalance disproportionately distributes invocations to certain containers, thereby elevating the frequency of cache replacements in these containers. Frequent cache replacements lead to a reduced cache hit rate, which in turn triggers the creation of redundant resources for I/O functions, adversely impacting their execution efficiency. Achieving a satisfactory cache hit rate is an essential metric for evaluating the proficiency of invocation routing algorithms in the realm of distributed caching. In the following section, we conduct a thorough evaluation of our specialized design approach for I/O functions, focusing specifically on the metric of cache hit rates.

Fig. 22 illustrates the cache hit rates of different request routing strategies, i.e., Vanilla and Content-aware. Please note that we leverage LRU as the scheduling strategy inside container. The Vanilla strategy signifies that the invocation mapper allocates I/O functions to containers based on consistent hashing alone. In contrast, the content-aware represents the incorporation of each container's cache information on top of consistent hashing. Experimental results demonstrate that the content-aware request routing strategy improves the cache hit rate in distributed caching. On average, it decreased by 8.21% compared to the Vanilla strategy, with a maximum reduction of 19.89%. This indicates that the content-aware routing strategy can effectively integrate in-container cache information to allocate requests from a global perspective.

VI. RELATED WORK

Realizing that starting a single container for each incoming request leads to huge resource waste in traditional serverless

computing platforms, many works have been proposed to optimize service overhead through batch processing. For instance, GrandSLAM [14] proposed a dynamic slack-aware batching policy to minimize SLA violation in a microservice architecture. On the basis of the notion of slack, many works utilize a batching strategy to execute latency-critical tasks. Some researchers leverage history-based workload forecasting strategies along with adaptive batching support to reduce service costs while ensuring SLOs Mark [13], BATCH [28], Fifer [12], Kraken [15] and Cypress [16]. In addition, Zu et al. strove to improve the efficiency of serverless computing by limiting container execution to a single CPU core and thus avoiding CPU preemption [29]. Similarly, SFS schedules each function invocation to a specific CPU core, aiming at optimizing the performance of short functions at the cost of increasing the execution time of long functions [26].

Existing batch solutions mainly focus on workflow prediction techniques to predict functional workloads to alleviate resource costs. Unlike the aforementioned approaches, our FaaSBatch reduces the overhead of function invocation based on the insightful observation that provisioning one single container to carry out all concurrent invocations provides equivalent performance. In particular, the following three salient features distinguish our FaaSBatch from existing schemes: (i) Mapping concurrent invocations to batches in a single container, reducing resource the creation of additional containers and lowering resource utilization. (ii) Executing these batched invocations in parallel, cutting down latency. (iii) Multiplexing and scheduling redundant resources inside containers to further decrease latency and memory usage.

VII. DISCUSSION

Now we are in the position to discuss the effectiveness of our FaaSBatch, including (1) examining the impact of increased dispatch intervals on performance, and (2) analyzing FaaSBatch's behavior under high-concurrency invocations issued from various functions in different dispatch interval.

A. Efficiency Varying the Dispatch Intervals

In this part of discussion, we analyze the impact of varying dispatch intervals on the invocation latency of FaaSBatch.

1) *Invocation Latency*: As depicted in Fig. 15, it is noteworthy that although the scheduling latency of FaaSBatch slightly increase with the growth of dispatch intervals, it consistently outperforms other strategies in reducing scheduling latency. The cold-start latency, on the other hand, as illustrated on Fig. 16, shows a decrease trend with the increase of dispatch interval. This is because a larger dispatch interval results in more concurrent invocations being assigned to the same container for execution, thereby reducing the number of container provisioned. Fig. 17 shows the opposed trend of FaaSBatch's performance when executing CPU-intensive and I/O functions. Specifically, as the dispatch interval increases, the latency of FaaSBatch executing CPU-intensive functions gradually decreases, while the performance in executing I/O function shows a slow upward

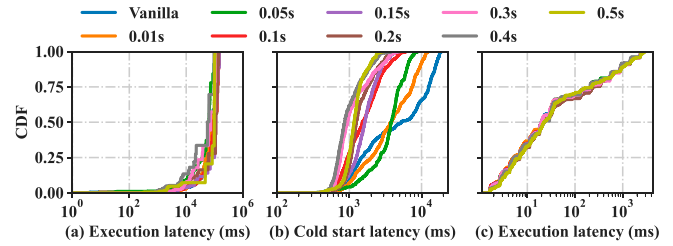


Fig. 23. CDF of latency across different dispatch intervals.

trend. This occurs because as the number of concurrent invocations to be processed increases, the resource multiplexer's assessment of redundant resource modules and the implementation of the cache scheduling strategy incur a minor execution overhead. However, FaaSBatch still maintains efficient processing performance relative to other strategies.

2) *Resource Cost*: In this group of evaluations, we conduct experimental analysis on the results driven by the I/O function workload. This is due to the observation that FaaSBatch's performance, when executing CPU-intensive and I/O functions, demonstrates similar trends. Fig. 18(b) demonstrates a trend that as the dispatch interval increases, the performance improvement of FaaSBatch shows an ascending trend. For instance, the total memory usage of FaaSBatch decreases from 0.95 GB to 0.31 GB as the dispatch interval increases. In contrast, the total memory usage of Vanilla and SFS increases, and Kraken's total memory usage fluctuates around 2.1 GB. The reason for FaaSBatch's increasing improvement is two-fold: (i) a larger interval value helps the invocation mapper module stuff more invocations into a single container, which reduces the number of containers started, thereby cutting down memory overhead; (ii) the resource multiplexer module can provide the shared resources for more function executions, which avoids the creation of redundant resources, thus greatly slashing memory requirements.

B. Performance Under High-Concurrency Invocations

In this section, we assess the performance of FaaSBatch when receiving high-concurrency invocations across various dispatch intervals (from 0.01 seconds to 0.5 seconds), in comparison to the Vanilla strategy.

Fig. 23(a) shows the CDF of scheduling latency for 1000 concurrent invocations belonging to different functions under both Vanilla and FaaSBatch strategies. It can be observed that with the increase of dispatch intervals, there is no significant trend in scheduling latency of FaaSBatch. This indicates that under high concurrent invocations, different dispatch intervals of FaaSBatch do not significantly adversely affect the scheduling latency.

Fig. 23(b) and Fig. 23(c) illustrate the CDFs of cold-start latency and execution latency for invocations in the same experimental settings. Fig. 23(b) discloses a stochastic pattern in the distribution of cold-start latency across different scenarios. This variability is expected, considering that different

dispatch intervals influence the timing of container creation, and these creation tasks are interdependent due to the resource contention. Therefore, it can be observed that the impact of different dispatch intervals on cold-start latency within FaaS-Batch is random, depending on the pressure of container creation task. On the other hand, the results depicted in Fig. 23(c) show that execution latency remains remarkably consistent across various scenarios. This consistency suggests that, within the context of short-duration, high-concurrency invocation patterns, the different dispatch intervals employed in FaaSBatch do not result in adversely effect the execution efficiency of the functions.

VIII. CONCLUSION

In our study, we introduced FaaSBatch, a comprehensive batch system for serverless computing, designed to decrease invocation latency and resource overhead. FaaSBatch, motivated by the efficiency of using a single container for concurrent invocations, integrates an *invocation Mapper* for global request allocation and an *inline-parallel producer* for executing batched invocations in parallel. Additionally, its *resource multiplexer* module efficiently reuses resources, reducing container resource pressure and further cutting invocation latency. Extensive experiments confirm FaaSBatch's efficacy in significantly reducing invocation latency and resource overhead compared to existing solutions.

REFERENCES

- [1] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. 31st USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.
- [2] "AWS Lambda." Amazon. Accessed: Nov. 15, 2022. [Online]. Available: <https://aws.amazon.com/lambda/>
- [3] "Azure function." Azure. Accessed: Nov. 15, 2022. [Online]. Available: <https://azure.microsoft.com/en-us/services/functions/>
- [4] "OpenWhisk." Apache. Accessed: Nov. 15, 2022. [Online]. Available: <https://openwhisk.apache.org/>
- [5] "OpenFaaS." Accessed: Nov. 15, 2022. [Online]. Available: <https://www.openfaas.com/>
- [6] M. Shahrad et al., "Architectural implications of function-as-a-service computing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 1063–1075.
- [7] T. Yu et al., "Characterizing serverless platforms with ServerlessBench," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 30–44.
- [8] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," in *Proc. 32nd USENIX Annu. Tech. Conf.*, 2021, pp. 443–457.
- [9] Z. Li et al., "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. 33rd USENIX Annu. Tech. Conf.*, 2022, pp. 69–84.
- [10] Z. Li et al., "FaaSFlow: Enable efficient workflow execution for function-as-a-service," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2022, pp. 782–796.
- [11] Y. Zhang et al., "Faster and cheaper serverless computing on harvested resources," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ.*, 2021, pp. 724–739.
- [12] J. R. Gunasekaran et al., "Fifer: Tackling resource underutilization in the serverless era," in *Proc. 21st ACM/IFIP Int. Middleware Conf.*, 2020, pp. 280–295.
- [13] C. Zhang et al., "MARk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in *Proc. 30th USENIX Annu. Tech. Conf.*, 2019, pp. 1049–1062.
- [14] R. S. Kannan et al., "GrandSLAM: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. 14th ACM Eur. Conf. Comput. Syst.*, 2019, pp. 1–16.
- [15] V. M. Bhasi et al., "Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms," in *Proc. 12th ACM Symp. Cloud Comput.*, 2021, pp. 153–167.
- [16] V. M. Bhasi et al., "Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms," in *Proc. 13th ACM Symp. Cloud Comput.*, 2022, pp. 257–272.
- [17] "Using AWS Lambda with Amazon S3." Amazon. Accessed: Nov. 17, 2022. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html>
- [18] "Azure blob storage." Azure. Accessed: Nov. 17, 2022. [Online]. Available: <https://azure.microsoft.com/en-us/products/storage/blobs/>
- [19] Z. Wu et al., "FaaSBatch: Enhancing the efficiency of serverless computing by batching and expanding functions," in *Proc. IEEE 43rd Int. Conf. Distrib. Comput. Syst.*, 2023, pp. 1–11.
- [20] Z. Xu et al., "Stateful serverless application placement in MEC with function and state dependencies," *IEEE Trans. Comput.*, vol. 72, no. 9, pp. 2701–2716, Sep. 2023.
- [21] F. Romero et al., "FaaS: A transparent auto-scaling cache for serverless applications," in *Proc. 12th ACM Symp. Cloud Comput.*, 2021, pp. 122–137.
- [22] S. Kotni et al., "Faastlane: Accelerating function-as-a-service workflows," in *Proc. 32nd USENIX Annu. Tech. Conf.*, 2021, pp. 805–820.
- [23] Q. Huang et al., "Characterizing load imbalance in real-world networked caches," in *Proc. 13th ACM Workshop Hot Topics Netw.*, 2014, pp. 1–7.
- [24] Y. Hong et al., "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–17.
- [25] K. Maziarz et al., "Hashing modulo Alpha-equivalence," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2021, pp. 960–973.
- [26] Y. Fu et al., "SFS: Smart OS scheduling for serverless functions," in *Proc. 34th IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 584–599.
- [27] A. Fuerst and P. Sharma, "FaaSCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2021, pp. 386–400.
- [28] A. Ali et al., "BATCH: Machine learning inference serving on serverless platforms with adaptive batching," in *Proc. 32nd IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.
- [29] P. Zuk et al., "Call scheduling to reduce response time of a FaaS system," in *Proc. 24th IEEE Int. Conf. Cluster Comput.*, 2022, pp. 172–182.



Zhaorui Wu received the M.S. degree in computer architecture from the Computer Science Department of Jinan University. He is working toward the Ph.D. degree with Computer Science Department, Jinan University, China. He has published several papers in IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, IEEE DATE, and IEEE ICDCS. His current research interests include serverless computing, computer architecture, and performance evaluation.



Yuhui Deng received the Ph.D. degree from the Huazhong University of Science and Technology, in 2004. He is a Professor with the Computer Science Department, Jinan University. Before joining Jinan University, he worked as a Senior Research Scientist with EMC Corporation, from 2008 to 2009. He worked as a Research Officer with Cranfield University, U.K., from 2005 to 2008. He was on the finalist of storage challenge in ACM/IEEE SC2007, and on the finalist of EMC global innovation showcase in 2008. He was awarded the Annual Best Paper Award from JISE in 2017, and Best Journal Paper Award from the Big Data Technical Committee of IEEE Communications Society in 2019. His research interests include computer architecture and cloud computing.



Yi Zhou received the Ph.D. degree in computer science from Auburn University, in 2018. He is currently an Assistant Professor with TSYS School of Computer Science at Columbus State University. Prior to joining Columbus State University, in 2018, he has been a Software Engineer in Alcatel-Lucent Technologies (China) Co., Ltd., for four years. His research interests include energy-saving techniques, big data techniques, and parallel computing.



Shujie Pang received the M.S. degree from the Computer Science Department of Yantai University. He is currently working toward the Ph.D. degree with the Computer Science Department, Jinan University, China. He has published several papers in ACM TOS and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. His current research interests include storage system, computer architecture, and performance evaluation.



Jie Li received the B.E. degree in automation from Wanjiang University of Technology, Anhui, China, in 2016, and the M.S. degree in computer science and technology from the Guangxi University for Nationalities, Nanning, China, in 2019. He is currently working toward the Ph.D. degree with the Computer Science Department, Jinan University. His current research interests include parallel and distributed computing, data center architecture, cloud computing, and data replica placement.



Xiao Qin received the Ph.D. degree in computer science from the University of Nebraska–Lincoln, Lincoln, Nebraska, in 2004. He is currently an Alumni Professor and the Director of Graduate Programs with the Department of Computer Science and Software Engineering at Auburn University. He won an NSF CAREER Award, in 2009. His research interests include parallel and distributed systems, storage systems, and performance evaluation.