FaaSBatch: Enhancing the Efficiency of Serverless Computing by Batching and Expanding Functions

Zhaorui Wu, Yuhui Deng, Yi Zhou, Jie Li, Shujie Pang

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 92.18%, 89.54%, and 90.65%, respectively; FaaSBatch also slashes the resource overhead of Vanilla, SFS, and Kraken by 58.89% to 94.77%, 43.72% to 90.39%, and 42.99% to 78.88%, 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], [12], 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 (or batch)1 functions by introducing the notion of slack, thereby reducing the number of launched containers [13], [14], [15], [16], [17]. 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 simply batching invocations into a single container causes 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 [18] or Azure Blob storage [19]). And these client instances are created repeatedly when a platform handles multiple function executions. The overhead of these instances created repeatedly is considerably high, especially in cases where function execution times are generally short.

These insightful observations motivate us to devise FaaSBatch - 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 low-

1We use "queue" and "batch" interchangeably in this paper.
est execution latency when spawning containers to service incoming requests for functions, however, this comes at the cost of starting a large number of 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 batches all concurrent invocations to reduce the number of provisioned containers. The invocation mapper places all invocations received within a specific time-window interval into a single container instead of batches them in multiple containers. 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 (please see the detailed design 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. In doing so, there is no need to create any additional redundant resources, which further suppresses resource pressure and invocation delay inside a container. To sum up, we make the following contributions in this study.

- 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 redundant resources.
- We conduct extensive experiments to evaluate the performance of 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). Then, we present the necessity of multiplexing redundant resources during function execution (Sec. II-B).

### 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 the 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 function execution time.

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 performance.

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 [10]. Digging a bit deeper into the execution characteristics, we analyze the Azure Trace [12] and plot the experimental result in Fig. 2. Fig. 2 shows the execution pattern of three 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. 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.

**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.

Code Listing 1: Creating cloud storage clients
```python
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...

# 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...
```

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 [18] or Azure Blob storage [19]. 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 CURD operations). 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 the same client reside in container-owned memory, resulting in wasted memory.

To shed light on the accesses patterns of cloud storage objects, we investigate the Azure Blob trace [20], which contains 14 days of logs, including 33.1 million invocations with 44.3 million data accesses. The analysis results are plotted in Fig. 3, which shows the cumulative distribution function (CDF) of the inter-arrival time (IaT) for the blobs with more than 2 accesses (there is no IaT otherwise). Fig. 3 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. 3 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 [20]. 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. 4 and Fig. 5. Fig. 4 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 (3165 ms). Fig. 5 illustrates a rising trend of memory overhead with respect to the repeated creation of S3 socket clients. Specifically, as the concurrency degree goes up from 1 to 9, the memory consumption of a single container increases from 9 MB to 60 MB.

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.

**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 resource utilization.

### III. DESIGN OF FaaSBatch

#### A. Architecture Overview

The aforementioned inspiring observations and analysis reveal that there exists a pressing demand behind a novel strategy 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. 6, FaaSBatch consists of three modules: **Invoke Mapper**, **Inline-Parallel Producer**, and **Resource Multiplexer**. Concurrent invocations issued by customers are first classified by Invoke Mapper according to their function type. Next, a group of invocations is placed into a single container. Given the mappings between containers and function invocations Inline-Parallel Producer expands the batched invocations of containers and executes them in parallel. Finally, Resource Multiplexer is responsible for monitoring the execution course of the grouped functions, during which Resource Multiplexer figures out reusable redundant resources.

#### B. Invoke Mapper

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. IV. For example, during a single time interval, assuming the platform in Fig. 6 receives two function $\lambda_A$ (consisting $\lambda_{A1}$ and $\lambda_{A2}$) and three function $\lambda_B$ invocations (involving $\lambda_{B1}$ to $\lambda_{B3}$). Our invocation mapper first divides these invocations into two function groups (see the purple and yellow rectangles in Fig. 6). 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.

#### 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. 7):

1. **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. 7).
2. **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. 7), 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 (see 3 in Fig. 7) 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.

3) **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. 7). Similar to all other batch schemes applied to serverless computing scenarios [14], [13], [21], [16], the HTTP request is returned to FaaSBatch only after all invocations of the function group have completed. It is a non-trivial task to return completed invocations early among all the parallel executions, and we leave it for future work.

### D. Resource Multiplexer

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. 8). 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 function 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 [22].

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. 8) 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. 8). 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 redundant resources. In a nutshell, FaaSBatch cuts down execution overheads in terms of invocation latency and resource utilization by preventing the repeated creation of functions that request instances of the same resource.

### 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. To evaluate the performance of FaaSBatch, we compare FaaSBatch with the following three state-of-the-art approaches.

1) **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.

2) **Kraken:** [16] 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 first 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.
3) SFS: [23] 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.

Comparison. FaaSBatch is distinctly different from the above methods in the following aspects. First, it leverages an inline-parallel technique to eliminate waiting time during batch processing. Second, it further multiplexes redundant resources to reduce memory overhead and expedite the execution of function invocations. Furthermore, it batches all concurrent invocations into a single container, which extremely reduces the number of provisioned containers.

Porting Kraken and SFS Strategies. 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.

When it comes to the SFS scheme, it provides an easy-to-port version that only needs to transfer the PID (Process Identification) of a function invocation with a globally unique ID, and its built-in scheduler can perform CPU scheduling in user space.

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 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 the amount of time CPUs take to execute an invocation.

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 inures to execute functions. Typical resource costs include the number of provisioned containers, memory consumption, and CPU utilization.

<table>
<thead>
<tr>
<th>Duration range (ms)</th>
<th>0.0</th>
<th>0.2</th>
<th>0.4</th>
<th>0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0[0, 50)</td>
<td>55.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50[50, 100)</td>
<td>6.96%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100[100, 200)</td>
<td>5.61%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200[200, 400)</td>
<td>11.08%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400[400, 1550)</td>
<td>11.09%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1550[1550, )</td>
<td>10.14%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 9: Probability distribution of function duration

Fig. 10: Invocation pattern of the generated workload

Benchmarks. To evaluate the performance, we generate serverless function workloads based on the Azure Functions traces [12]. The Azure trace contains 1,980,951 function execution times and invocation timestamps spanning 14 days. Motivated by the experimental settings in previous study [23], we analyze the execution time of function invocations. Fig. 9 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 [23]. 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 [23]. Additionally, in conjunction with the Azure trace, we follow
the process in Listing 1 to create AWS S3 clients (I/O functions) and evaluate the resource multiplexer module in FaaSBatch. We replayed the total of 800 invocations made within 1 minute (from 22:10 to 22:11) of the Azure Day 13 trace. And the invocation patterns are shown in Fig. 10. During the evaluation of I/O functions, we found that such a high function concurrency causes the accumulation of tasks, which in turn leads to worker VM downtime. Thus, to evaluate the I/O functions, we make use of the first 400 function invocations of the Azure trace. We choose this representative function invocation pattern because it serves as a strong indicator of the burstiness of serverless functions.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of FaaSBatch driven by Azure traces [12]. 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. We plot CDFs of the invocation latency of different approaches for CPU-intensive and I/O functions in Fig. 11 and Fig. 12. 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.

Scheduling Latency When Executing CPU-intensive Functions: Fig. 11(a) plots the CDF of scheduling latency caused by the four strategies when executing the CPU-intensive functions. It shows that our FaaSBatch takes the shortest time to dispatch function invocations to containers, which demonstrates the superiority of FaaSBatch over the Vanilla and SFS in terms of scheduling latency. And this advantage will become more expressive as the number of concurrency increases. On the other hand, the better-performing policy Kraken is comparable to FaaSBatch in terms of decision time but starts to show a gap between itself and FaaSBatch after the 96%-th latency (please see the red vertical line).

Scheduling Latency When Executing I/O Functions: When it comes to the CDF of scheduling latency caused by the four strategies’ I/O functions, we can observe from Fig. 12(a) that FaaSBatch provides a significant advantage over SFS and Vanilla in terms of decision-making time. For example, 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 in most cases. However, the queuing overhead incurred by Kraken impacts the extended invocation latency (detailed in Sec. V-A3).

2) Cold-start Latency: The distribution of the cold-start time required by the four strategies is shown in Fig. 11(b) and Fig. 12(b). A zero cold-start latency signifies that a function invocation is associated with a keep-alive container for processing. In other cases, the factors influencing cold-start latency primarily involve two aspects, namely, the number of containers and the image size of a container [7], [24]. Since we provide an identical container image for different function invocations in our experiments, the cold-start latency increases with the number of provisioned containers. Therefore, lower cold-start latency means fewer containers are provisioned.

Cold-start Latency When Executing CPU-intensive Functions: Fig. 11(b) shows the CDF of cold-start latency when executing CPU-intensive functions. And our FaaSBatch gains significant savings in cold-start overhead. 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.

Cold-start Latency When Executing I/O Functions: Fig. 12(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. In contrast to the cold-start latency when executing the CPU-intensive functions, the cold-start latency caused by the four strategies is all reduced. The reason behind this is the number of evaluated function invocations is different (see benchmark setting in Sec. IV), which results in different requirements for booting containers.

3) Execution and Queuing Latency: There are two types of latency measurements. The first one is execution latency, which measures the amount of time CPUs take to execute an invocation (see Fig. 11(c)). The second one is the sum of execution latency and queuing latency (see Fig. 12(c)). Please note that only Kraken policy has queuing latency when executing concurrent invocations, which is labeled as Kraken: Exec+Queue (see the purple curves in Fig. 11(c) and Fig. 12(c)).

Execution and Queuing Latency When Executing CPU-intensive Functions: Fig. 11(c) shows the execution latency of the four strategies when executing CPU-intensive functions.
It shows that SFS improves the efficiency of short functions at the cost of the execution time of long functions, however, it fails to achieve a strong substitution effect in the function invocation pattern we set up (please see Fig. 9 and Fig. 10). Vanilla, on the other hand, delivers similar performance to FaaSBatch. However, 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 the other three strategies. Unfortunately, its processing time (marked with Kraken: Exec+Queue in Fig. 11(c)) is much higher since Kraken generates huge queued invocations.

Execution and Queuing Latency When Executing I/O Functions: Fig. 12(c) depicts the execution latency of the four strategies when executing I/O functions. Fig. 12(c) demonstrates that our FaaSBatch significantly cuts back invocation latency. 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. And the function execution latency of Vanilla, SFS, and Kraken, varies widely, from around 10 milliseconds to as high as 10 seconds. On top of the queuing overhead caused by Kraken, 50% of the I/O functions take at least 1 second to complete the invocation. Unlike Vanilla, SFS, and Kraken, the creation of redundancy resources is highly avoidable in FaaSBatch. Thus, the execution latency of FaaSBatch exhibits little variation, ranging from 10 ms to 100 ms. Specifically, almost all function invocations in FaaSBatch accomplish execution within a short time range between 10 ms to 100 ms. These results illustrate that FaaSBatch enables redundant resource multiplexing at the granularity of a single container.

In a nutshell, powered by the invocation mapper module, FaaSBatch is capable of dispatching the concurrent invocations received by a serverless computing platform into a single container, where FaaSBatch executes the invocations in parallel using the inline-parallel producer module. The collaboration of the invocation mapper module and the inline-parallel producer module enables FaaSBatch to reduce a considerable number of provisioned containers while ensuring the execution latency of function invocations, thereby lowering the cold-start overhead.

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; and the memory footprint of resource instances is derived by observing the memory changes around the creation procedure. Fig. 13 and Fig. 14 show the resource overheads of CPU-intensive and I/O functions under different experimental settings. Specifically, Fig. 13(a), Fig. 14(a) and Fig. 14(d) show the memory usage (mainly consumed by the provisioned containers) of the four strategies, and Fig. 13(a) and Fig. 14(a) demonstrate the system memory usage of CPU-intensive and I/O functions, respectively. Fig. 14(d) depicts the memory consumption for each invocation to execute the creation of AWS S3 client.

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

Memory Resource Overhead When Executing CPU-intensive Functions: Again, Fig. 13(a) shows the total system memory usage caused by the four compared strategies with
Number of provisioned containers

- Vanilla: 200
- Kraken: 400
- SFS: 600
- FaaSBatch: 800

Memory footprint for each I/O client

- Vanilla: 20
- Kraken: 40
- SFS: 60
- FaaSBatch: 80

CPU consumption

- Vanilla: 0.01
- Kraken: 0.02
- SFS: 0.03
- FaaSBatch: 0.04

The approach of mapping invocations into a single container slashes the number of provisioned containers during the execution.

The Number of Provisioned Containers When Executing CPU-intensive Functions: Fig. 13(b) 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. 13(b) that FaaSBatch is superior to Vanilla, Kraken, and SFS spawn in terms of spawned containers. For example, Vanilla, Kraken, and SFS averagely spawn more containers than FaaSBatch by 85.79%, 12.44%, and 86.81%, 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.

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

The Number of Provisioned Containers When Executing I/O Functions: The number of provisioned containers 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 93.80% and 93.96%, respectively. Please note that on average, 266.25 and 273.25 containers are 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 93.80% and 93.96%, respectively. Please note that on average, 266.25 and 273.25 containers are provisioned caused by the policies with respect to different dispatch intervals under the I/O function workload.

- Vanilla: 200
- Kraken: 400
- SFS: 600
- FaaSBatch: 800

- Vanilla: 20
- Kraken: 40
- SFS: 60
- FaaSBatch: 80

- Vanilla: 0.01
- Kraken: 0.02
- SFS: 0.03
- FaaSBatch: 0.04

The root cause is that FaaSBatch leverages its resource multiplexer module to eliminate resource redundancy. Moreover, Fig. 13(a) and Fig. 14(a) 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.

Memory Resource Overhead When Executing I/O Functions: Fig. 14(a) 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. 14(a) 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 69.72% to 90.39% and 44.83% to 89.19%, respectively. On the other hand, Kraken achieves a memory resource optimization comparable to FaaSBatch. This is to be expected since Kraken is mainly focusing on optimizing the number of container starts and thus reducing memory consumption. 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.

Memory Resource Overhead When Executing I/O Functions: Fig. 14(a) 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. 14(a) 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 69.72% to 90.39% and 44.83% to 89.19%, respectively. On the other hand, Kraken achieves a memory resource optimization comparable to FaaSBatch. This is to be expected since Kraken is mainly focusing on optimizing the number of container starts and thus reducing memory consumption. 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.

Memory Resource Overhead When Executing I/O Functions: Fig. 14(a) 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. 14(a) 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 69.72% to 90.39% and 44.83% to 89.19%, respectively. On the other hand, Kraken achieves a memory resource optimization comparable to FaaSBatch. This is to be expected since Kraken is mainly focusing on optimizing the number of container starts and thus reducing memory consumption. 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.
only 16.5 to achieve the same computing purpose. In other words, FaaSBatch optimizes the performance of Kraken by 78.28%. The following aspects can explain the superior performance delivered by FaaSBatch. To satisfy the executed SLOs, Kraken must start multiple containers for a set of concurrent invocations to reduce queuing overhead. FaaSBatch on average serves 24.39 (400/16.5) invocations with one container, while Vanilla, SFS, and Kraken must activate a container when receiving 1.50 (400/266.25), 1.46 (400/273.25) and 5.26 (400/76) 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 54.25 when processing I/O functions to 76 when processing CPU-intensive 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. 13(c) and Fig. 14(c) 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 consumes CPU resources.

CPU Resource Overhead When Executing CPU-intensive Functions: Fig. 13(c) depicts the CPU utilization of the four policies with respect to different dispatch intervals under the CPU-intensive function workload. Unsurprisingly, FaaSBatch outperforms 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. 11(a)); (2) Vanilla, SFS, and Kraken incur more cold starts, and starting containers take up more CPU resources (please see Fig. 11(b)).

CPU Resource Overhead When Executing I/O Functions: Fig. 14(c) 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 81.39% to 91.15%, 79.89% to 90.33%, and 84.76% to 93.12%, 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.

4) Memory Overhead for Resource Creation: Fig. 14(d) demonstrates more detailed memory utilization of the four compared policies. It reveals that the average memory footprint of creating an AWS S3 client when using Vanilla, SFS, or Kraken is approximately 15 MB, while our FaaSBatch merely consumes an average of 0.87 MB thanks to its salient feature of multiplexing resources. In other words, FaaSBatch requires only 1/16 the memory consumed by other policies to serve the same function invocations.

5) Efficiency Varying the Dispatch Intervals: Finally, we focus on the efficiency of FaaSBatch at different dispatch intervals. In this group of experiments, we conduct experimental analysis on the results driven by the I/O function workload, because Fig. 13 (CPU-intensive function) and Fig. 14 (I/O function) exhibit similar performance trends.

Fig. 14 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.

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 [15] 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 [14], BATCH [25], Fifer [13], Kraken [16] and Cypress [17]. 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 [26]. 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 [23].

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) FaaSBatch enables serverless computing platforms to map concurrent invocations into batches, thus avoiding the creation of additional containers and lowering resource utilization. (ii) FaaSBatch executes the batched invocations of the populated single container in parallel to reduce invocation latency. (iii) FaaSBatch multiplexes redundant resources created by all functions of the populated single container to slash invocation latency and memory usage.

VII. CONCLUSIONS

In this study, we proposed FaaSBatch - a holistic batch system to reduce invocation latency and resource overhead in serverless computing. Motivated by the observation that provisioning one single container to carry out all concurrent invocations provides performance equivalent to launching a container per single invocation, we designed an Invocation Mapper module and an Inline Parallel Processor module embedded in FaaSBatch to dispatch and execute invocations in a fold-and-spread manner. In conjunction with the Resource Multiplexer module that monitors and reuses redundant resources, resource pressure and invocation latency inside containers are further suppressed, thereby eliminating the requirement to create additional redundant resources. We conducted extensive experiments to validate the effectiveness of our FaaSBatch. The experimental results unveil that FaaSBatch remarkably cuts back the invocation latency and the resource overhead of the existing solutions by up to 92.18% and 90.39%, respectively.

ACKNOWLEDGEMENTS

This work is sponsored by the National Natural Science Foundation of China under Grant No. 62072214, the Guangdong Basic and Applied Basic Research Foundation under Grant No.2021B1515120048, the Science and Technology Planning Project of Guangzhou under Grant No.202103000036, the Open Project Program of Wuhan National Laboratory for Optoelectronics No.2020WNLokf006, the Industry-University-Research Collaboration Project of Zuhai under Grant No.ZH20210701210048PWc, and the Outstanding Innovative Talents Cultivation Funded Programs of Zhuhai under Grant No.ZH2022010048WPC, and the Industry-University-Research Collaboration Project of Wuhan National Laboratory for Optoelectronics No.2020WNLOKF006, the Industry-University-Research Collaboration Project of Zuhai under Grant No.ZH20210701210048PWc, and the Outstanding Innovative Talents Cultivation Funded Programs for Doctoral Students of Jinan University. The corresponding author of this paper is Yuhui Deng.