HashCache: Accelerating Serverless Computing by Skipping Duplicated Function Execution

Zhaorui Wu, Yuhui Deng, Yi Zhou, Lin Cui, and Xiao Qin

***Abstract*—Serverless computing is a leading force behind deploying and managing software in cloud computing. One inherent challenge in serverless computing is the increased overall latency due to duplicate computations. Our initial investigation into the function invocations of serverless applications reveals an abundance of duplicate invocations. Inspired by this critical observation, we introduce *HashCache*, a system designed to cache duplicate function invocations, thereby mitigating duplicate computations. In HashCache, serverless functions are classified into three categories, namely, computational functions, stateful functions, and environment-related functions. On the grounds of such a function classification, HashCache associates the stateful functions and their states to build an adaptive synchronization mechanism. With this support, HashCache exploits the cached results of computational and stateful functions to serve upcoming invocation requests to the same functions, thereby reducing duplicate computations. Moreover, HashCache stores remote files probed by stateful functions into a local cache layer, which further curtails invocation latency. We implement HashCache within the *Apache OpenWhisk* to forge a cache-enabled serverless computing platform. We conduct extensive experiments to quantitatively evaluate the performance of HashCache in terms of invocation latency and resource utilization. We compare HashCache against two state-of-the-art approaches - *FaaSCache* and *OpenWhisk*. The experimental results unveil that our HashCache remarkably reduces invocation latency and resource overhead. More specifically, HashCache curbs the 99-tail latency of FaaSCache and OpenWhisk by up to 91.37% and 95.96% in real-world serverless applications. HashCache also slashes the resource utilization of FaaSCache and OpenWhisk by up to 31.62% and 35.51%, respectively.**

***Index Terms*—Cloud computing, function-as-a-service, performance optimization, serverless computing.**

Manuscript received 20 April 2023; revised 2 October 2023; accepted 5 October 2023. Date of publication 10 October 2023; date of current version 24 October 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62072214, in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2021B1515120048, and in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics under Grant 2020WNLOKF006. The work of Xiao Qin's was supported by the U.S. National Science Foundation under Grants IIS-1618669 and OAC-1642133. Recommended for acceptance by K. Gopalan. *(Corresponding author: Yuhui Deng.)*

Zhaorui Wu and Lin Cui are with the Department of Computer Science, Jinan University, Guangzhou 510632, China (e-mail: diom_wu@163.com; tcuilin@jnu.edu.cn).

Yuhui Deng is with the Department of Computer Science, Jinan University, Guangzhou 510632, China, and also with the Wuhan National Laboratory for Optoelectronics, Wuhan 430079, China (e-mail: tyhdeng@jnu.edu.cn).

Yi Zhou is with the 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/TPDS.2023.3323330

I. Introduction

SERVERLESS computing - a cloud computing model - has captured popularity among software engineers to develop and deploy applications without concern about underlying infrastructures. Generally speaking, serverless computing leverages a Function-as-a-Service (FaaS) module to decompose computation into a set of functions. To deploy applications on a FaaS platform, developers are only obliged to upload the code of one or several functions and set trigger events like HTTP requests and timers that invoke these functions. The FaaS platform is in charge of deploying the uploaded functions and resource provisioning. Thanks to serverless computing's advantages such as rapid deployment, enhanced scalability, and low cost, FaaS is widely adopted by various enterprises, especially when deploying computing-intensive applications [1], [2], [3]. Representative FaaS platforms include AWS Lambda [4], Google Cloud Functions [5], and Azure Functions [6]. Open source communities not only embrace this cutting-edge technology, but also implement numerous FaaS systems represented by Apache OpenWhisk [7] and OpenFaaS [8].

FaaS platforms execute functions in an isolated runtime environment created by the virtualization techniques such as docker container [9] or ultra-lightweight Virtual Machine (VM) [10]. The creation of an isolated runtime environment involves two phases, namely, (i) creating and launching the base execution environment and (ii) fetching and installing necessary libraries and dependencies. The time interval of building an execution environment is referred to as a *cold start* phase. FaaS is slated to run functions only after the cold start phase is accomplished. In event-driven serverless computing platforms, a function invocation involves two steps: (i) invoking the function based on predefined input parameters and (ii) receiving a computing result of the invoked function. For stateful functions that depend on external data [11], [12], [13], [14], step (i) encompasses the procurement of these external states. In other words, the procurement of external states by stateful functions can also be regarded as the acquisition of input parameters. We propose to model a serverless function invocation as three-component parts, namely, input parameters, a black-box of function execution, and computing results. We infer that for most of the serverless functions (computational and stateful functions, detailed in Section II-B), the only factor that changes the result of their computation and execution process is their input parameters.

To validate the efficacy of our model and its underlying assumptions, we conduct a series of experiments. The experimental results confirm that in realistic applications, computing

results and execution course of the functions are solely determined by input parameters. On top of that, our findings reveal that because of the existence of duplicate input parameters among various function requests, some functions may be repeatedly executed multiple times – causing the wastage of computing resources and application latency (see Section II-B2 for the details). Such an intriguing observation galvanizes us to devise *HashCache* – a novel caching system that avoids duplicate computations by *caching computing results of serverless functions*. More prosaically, HashCache caches computing results on a serverless platform, allowing subsequent invocations to directly retrieve expected results from the platform without starting new containers and executing functions. We create a module - *Behavior Monitor*, an embedded in the container image, to gauge whether computing results of functions are effective caching candidates. We advocate for classifying serverless functions into three categories, namely, computational functions, stateful functions, and environment-related functions. Accordingly, we develop an *Action Mapper* module for the purpose of caching functions' computing results. The dependency state of stateful functions is maintained in the *State Bridge* module, which promptly guarantees the validity of cached results in *Action Mapper* when the state is updated. To bolster retrieval efficiency and reduce memory overhead, the Action Mapper constructs a *function-input-output* relation based on the hashing technology and uses LRU strategy to manage the cache of results. We implement a hash table using the 32-bit MurmurHash technique [15], where the sensitivity of the cache size is articulated in Section IV-D.

Our HashCache exhibits three salient and distinctive features. First, HashCache equips serverless computing platforms to reuse computation results cached in the Action Mapper upon arrivals of invocation requests for the same functions. Second, our function-level caching mechanism delivers the same optimization effect when an application is deployed as a function chain. Third, HashCache furnishes a State Bridge module to store the latest versions of states inquired by stateful functions, thereby curbing invocation latency. We implement our HashCache on the Apache OpenWhisk - a popular open source serverless functions cloud platform [7]. In summary, we make the following contributions in this study:

- We examine the execution of realistic application suites and reveal the existence of repeatedly invoked function executions due to duplicate input parameters among the functions.
- We classify serverless functions into three categories – *computational functions*, *stateful functions*, and *environment-related functions*, among which computational and stateful functions' results are eligible for caching. The rationality of this classification is feasible through the analysis of wide-ranged serverless applications.
- We design a cache system called *HashCache* that eradicates duplicate computations by directly reusing the cached results of functions, thereby achieving low invocation latency measures. Moreover, HashCache maintains states accessed by stateful functions to further curtail invocation latency.

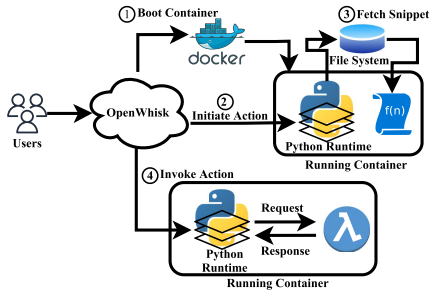


Fig. 1. Overview of OpenWhisk architecture.

- We implement HashCache on the Apache OpenWhisk, where we undertake extensive experiments to evaluate the performance of HashCache. The experimental results illustrate that HashCache remarkably shortens invocation latency and resource overhead compared with the other state-of-the-art approaches.

The rest of the paper is organized as follows. Section II introduces background and motivation. The design and implementation of HashCache are articulated in Section III. Section IV validates the performance of HashCache through quantitative evaluation. Section V paints a picture of prior studies related to our work. Last, Section VI concludes this study.

HashCache's implementation is open-sourced for community adoption at https://github.com/dscLabJNU/HashCache.

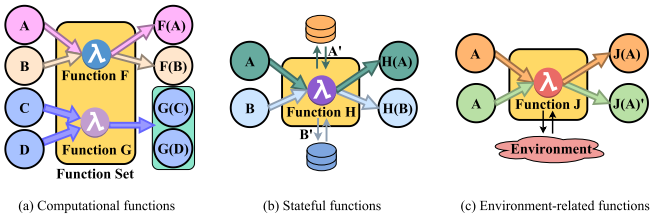
II. Background and Motivations

A. The OpenWhisk Background

OpenWhisk empowers software engineers to deploy serverless functions written in different programming languages along with specific event triggers. OpenWhisk limits the size of functional dependencies to 48 MB to retain simple behaviors of individual functions. In doing so, combining multiple functions to form function chains – sequences in OpenWhisk – is a common and viable practice in complex applications. To deploy an application on OpenWhisk, developers create a series of actions, each of which consists of user functions, function dependencies, and a proxy that implements the standard integration protocol. In OpenWhisk, the execution unit of an action is a docker container; OpenWhisk provides several container runtime options to an application deployment, depending on the programming languages such as Python, Java and NodeJs. To enable interaction between OpenWhisk and function invocations, each function's container runtime has a built-in web proxy called *ActionLoop* to handle HTTP requests from OpenWhisk.

Now let us use a Python application to elaborate on the typical three steps through which OpenWhisk invokes an action (see Fig. 1):

Step 1: OpenWhisk boots a container (see ① in Fig. 1) from a docker image after an action invocation request is admitted. Since a proxy that communicates with OpenWhisk is built in the docker image, the proxy is activated as a service after the container is booted. In other words, a Python container runtime with the proxy functionality is booted as illustrated in Fig. 1.



(a) Computational functions (b) Stateful functions (c) Environment-related functions

Fig. 2. Illustration of classified functions.

It is noteworthy that this step only occurs during the cold start phase where docker image downloads may take place.

Step 2: After the container is working in full swing, OpenWhisk initiates the requested action by sending a */init* HTTP request (see ② in Fig. 1). Next, OpenWhisk collects and injects a code snippet associated with the invoked action and its dependencies into the container (see ③ in Fig. 1).

Step 3: Finally, OpenWhisk invokes a requested action by releasing a */run* HTTP request (see ④ in Fig. 1). In particular, the runtime executes the main function of the requested action according to wrapped input parameters. Upon the completion of the action, the runtime passes the response to OpenWhisk, which eventually relays the response back to the user.

Existing Optimization Methods: A handful of studies accelerated the function execution by improving the three steps of the serverless function execution process. For example, some researchers attempted to slash the sandbox startup latency in Step 1 by function scheduling strategies or virtual environment accelerating strategies [10], [16], [17]. In other studies, the function startup process was optimized by reusing the initialization information in Step 2 [18]. And other optimizations endeavored to cache container instances or fetch transferred files in Step 3 [19], [20]. Our HashCache distinctly differs from the aforementioned methods in that HashCache leverages a container runtime modification in Step 1 to monitor action behaviors, monitoring the execution information in Step 2, followed by applying a caching strategy in Step 3 based on valuable monitoring information.

B. Motivations

HashCache is conducive to bypassing function executions by caching computing results of serverless functions, reducing invocation latency and resource utilization via avoiding duplicate function execution. A primary challenge in this study is that certain function executions are unsuitable for bypassing due to the diversity of serverless applications. For example, if the execution of a function designed to modify an external state is bypassed, serious data inconsistency problems will become inevitable.

1) Serverless Function Classification: According to the characteristics of serverless applications, we classify serverless functions into three categories, namely, *computational functions*, *stateful functions*, and *environment-related functions* (see Fig. 2). The classification phase is of paramount importance because it identifies functions that are good fit for our caching strategy.

Computational Functions: Amid the execution of a computational function, the only factors that affect its output are the input parameters. Typical computational functions include those that compute the Fibonacci sequence, MapReduce WordCount application, and the like. Fig. 2(a) depicts the relationship between the input and output of computational functions F and G. Assuming that two different input parameters A and B are issued in different invocations to function F, then function F will produce two different output results $F(A)$ and $F(B)$.

Function G receives two different invocations with input parameters C and D, respectively, it may produce two identical results $G(C)$ and $G(D)$ (i.e., $G(C) = G(D)$). For example, to determine if the numbers 4 and 6 are even. These two cases indicate that function callers are only concerned with the computational functions' execution results, which are affected by nothing but input parameters.

Stateful Functions: A stateful function may generate requests for external resources (e.g., AWS S3) during its execution, resulting in network traffic. Fig. 2(b) depicts the execution course of stateful function H. Upon receiving input A, function H inquires about another input A' from external object storage. Then, with both A and A' in place, function H produces output $H(A)$. Similarly, H produces $H(B)$ for input B. These function executions shed bright light on I/O operations between function instances and object storage systems. There is a raft of scenarios in which FaaS users set up stateful functions. For example, invoking a machine learning function requires training data to be loaded. Another representative example is persisting data from a web server to a database system.

Environment-related Functions: When it comes to environment-related functions, the output varies depending on the resource utilization of the current system or the container runtime. Typical examples of environment-related functions include those evaluating startup latency or resource utilization. Fig. 2(c) shows an environment-related function J that receives the same input A twice. Because function J ought to interact with the current runtime environment to obtain runtime information (e.g., finding the current timestamp and polling resource utilization) during execution, function J produces two different results where $J(A)!= J(A)'$.

Function-classifying Strategy Validation: To verify the effectiveness of the above function-classifying strategy, we examine the function workflows driven by the eight realistic workloads or benchmarks: FunctionBench [21], Faastlane [22], FaaSProfiler [23], ServerlessBench [2], TrainTicket [24], HotelReservation [25], SocialNetwork [25] and FireBase [26]. We derive the type of these functions by combining manual analysis and dynamic detection (detailed in Section III-B), which obtain the same results. Table I summarizes the experimental results after removing simple functions (e.g., functions that print "hello world"). The finding reveals that our function-classifying strategy effectively covers 100% of all the functions. In particular, among the total number of 129 functions, 25% are computational functions, 72% accounts for stateful functions, and 3% belongs to environment-related functions.

2) Computational Duplication: To further validate the feasibility of avoiding function execution by the virtue of

TABLE I
DISTRIBUTION OF THE REAL FUNCTION WORKFLOW

	Comp.	*State.*	*Env.*
FunctionBench [21]	5	21	0
FireBase [26]	4	17	0
FaaSProfiler [23]	7	0	4
ServerlessBench [2]	6	3	0
Faastlane [22]	3	8	0
TrainTicket [24]	6	27	0
HotelReservation [25]	1	5	0
SocialNetwork [25]	0	12	0
Total	32 (25%)	93 (72%)	4 (3%)

* ***Comp.*** indicates the computational functions
* ***State.*** represents the stateful functions
* ***Env.*** refers to the environment-related functions

caching results of computational and stateful functions, we explore three representative real-world serverless application suites: TrainTicket [24], HotelReservation [25] and SocialNetwork [25]. Specifically, TrainTicket comprises 33 functions, however, we found that only 29 functions are implemented within the application suite, while the remaining 4 functions are in fact, not invoked. HotelReservation consists of 6 functions, and SocialNetwork has 12 functions. We analyze the input-output relationships of functions during the execution of these applications. The experimental setup of TrainTicket and HotelReservation are detailed in Section IV-A. For the SocialNetwork workload, we performed testing with the benchmark data, which can be found at the GitHub repository[1]. SocialNetwork has constructed a social network of a total of 100 users. It provides test cases that exceed a scale of 1,000,000 based on a normal distribution.

Duplication Overview: Performance metrics are of a great importance for validating the feasibility of our caching strategy. To this end, we propose the following four metrics to analyze the duplication in serverless function execution under three applications, focusing on functions with consistent input-output relationships during execution (either computational or stateful, collectively referred to as consistent functions.): (1) the proportion of consistent functions to all functions $\mathrm{P}_N(Const./Total)$, (2) the proportion of consistent function invocations to all function invocations $\mathrm{P}_I(Const./Total)$, (3) the proportion of duplicate invocations to all consistent functions $\mathrm{P}_I(Dup./Const.)$, where *duplicate invocations* refer to the invocations with the same input and output, (4) the proportion of the execution time of duplicate invocations to the execution time of consistent functions $\mathrm{P}_T(Dup./Const.)$. Fig. 3 depicts the performance of the three applications TrainTicket, HotelReservation and SocialNetwork in terms of $\mathrm{P}_N(Const./Total)$, $\mathrm{P}_I(Const./Total)$, $\mathrm{P}_I(Dup./Const.)$, and $\mathrm{P}_T(Dup./Const.)$. Fig. 3 reveals that all functions within the application are consistent functions, as evidenced by the fact that all $\mathrm{P}_N(Const./Total)$ values amount to 100%. This, in turn, leads to 100% of the functions invocations being consistent function invocation, reflected by the average $\mathrm{P}_I(Const./Total)$ is 100%.

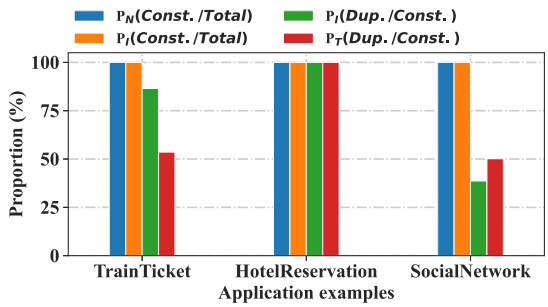


Fig. 3. Distribution of invocations for serverless applications.

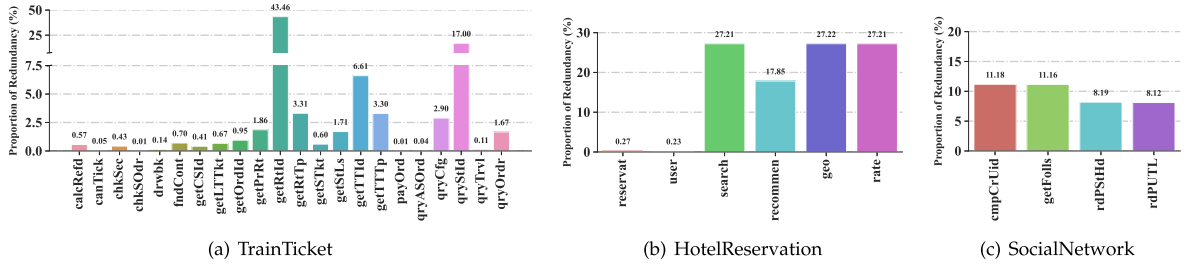
More importantly, we observe that 75.05% of consistent function invocations have duplicate inputs, resulting in the same outputs (Avg. $\mathrm{P}_I(Dup./Const.)$ is 75.05%). Execution time of these duplicate consistent functions accounts for 68% of the total execution time of the consistent functions ($\mathrm{P}_T(Dup./Const.)$ is 68%). In short, the duplicate invocations during application execution spark remarkable waste of computing resources and significant execution latency.

Function Details: Let's delve deeper into the analysis of duplicate execution of functions across the three application suites. The succeeding illustration (Fig. 4) presents the statistics of duplicate computations and stateful function invocations (essentially repeated input-output associations) during the entire execution process, i.e., the proportion of duplicate computations for each function.

Fig. 4 indicates that out of 29 total functions in TrainTicket, 22 give rise to duplicate function invocations. Among these, `canTick`, `getLTi`, `qryTrvl`, `calcRefd`, `fndCont`, and `chkSec` are computational functions, while the remaining eighteen are stateful functions. The occurrence of execution duplication in TrainTicket is largely due to its function as a ticket reservation system, where the majority of system read operations are predominantly generated by user queries. Given the user's selected point of departure and destination, applications linked to ticket inquiries tend to perform repetitive invocations quite often. More specifically, the set of locations in the ticket system remains relatively consistent, thereby leading to stable corresponding location information, train details, and route data. As a result, within a ticket booking system that is predominantly governed by read operations, it's expected to observe a substantial amount of duplicate function invocations.

When it comes to HotelReservation, it consists of a total of 6 functions, among which 1 is a computational function (Search), and 5 are stateful functions that generate a substantial number of duplicate invocations during execution. Similar to TrainTicket, HotelReservation is also a ticket query system. In HotelReservation, querying hotels around a given location is a common process during hotel booking. This process will generate numerous identical operations, such as multiple users querying the same hotel price multiple times, querying recommended places to visit under a given hotel, or submitting user information when booking a hotel.

[1]https://github.com/delimitrou/DeathStarBench.



(a) TrainTicket (b) HotelReservation (c) SocialNetwork

Fig. 4. Duplicate execution of different investigation applications.

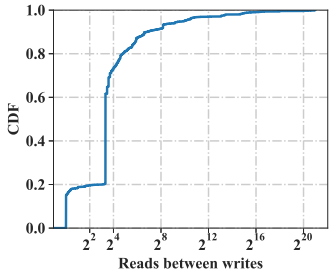


Fig. 5. CDF of reads between two consecutive writes.

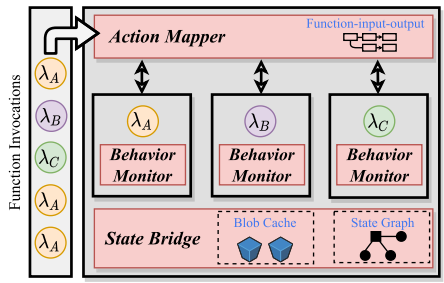


Fig. 6. Overview of HashCache.

As for SocialNetwork, it consists of 12 stateful functions, a third of which generate a large number of duplicate invocations during execution. In this application, browsing posted texts in the social network is considered a normal operation. This is also very consistent with the usage habits of social networks in real life, that is, many operations are generated in read requests. Read requests produce varying results only when the corresponding resources undergo changes, and HashCache handles this update issue with the introduced State Bridge module.

3) Infrequent External State Update: Due to the stateless nature, serverless functions have to maintain their generated intermediate data or state via access to cloud object storage like AWS S3 [27] and Azure Blob storage [28]. Duplicate invocation on stateful functions should be meticulously optimized, as bypassing updates to state leads to data inconsistency. To put the accesses patterns of cloud storage objects under the microscope, we investigate the Azure Blob trace [29], which contains 14 days of logs, including 33.1 million invocations with 44.3 million data accesses. To avoid the impact of cold data on the overall situation, we select blob entries with accesses (read + write) greater than 10 and evaluated the number of read requests between writes. The analytical results are plotted in Fig. 5, which shows the Cumulative Distribution Function (CDF) of the number of Reads Between two consecutive Writes (RBW) for each blob.

We observe from Fig. 5 that nearly 20% of RBW data are spread in single-digit (< 10) percentages, whereas the remaining 80% of RBW values exist anywhere between tens and millions. This phenomenon conforms to the commonly observed Zipf distribution in cloud computing environments [30], [31], [32]. Statistically, among the more than twenty million blob access entries evaluated, the mean value of RBW is 7292, the median value is 10, and the maximum value is more than two million. It is noteworthy that when servicing these blob accesses, numerous external requests for the identical blob will be sent to cloud object storage, thereby wasting computing resource and prolonging function invocation latency.

In summary, serverless computing scenarios rarely involve external state updates, entailing that functions commonly and repeatedly operate on the same state.

III. Design of HashCache

The aforementioned inspiring observations and analysis reveal that there exists a pressing demand behind a novel strategy to bolster the performance of serverless functions. In this regard, we propose HashCache to improve the performance of serverless functions, aiming to avoid duplicate computation, and therefore reduce invocation latency and optimize resource utilization.

A. Architecture Overview

Now we present the design of our HashCache as sketched in Fig. 6. The overarching goal of HashCache is (1) to eradicate duplicate execution by caching computing results yielded by computational and stateful functions and (2) to shorten execution latency by caching remote state requested by stateful functions. HashCache embraces three modules: *Action Mapper*, *Behavior Monitor* and *State Bridge*. A function invocation request issued by users is first analyzed by the *action mapper* to determine if a cached copy exists. Please note that the computed results of both computational and stateful functions can be targeted for caching by the action mapper. The precondition for caching the output of stateful functions is that the state they depend on remains unchanged. The state of the stateful functions is

maintained in the *state bridge*. If it is a cache hit, the action mapper directly returns the cached computing result to the requested function invocation without paying any startup and execution cost. Otherwise, it will be invoked through a typical three-step procedure articulated in Section II-A. The *behavior monitor* is responsible for communicating with *state bridge* around the execution course of a requested function, passing along the function's type of operation on the state. If an external state request is generated during function execution, the *state bridge* will intercept and handle the request, followed by a dependency established between the function and the state for adaptive synchronization of updates.

B. Behavior Monitor

The fundamental challenge for effectively caching computational results lies in the diversity of serverless functions. To tackle this challenge, we enforce the behavior monitor to help with state bridge (detailed in Section III-D) to identify and classify serverless functions based on function characteristics. More prosaically, the behavior monitor checks the network connection construction activity corresponding to the stateful functions. After the function execution is complete, the behavior monitor collectively analyzes this gleaned behavioral information, followed by determining the function type for the completed function. As previous studies on serverless systems [13], [18], [22], we implement the behavior monitor based on OpenWhisk container runtimes. In what follows, we elaborate on a detailed approach to classifying serverless functions into three different types.

Computational Functions: We adopt a proactive approach to discern computational functions. In this method, HashCache cultivates developers' ability to define customized annotation (`compute_cache`) for computational functions implemented in applications. The annotation mechanism is not only flexible but also particularly well-suited for serverless computing environments, as evidenced by existing research [22], [33]. When developers specify a function as computational during its registration, HashCache will automatically cache the computational results upon invocation. This approach significantly improves computational efficiency by reducing the need for duplicate computations, thus offering distinct advantages in computation-intensive tasks.

Stateful Functions: As evidenced by recent studies [11], [13], [34], a stateful function preserves and updates state information between multiple or across different invocations. Currently, using TCP/IP protocol to manage shared state with cloud database (CouchDB, MongoDB) or object storage (AWS S3, Azure Blob Storage) is the most common way to ensure data reliability. Taking advantage of this practice, we examine changes in network connections before and after function execution to dig out whether a function is stateful. Remote state accesses are identified by checking the `/proc/net/tcp` file[2] in Unix-like systems, where file provides information about currently activate TCP connections. Hence, we classify a function as a stateful one if the `/proc/net/tcp` file, which logs active network connections pertaining to remote state access, shows a difference when compared before and after the execution of function requests. Please note that a container environment can only serve a single function due to the presence of package dependencies. This one-to-one mapping relationship simplifies the `/proc/net/tcp` file, and the behavior monitor facilitates accurate identification. Due to the read-skewed nature of stateful function access (see Section II-B3), as with computational functions, we define an annotation flag (`state_cache`) to turn on the computing result cache.

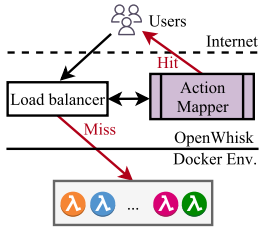


Fig. 7. Action mapper.

Environment-related Functions: As mentioned before, we provide two annotations, `compute_cache` and `state_cache`, to specify the cache ability of computational and stateful functions, respectively. Therefore, we exclude the identification of environment-related functions since we can actively identify and filter computational and stateful functions, which are the key areas of interest for HashCache's performance enhancements.

To sum up, HashCache determines whether to cache the computational results of invocations based on the annotations associated with the functions. If a function is not annotated, the action mapper module (detailed in Section III-C) will abstain from caching its computational results. The behavior monitor module serves as an automated detection mechanism for determining whether a function is stateful, a process that is independent of whether the function is annotated. In addition, state caching (in the state bridge module that described in Section III-D) will still occur even if the function lacks annotation.

C. Action Mapper

HashCache utilizes the action mapper to avert duplicate execution of the annotated functions that produce identical results, thereby achieving reduced invocation latency and resource overhead. We implement the action mapper into OpenWhisk controller module to handle function invocation requests (see Fig. 7). In particular, the controller serves as the central manager for multiple invokers in a cluster. In addition, the OpenWhisk controller supports a masterslave mode by setting multiple replications[3], allowing the contents of multiple action mappers to be shared across different controllers. This enhances the scalability of HashCache. Upon receiving invocation requests, the OpenWhisk controller module distributes the invocations to different

[2]https://www.kernel.org/doc/Documentation/networking/proc_net_tcp.txt.

[3]https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/configurationChoices.md#replication-factor

invokers on various worker nodes based on a load balancing protocol. The invoker then launches a container and executes the function invocation. By integrating the action mapper into the OpenWhisk controller module, it is able to determine and avoid duplicate computations before the request is assigned to the invoker.

The action mapper, maintaining *function-input-output* relationships in memory, searches for a matched entry in the cached relationships after intercepting the function invocation request and its input arguments. If an incoming request hits, the action mapper directly returns a corresponding result to serve the request. Otherwise, the function invocation request will be handled through the normal three-step procedure stipulated in Section II-A. In this case, HashCache checks the availability of cached results after the execution of each function using HTTP status codes. As a result, HashCache skips the results of functions that fail to execute, thus ensuring the validity of the cached results.

It is worth noting that we apply a hashing technique to input parameters of functions to curtail memory overhead and speed up the matching process. Furthermore, the LRU strategy is leveraged to schedule the cached results, and we evaluate the sensitivity of the cache size in Section IV-D.

Now let us take an example to illustrate the caching process orchestrated by the action mapper. Let $F \rightarrow (A, B)$ denotes a function invocation, where A and B are its input parameters. For the first request for invoking this function, we associate the invoked function with its input arguments as a relationship $F \rightarrow Hash(A, B)$. Next, the action mapper stores this relationship and starts the execution of the function, waiting for its computational results to form its corresponding *function-input-output* relationship. Once the function execution is completed, the action mapper first sends the computational result to the user and; then, function F and hashed input arguments $Hash(A, B)$ are associated with computational result $F(A, B)$ by storing a relationship $F \rightarrow Hash(A, B) \rightarrow F(A, B)$ in the action mapper. In doing so, if future requests inquiring about the same function invocation are issued, the action mapper will directly serve those requests with the cached computational results. To maintain correct computational results, HashCache reclaims the cached *function-input-output* relationships when HashCache acquires update requests for the function because developers may resubmit serverless functions due to version updates.

In a nutshell, HashCache curbs invocation latency and resource utilization by impeding duplicate execution of computational functions sharing the same results.

D. State Bridge

The *state bridge* is primarily obliged to maintain remote state requested by stateful functions, thereby reducing invocation latency by working with the action mapper and the behavior monitor. All external communication of stateful functions passes through the state bridge, which is responsible for connecting functions, external states, and the serverless platform. When the behavior monitor identifies a stateful function, it redirects the external state request to the state bridge, which analyzes and takes over the request. State bridge introduces two submodules *State Graph* and *Blob Cache* (Fig. 9) to facilitate the remote state maintenance.

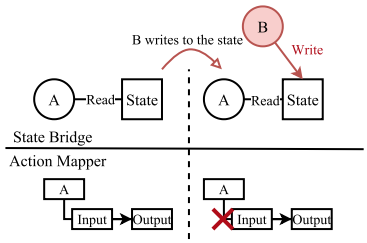


Fig. 8. State graph.

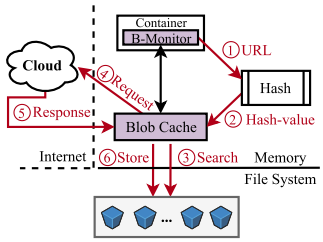


Fig. 9. Blob cache.

1) State Graph: As illustrated on Fig. 8, the state bridge creates a graph structure for the states operated by functions, where a state is connected to several actionable functions. For each state request, the state bridge records the manner in which the state is accessed (read or write). If the current request is read, it says that the current state accessed by the function is the latest version, marking the current function output as a cache candidate. For instance, the left side of Fig. 8 shows function A reads its state, and the action mapper stores the relationship $A \rightarrow Hash(Input) \rightarrow A(Input)$. Conversely, the state receiving a write request implies an update requirement. Since a state connects multiple functions, we mark all the function outputs as uncacheable targets and notify the action mapper to reclaim the corresponding cached data. As shown on the right side of Fig. 8, function B sends a write operation to the shared state; therefore the action mapper reclaims the relationship of function A. After that, the incoming function invocation will go through a typical three-step procedure articulated in Section II-A. Upon detecting these operations, the state graph always checks the last modification time (*mtime* in Linux) of the target object to prevent it from being modified by means other than the function, such as manual uploads by users. Specifically, if the state graph module detects a mismatch between the mtime value of the state that the function depends on and the latest value, it implies that the state stored externally has been modified. In this case, the state graph will clear the cached relationships in the action mapper, and rely on the blob cache (detailed in III-D2)to obtain the latest state the function depends on. This approach guarantees that even if a function's resources are updated through means other than the function itself, the function can still access the latest state when requested. Note that we only preserve the representative

symbols (IDs) of states and functions, not the entity contents. Therefore, for fast access and update, we store this graph data result in memory.

Overall, the distribution of reads and writes to external state by serverless computing functions is described in Section II-B3, confirming that HashCache brings forth acceleration benefits to heavily read-skewed stateful functions.

2) Blob Cache: Considering that various functions possibly read and write to the same state, we introduce the blob cache to shorten the latency of requesting state across functions. In collaboration with the behavior monitor, the blob cache fulfill its responsibility through the following three key steps (see Fig. 9).

Step 1. Hashing Request URLs: The behavior monitor redirects the URL of a requested state for hashing to avoid improper matches caused by character translation (see ① in Fig. 9). For example, character "#" corresponds to the URL encoding value "%23". Thus, URL *"https://example.com/file/example-file.obj"* becomes *-5332493745412849051* after hashing. Next, the redirected request coupled with its corresponding hash value is delivered to the blob cache (see ② in Fig. 9).

Step 2. Looking Up Matched Entries: When the blob cache obtains a redirected request, it searches the local file system to see if a requested state exists (see ③ in Fig. 9). If so, blob cache first reads the *mtime* of the matched state from the local file system; then, blob cache requests the remote server where the requested state resides for its *mtime* using its URL (see ④ and ⑤ in Fig. 9). Next, if the two mtimes echo each other, the redirected request will be served by the cache copy to be dispatched to the function.

Step 3. Serving with Up-to-date States: Otherwise, if the two modification times disagree with each other, the redirected request is served through the normal process - retrieving the requested state from a remote server where the state resides (see ④ and ⑤ in Fig. 9). After gathering the file, HashCache stores the state on a local file system, overwriting an out-of-date local copy (see ⑥ in Fig. 9).

The blob cache is to eliminate the need for multiple functions that request the same object over the network. For example, let us consider functions A, B, and C, all of which establish a relationship with cloud object O in the State Graph. When function A makes a write request to O, the previously cached computation results of the three functions in the Action Mapper need to be invalidated. Now, if function B makes a read request to O, the Blob Cache retrieves the latest version of object O from the cloud and stores it locally. If function C also makes a read request to O at this point, the Blob Cache can directly provide function C with the locally cached object O, thereby eliminating the need for network retrieval.

Importantly, within the context of cloud storage objects that provide modification time (mtime) information, the blob cache is state-type agnostic, as the cache reads content from TCP requests of accessing state, and persists it to the local file system. Therefore, the blob cache supports accessing a wide range of state types, such as images, binary objects, and files, to name just a few. However, it's worth noting the limitations and assumptions regarding the blob cache. The current implementation is tailored for cloud storage systems that can provide mtime information for objects. For databases or other storage systems that do not offer access to an object's mtime, the blob cache currently cannot be used to improve the efficiency of element retrieval.

IV. Performance Evaluation

A. Experimental Methodology

We implement HashCache on top of OpenWhisk [7], an open-source serverless computing platform, to undertake performance evaluation in terms of invocation latency and resource utilization. Moreover, in our experiments, we utilized LRU as the cache replacement policy in the action mapper to manage the computation results of functions. We set the cache size of LRU to 5,000 elements, meaning that the cache can store up to 5,000 function results. The sensitivity of the cache size is articulated in Section IV-D.

Setups and Baselines: We assess our prototype implementation on a Kubernetes cluster with 80 compute cores. The cluster consists of five nodes, with each node being a virtual machine with 16 vCPUs and 16 GB of memory. We compare our HashCache against two state-of-the-art strategies: OpenWhisk [7] and FaaSCache [20]. OpenWhisk is a serverless functions platform powered by Apache, and we pull the OpenWhisk from the GitHub repository on Aug 30, 2021 for a comparison purpose.[4] FaaSCache strives to lower cold start overhead and invocation latency through a function caching strategy and an in-memory scheduling tactic. In addition, we have set the idle time for containers at default (ten minute). This setting was uniformly applied to HashCache, FaaSCache, and OpenWhisk in all of our experiments.

Evaluation Metrics: HashCache optimizes *invocation latency* and *resource utilization*. In this study, we define invocation latency as the processing time of each function invocation – an interval between the start time when an invocation is received by a serverless platform and the finish time when the final output is returned. Resource utilization, on the other hand, represents the resource costs a serverless computing provider incurs to execute functions, which is consisted of CPU and memory utilization. In our experiments, we glean CPU and memory utilization data at a frequency of one measurement per second by reading the `/proc/stat` file and using the `free` command, respectively.

Application Suites: To comprehensively evaluate HashCache, we test three application suites: FaaSWorkflow, TrainTicket [24], and HotelReservation [25]. Table II tabulates the application in each suite.

1) *FaaSWorkflow:* We exploit the FaaS workflow that has three applications implemented in Python - `DForge`, `Prediction` and `SetCompute`, among which `Prediction` comes from ServerlessBench [2], while `SetCompute` and `DForge` are derived from our own construction.
2) *TrainTicket:* We select seven representative Java application workflows from the serverless TrainTicket suite [24]. All the seven workflows are directly interactive with the

[4]The commit ID is cf36299d5ee45aa014ec84326d3a69f5b2df446c.

TABLE II
APPLICATION SUITES USED IN THE EVALUATION

FaaSFlow - 3 real-world FaaS applications	
DForge	Make classification decisions (4 functions)
Prediction [2]	Resize, predict, render images (3 functions)
SetCompute	Compute, find, check sets (3 functions)
TrainTicket - 7 open-source FaaS applications	
GetContact	Get user-specific contacts (1 functions)
PayOrd	Pay an order (3 functions)
PresvTickt	Preserve a ticket (31 functions)
QueryOrd	Get user-specifc orders (2 functions)
GetTickts	Get trip-specific tickets (17 functions)
CalRef	Calculate the ticket refund (2 functions)
CancOrd	Cancel an order (4 functions)
HotelReservation - 4 open-source FaaS applications	
Reco	Get recommended hotels (3 functions)
Reserve	Reserve a hotel room (2 functions)
Search	Search hotels around (5 functions)
User	Obtain user information (1 function)

user and are invoked directly by the user in the TrainTicket application.

3) *HotelReservation:* This application suite is originally come from the DeathStarBench microservices suite [25], where we develop and migrate microservice components as the Python serverless functions.

Application Workloads. In the *FaaSWorkflow* case, synthetic data based on the normal distribution play a major role due to the lack of sizable input datasets. Specifically, for `DForge`, we generate random matrices of varying dimensions as the input data. For `Prediction`, we use an input data set consisting of 100 images, and we randomly select one for each run. Meanwhile, for the `SetCompute` workflow, we generate synthetic data by providing a random set S and a number K. Given the *TrainTicket* application suite, we apply TrainTicketAutoQuery [35] as a prototype and adopt real-world dateset [36] to generate large-scale input data as a high-quality input set. In *HotelReservation*, we use real-world data from the web repository [37]. In real-world environments, a serverless computing application typically consist of a small number of functions. An analysis on logs collected from the Azure cloud platform shows that approximately 54% of applications have only one function, 95% of applications have at most 10 functions, and less than 0.04% of applications have more than 100 functions [32]. Therefore, we deem the scale of serverless applications used in the current experimental environment to be representative. Furthermore, as in previous studies of serverless systems [23], [38], [39], we model requests' inter-arrival time using a Poisson distribution. In addition, we adopt Locust [40] as a load generator, and we set different levels of load for different applications. Specifically, the requests per second (*RPS*) for TrainTicket and HotelReservation are set to 23, while the *RPS* for FaaSWorkflow is set to 46.

B. Evaluating Invocation Latency

Now, we measure the invocation latency of applications, from the time when an invocation is received by a serverless platform to when the output is returned. Using this invocation latency, we calculate the normalized latency of HashCache and FaaSCache with respect to the OpenWhisk baseline.

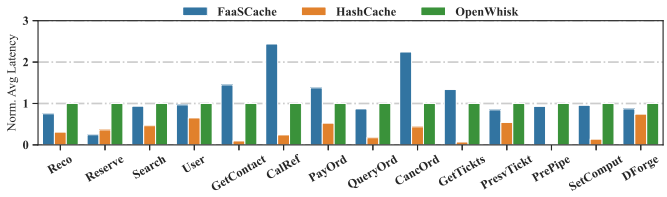


Fig. 10. Normalized average latency of all applications across three approaches.

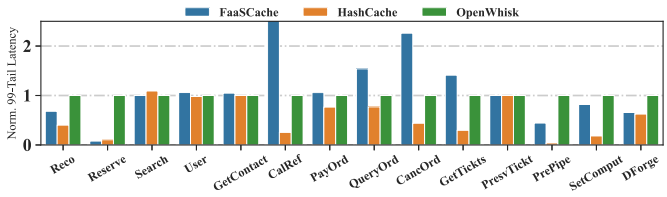


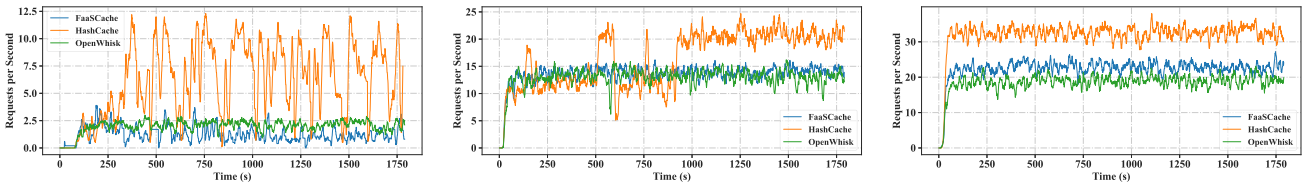
Fig. 11. Normalized 99-tail latency of all applications across three approaches.

1) Average Latency: Fig. 10 depicts the normalized average invocation latency of different applications across the three approaches. The results demonstrate that HashCache shortens the average invocation latency in FaaSCache by a window between 14.72% and 97.79%. In addition, HashCache curtails the average invocation latency of the OpenWhisk's actions by a remarkable ratio anywhere between 25.82% and 97.95%.

As a special case, we reckon that the average performance of HashCache in `Reserve` application is somewhat worse than FaaSCache because `Reserve` function has only one dependent function – and both functions always pair up to changing state when executed, which causes state update overhead in State Bridge, as discussed in Section IV-E for the overhead of the HashCache component. A similar application in the TrainTicket suite, `PresvTickt`, is apparently not affected by the state update overhead. The reason for this trend is that the application contains 31 functions, the vast majority of which are functions whose states are subject to less frequent updates. Therefore, the overhead incurred by state updates is negated by the optimization of the HashCache cache function computation results.

2) Tail Latency: Fig. 11 shows the normalized 99-tail invocation latency of HashCache, FaaSCache and OpenWhisk. Please note that OpenWhisk is tested as a baseline to normalize all 99-tail latencies. The findings unveil that HashCache takes the championship in the majority of applications. Compared with FaaSCache and OpenWhisk handling the functions, for example, our HashCache achieves a maximum latency reduction of 91.37% and 95.96% – in the case of `CalRef` and `PrePipe` applications – with an average reduction rate of 36.56% and 43.27%, respectively. In addition, HashCache delivers considerable improvements in the other applications. For instance, compared to FaaSCache and OpenWhisk, HashCache's enhancement ratios are 4.93% to 79.09% and 23.08% to 74.68% in the `Reco`, `PayOrd`, `QueryOrd`, `CancOrd`, `GetTickts`, `SetComput`, and `DForge` applications.

For the other applications, HashCache maintains the same tail invocation latency as the other techniques. The reason is



(a) TrainTicket (b) HotelReservation (c) FaaSWorkflow

Fig. 12. Throughput of different investigation applications.

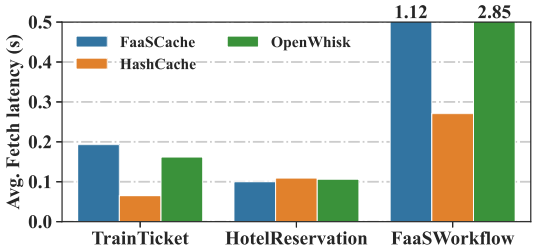
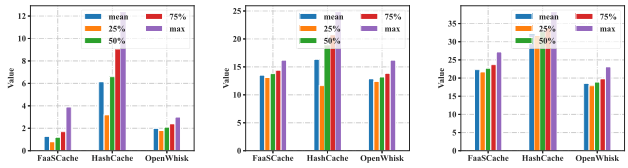


Fig. 13. Latency of fetching states.



(a) TrainTicket (b) HotelReservation (c) FaaSWorkflow

Fig. 14. Quantitative throughput for different applications.

twofold: (1) HashCache executes functions normally while reasoning whether the functions' results are cacheable; (2) when the states on which the functions depend are updated, HashCache reclaims the corresponding cache data in the action mapper – and performs step (1).

3) Fetching Latency: Typically, function execution entails an initial container start-up, incurring a start-up cost, followed by the function's computational process, which adds computational overhead. However, dissecting the container start-up and computational overhead attributed to HashCache is not meaningful. This is because HashCache enhances function performance by eliminating duplicate computations. Based on this principle, duplicate invocations in HashCache neither trigger container start-ups nor initiate computational processes. Furthermore, for those function invocations whose output results are uncached (possibly due to state changes they depend on), the execution process of HashCache is identical to that of FaaSCache or OpenWhisk, save for fetching the external state. This fetch process can be optimized by the state bridge module. In this part, we compare the fetching states overhead of HashCache with other strategies.

Fig. 13 depicts the overhead of external state fetching for different application suites under three strategies. Compared with FaaSCache and OpenWhisk, HashCache optimized the time overhead of fetching external states in the TrainTicket suite by 66.28% and 59.8% respectively. When it comes to FaaSWorkflow application, the optimizations go up to 75.91% and 90.49%. In HotelReservation application, all three strategies delivered similar latency for fetching external states. The reason is that the function states that the HotelReservation application relies on are deployed in the local cluster, thus the serverless functions can obtain the dependent states without requiring access to third-party external storage. As a result, the state bridge module of HashCache did not produce significant optimization. Considering the preceding analysis on latency, the optimization of the HotelReservation application by HashCache primarily lies in the caching of computational results.

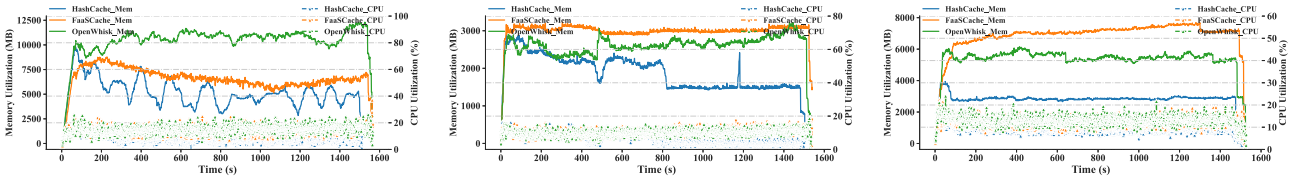
4) Throughput: Now, we are in the position of evaluating throughput across different approaches. We define throughput as the number of requests per second processed by the serverless platform. Fig. 12 illustrates the throughput of each application suite for the three approaches. From the figure, we observe that HashCache can achieve higher throughput during the execution of applications. Specifically, HashCache boosts the throughput of the three application suites by up to 3.08× to 4.83× compared to the other two strategies.

Looking further at the throughput rates, Fig. 14 summarizes the quantile values of effective throughput for different investigation cases, and the experimental results plotted in the figure depict the efficiency of HashCache in terms of throughput. On average across application suites, compared with FaaSCache and OpenWhisk, HashCache improves the throughput by 3.08× to 4.81×, 1.21× to 1.27× and 1.43× to 1.72× with respect of TrainTicket, HotelReservation and FaaSWorkflow application suites.

We also observed fluctuations in the TrainTicket application, as well as similar results during the pre-execution period of HotelReservation. This is related to the state change of the maintenance function of the State Bridge module. We will discuss this in more detail in Section IV-C2, within the context of resource utilization.

C. Evaluating Resource Utilization

Resource utilization is an integral factor affecting serverless computing services. Serverless computing providers ramp up service throughput by lowering resource overhead required to process the same request. For this reason, we gauge the host



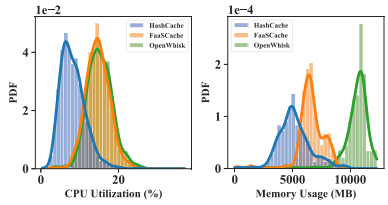
(a) TrainTicket (b) HotelReservation (c) FaaSWorkflow

Fig. 15. Resource utilization with different approaches across different investigation cases.

CPU utilization and memory usage of under different application suites. We combine the host resource utilization of distributed systems. Specifically, CPU utilization is the average of distributed host CPU utilization, whereas memory usage is aggregated from multiple host memory utilization.

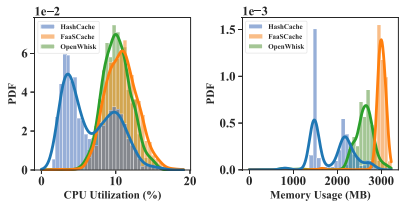
1) Average Utilization: Fig. 15, in which the CPU utilization and memory usage of HashCache, FaaSCache and OpenWhisk are plotted as a function of the execution time of the three application suites, unveils that HashCache outperforms the other two strategies in light of both CPU utilization and memory usage. For example, HashCache cuts back the average CPU utilization and the memory usage of FaaSCache and OpenWhisk by 7.28%, 6.94%, and 31.62%, 35.51%, respectively. These dramatic improvements are expected: HashCache leverages the action mapper module to cache the computing output of functions, thus avoiding the duplicate computation. By working with the function states maintained in the state bridge, the action mapper is slated to yield accurate computation results. Thus, with the cache data maintained in the action mapper, HashCache eliminates the requirement to launch new containers for executing the function invocations. Consequently, the executions of computational actions are bypassed to curb CPU overhead. Moreover, staying away from duplicate executions lowers the need for a platform to maintain the corresponding containers, which in turn mitigates memory usage on serverless platforms. And the memory space conserved in HashCache can be utilized by other actions to boost service throughput as evidenced by the experimental results in Section IV-B4.

2) Correlation With Throughput: Combining the results from Figs. 12 and 15, we sense a strong correlation between resource utilization and throughput metrics. Take TrainTicket application suite for example, peaks in the performance of HashCache are observed in Fig. 12(a), which corresponds to troughs in Fig. 15(a). These troughs are the root cause of the peaks, as the storage of function computation results on the action mapper effectively increased throughput. Recall that cached results in the action mapper are reclaimed only upon the updates of corresponding function states. For example, the `CancOrd` application modifies the order state, and the `QueryOrd` application is in need to launch some new containers to acquire the latest order information. Therefore, the troughs in Fig. 12(a) (or peaks in Fig. 15(a)) are introduced by the status updates. In this scenario, HashCache deploys new containers to execute the function invocations, thus supplying up-to-date computation results to users. To summarize, despite the fact that HashCache originates new containers in extenuating scenarios, the memory resources consumed by HashCache are on par with those of FaaSCache – a technique that is adroit at optimizing container utilization.



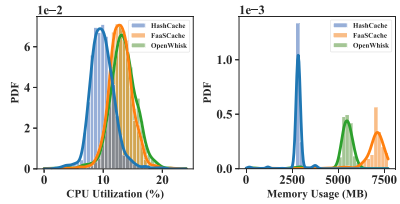
(a) (b)

Fig. 16. PDF of TrainTicket.



(a) (b)

Fig. 17. PDF of HotelReservation.



(a) (b)

Fig. 18. PDF of FaaSWorkflow.

3) Resource Utilization Distribution: To unveil the resource utilization more clearly, we investigate the distribution of the resource utilization in the cases of HashCache, FaaSCache, and OpenWhisk. Figs. 16, 17, and 18 show the probability density function or PDF of the resource utilization of the three systems executing the different application suites. The results demonstrate that HashCache exhibits significantly lower resource utilization than those of FaaSCache and OpenWhisk, as evidenced by its distributed resource utilization at much lower levels.

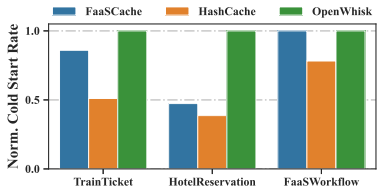


Fig. 19. Normalized cold start rate.

CPU Utilization: The PDFs plotted in Figs. 16(a), 17(a), and 18(a) reveal that the action mapper and state bridge modules, despite introducing additional operations to store computation results and maintain function status, bring forth impressive optimization benefits to the overall serverless computing system. Specifically, the low CPU distribution achieved by HashCache is attributed to the function calculation results stored by the action mapper, which effectively averts container startup and duplicate invocations. Thus, we conclude that these additional overheads are offset by the gains in the overall performance of the serverless computing system.

Memory Utilization: By avoiding the duplicate computation, HashCache is able to drastically mitigate invocation latency at a negligible cost of CPU overhead. Figs. 16(b), 17(b), and 18(b) illustrate the PDFs of the memory utilization across wide ranged application suites. We observe from the results that FaaSCache and OpenWhisk yield higher memory usage than HashCache. On average across application suites, compared with FaaSCache and OpenWhisk, HashCache improves the memory utilization by 1.57× and 1.58×.

In brief, powered by modules the action mapper and state bridge, HashCache cuts back invocation latency without introducing additional overheads. HashCache also fends off container start-ups and alleviates high memory usage of serverless platforms by dodging duplicate function executions.

4) Cold Starts: The primary objective of HashCache is to mitigate duplicate computations by caching the computing results of serverless functions. Avoiding duplicate computations eliminates the necessity for functions to initiate new container and execute computational tasks. As a result, serverless computing platforms can effectively avoid cold starts due to this feature of HashCache.

Now we evaluate the cold start introduced by the three approaches. A cold start rate is defined by dividing the number of cold starts and the number of function invocations. To achieve fair comparisons, we conduct the same workload on three different strategies and normalize the experimental results of different strategies to the level of OpenWhisk. Fig. 19 shows the normalized cold start rates generated by the three policies when executing the different application suites. As a tactic to eliminate duplicate execution, HashCache proactively averts cold starts: (i) HashCache significantly reducing the need for container cold starts by caching computational results, and (ii) HashCache initiates a container only when the function's dependency state changes, and the real-world function state read skew (see Section II-B) helps reduce the probability of starting the container unnecessarily.

TABLE III
AVERAGE CACHE HIT RATE IN HASHCACHE

Application	TrainTicket	HotelReservation	FaaSWorkflow
Avg. hit rate	84.3%	85.9%	66%

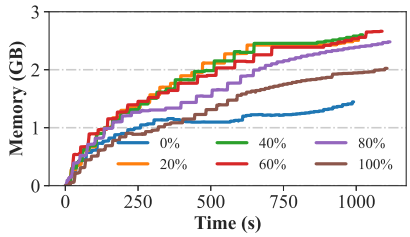


Fig. 20. Memory overhead of Action Mapper with different cache coverage.

Although caching computation results incurs a certain memory overhead as detailed in Section IV-E, the overhead is justified by dodging duplicate computing costs.

5) Cache Hits: Table III summarizes the average cache hit rate of the evaluation application suites. HashCache utilizes LRU to schedule the cached outputs. Table III shows that HashCache has a high hit rate for all three different application suites, which also confirms that serverless functions produce a large amount of duplicate computation during execution.

D. Impacts of Hyperparameters

In this section, we delve into the impact of the cache coverage of LRU on HashCache. To measure the overhead more accurately in extreme cases, we synthesize the bulk input parameters that obey a normal distribution.

Fig. 20 depicts the impact on the action mapper of running 10,000 invocations with various cache coverage – from 0% to 100% – for a computational function with a total input and output size of 500 MB. Since we are concerned with the memory overhead of our Action Mapper module, we use a sub-millisecond function to overlap the effects of function execution. Let the size of the total parameters be 100 MB, and a coverage of 20% means that the cache size of LRU is 20 MB. Since the memory usage of Action Mapper is directly related to the amount of data cached, we believe that the experiment can be extended to a large-scale serverless computing framework. In this experiment, we analyze the memory overhead introduced by the action mapper orchestrated by the HashCache scheme.

The 0% coverage scenario becomes the best case as there is no extra memory for caching computing results. Additionally, the probability of a cold start is extremely low because we have thousands of invocations for a single function. Furthermore, without the overhead of in-memory access to cached results and the time overhead of container cold starts, this case leads to the fastest completion time compared to other LRU coverage scenarios. When it comes to other coverages, intriguingly, we observe that the memory strain on HashCache decreases as the cache coverage rises. This pattern is caused by the fact that our HashCache avoids duplicate computation. Specifically, OpenWhisk creates corresponding class instances to manage started

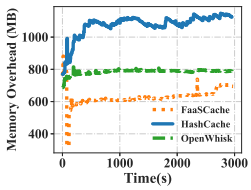


Fig. 21. Invocation controller overhead.

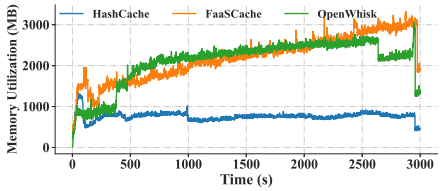


Fig. 22. Memory utilization.

containers (e.g., container objects). Our HashCache wards off starting containers by the virtue of cached computation results: upon receiving duplicate invocations, launching extra containers becomes unnecessary. Furthermore, with OpenWhisk programming language level garbage collection, previously created class instances (while the HashCache is not already serving its purpose) are released. Thus, as the coverage rises, the number of surplus containers dodged by HashCache goes up – and the memory usage for maintaining class instances drops.

E. Overheads of HashCache Components

The *behavior monitor* module distinguishes stateful functions by comparing the changes in the `/proc/net/tcp` file before and after function execution. Since a container in the serverless computing platform serves only invocations for the same function, the number of entries in the `/proc/net/tcp` file is significantly reduced. Consequently, the monitoring overhead of comparing file changes can be considered negligible. Therefore, the major runtime overheads in HashCache are imposed by the *action mapper* and *state bridge* modules. Let us analyze such overhead in this subsection.

1) Action Mapper: The action mapper utilizes host OS memory to cache computing results managed by the LRU strategy. In this experiment, we quantify the LRU impact on the action mapper that performs 15,000 invocations for twenty computational functions with a total input and output size of 2 GB. Figs. 21 and 22 show the caching overhead and resource benefits – memory savings – of the action mapper.

Memory Overhead of Action Mapper: We implement the action mapper module in OpenWhisk's invocation controller module, which is used to schedule function invocations into containers. Therefore, comparing the memory resource utilization generated by the invocation controller gives a sound estimate of the memory overhead used by the action mapper to cache the results of function calculations. Fig. 21 illustrates the memory overhead while running the action mapper module. The total memory usage of the HashCache is on average 448.08 MB and 290.48 MB more than that of FaaSCache and OpenWhisk, respectively. These extra memory usages offer immense benefits to serverless systems.

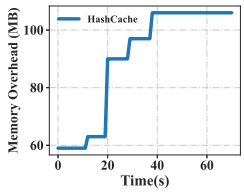


Fig. 23. Blob Cache memory overhead.

Memory Saved by Action Mapper: Fig. 22 depicts the host memory usage of the three approaches during the invocation execution. The average memory utilization of HashCache, FaaSCache and OpenWhisk are 973.11 MB, 1985.79 MB and 1936.36 MB, respectively. Although the caching mechanism introduces additional memory overhead, the memory overhead is well offset by the performance gains from reducing the number of duplicate executions.

2) State Bridge: The state bridge maintains (i) the function-state relationship in memory; (ii) the latest states (blobs) in the local file system, indexing the states by hashing the requested URL. For instance, state bridge caches demand image objects of different sizes of an image fetching function (`image` function) in a local file system as a way of accelerating the invocation process. Fig. 23 illustrates the memory overhead of the `image` function of the state bridge service on five consecutive requests for each of the 10 file sizes – 1 KB, 16 KB, 32 KB, 128 KB, 256 KB, 512 KB, 1024 KB, 2048 KB, 3072 KB and 4096 KB. The total size of the required files is 55 MB. The findings confirm that less than 110 MB is required to run all the 50 invocations. The state bridge itself consumes up 60 MB of memory – and the resources required to forward files takes up 50 MB.

V. Related Work

Fine-grained function computations impose a performance barrier on serverless computing, where a growing number of schemes were proposed to tackle this problem by curtailing container startup overhead [17], [41]. Such an optimization, however, only alleviates the impact of cold starts, with the affliction remaining in serverless computing. Very recently, researchers adopted VM-based sandbox approaches that provide a strong isolation among functions to mitigate the cold start problem [10], [42], [43]. Though the strong isolation offer stable operations among functions, this solution causes additional start-up costs.

Interestingly, special attention was paid to leveraging caching strategies to cope with the cold start issue. For example, Du et al. proposed the Init-less approach that aims to boost startup of diverse serverless applications [18]. Init-less caches most metadata amid function initialization to reduce the amount of data initialized when the same function is invoked. Ana Klimovic et al. designed Pocket to enable elastic and distributed data storage when exchanging intermediate data during function invocations [44]. Wang et al. built

a caching system, referred to as *InfiniCache*, to facilitate an in-memory object caching service [45]. Francisco et al. present Faa$T - an in-memory caching layer to cache remote I/O requests into local cluster [29]. Mvondo et al. cached intermediate data among functions by recycling wasted memory resource overestimated by FaaS users [19]. Ustiugov et al. devised a FaaS orchestrator called *REAP*, which makes use of a page fault inspection technique and a prefetching mechanism to speed up function invocations [46]. Shillaker et al. implemented a serverless runtime - Faasm - to extend traditional WebAssembly modules, expecting to execute functions across distributed clusters [34]. Alexander et al. developed FaaSCache keeping function instances alive as analogous to caching objects, aiming to accelerate function invocations in serverless platforms [20]. On the other hand, HashCache accelerates serverless computing platforms by caching the output results of computational and stateful functions based on duplicate function executions and infrequent updates to external states. Compared to FaaSCache, HashCache significantly reduces duplicate computations while ensuring the correctness of function results, thereby reducing the number of containers required for executing functions. In short, HashCache reduces invocation latency and improves memory utilization by caching the computed results of functions.

The other existing solutions are mainly focused on lightweight virtualization technologies to lower container startup overhead – or to alleviate the cold start problem in FaaS platforms. Unlike the aforementioned approaches, our HashCache trims the overhead of function invocation at the fine granular-level: we cache the output of computational and stateful functions with an ambition to cut back the invocation latency between users and the FaaS platforms. HashCache maintains the external states of stateful functions, thereby bolstering caching performance – and facilitating fast access to remote objects. Furthermore, the FaaS platforms, equipped with HashCache, rule out duplicate function execution accompanied by low resource utilization.

VI. Conclusion

In this study, we proposed HashCache - a holistic caching system - to fend off duplicate execution in the arena of serverless computing. HashCache integrally orchestrates two key extensions – the action mapper and state bridge modules, which are conducive to caching data reused by serverless functions. The main responsibility of the action mapper is to cache computing results of computational and stateful functions, thus averting duplicate function execution to speed up performance. The state bridge, on the other hand, maintains the remote states requested by stateful, assisting the action mapper to cache results leading to shortened invocation latency. The experimental results confirm that HashCache is slated to eradicate cold starts, to curtail invocation latency, and to lower resource overhead on serverless platforms.

We implemented HashCache on top of OpenWhisk to validate the effectiveness of HashCache. We conducted the extensive experiments to compare the performance of our HashCache against the state-of-the-art solutions FaaSCache and OpenWhisk in terms of invocation latency and resource utilization. The experimental results unveil that HashCache, which governs a wide range of serverless workloads, immensely cuts back the invocation latency and optimizes the resource utilization of the leading-edge systems by 14.72%–95.16% and 6.94%–35.51%, respectively.

References

[1] T. J. Skluzacek et al., "A serverless framework for distributed bulk metadata extraction," in *Proc. 30th Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2021, pp. 7–18.

[2] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 30–44.

[3] 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.

[4] "AWS lambda," Accessed: Jul. 15, 2021. [Online]. Available: https://aws.amazon.com/lambda/

[5] "Google cloud function," Accessed: Jul. 15, 2021. [Online]. Available: https://cloud.google.com/functions/

[6] "Azure function," Accessed: Jul. 15, 2021. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[7] "Openwhisk," Accessed: Jul. 15, 2021. [Online]. Available: https://openwhisk.apache.org/

[8] "Openfaas," Accessed: Jul. 15, 2021. [Online]. Available: https://www.openfaas.com/

[9] "Docker Container," Accessed: Jul. 15, 2021. [Online]. Available: https://www.docker.com/

[10] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Networked Syst. Des. Implementation*, 2020, pp. 419–434.

[11] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 691–707.

[12] V. Sreekanti et al., "Cloudburst: Stateful functions-as-a-service," in *Proc. VLDB Endow*, vol. 13, no. 12, pp. 2438–2452, Jul. 2020, doi: 10.14778/3407790.3407836.

[13] M. Arif, K. Assogba, and M. M. Rafique, "Canary: Fault-tolerant FaaS for stateful time-sensitive applications," in *Proc. 34th Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 568–583.

[14] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 1187–1204.

[15] F. Yamaguchi and H. Nishi, "Hardware-based hash functions for network applications," in *Proc. IEEE 19th Int. Conf. Netw.*, 2013, pp. 1–6.

[16] D. K. Kim and H.-G. Roh, "Scheduling containers rather than functions for function-as-a-service," in *Proc. 21st Int. Symp. Cluster Cloud Internet Comput.*, 2021, pp. 465–474.

[17] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 57—70.

[18] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 467–481.

[19] D. Mvondo et al., "OFC: An opportunistic caching system for faaS platforms," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 228–244.

[20] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 386–400.

[21] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *Proc. 12th Int. Conf. Cloud Comput.*, 2019, pp. 502–504.

[22] S. Kotni et al., "Faastlane: Accelerating function-as-a-service workflows," in *Proc. 2021 USENIX Annu. Tech. Conf.*, 2021, pp. 805–820.

[23] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. IEEE/ACM 52nd Annu. Int. Symp. Microarchitecture*, 2019, pp. 1063–1075.

[24] Fudan SE Lab, "Serverless train ticket," 2020. [Online]. Available: https://github.com/FudanSELab/serverless-trainticket

[25] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.

[26] Google, "Firebase," 2020, [Online]. Available: https://github.com/firebase/functions-samples
[27] "Using AWS Lambda with Amazon S3," Accessed: Nov. 17, 2022. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html
[28] "Azure blob storage," Accessed: Nov. 17, 2022. [Online]. Available: https://azure.microsoft.com/en-us/products/storage/blobs/
[29] F. Romero et al., "Faat: A transparent auto-scaling cache for serverless applications," in *Proc. 12th ACM Symp. Cloud Comput.*, 2021, pp. 122–137.
[30] Z. Wu, Y. Deng, H. Feng, Y. Zhou, G. Min, and Z. Zhang, "Blender: A container placement strategy by leveraging Zipf-like distribution within containerized data centers," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 2, pp. 1382–1398, Jun. 2022.
[31] A. Mahgoub et al., "WISEFUSE: Workload characterization and DAG transformation for serverless workflows," in *Proc. ACM Meas. Anal. Comput. System*, vol. 6, no. 2, Jun. 2022, Art. no. 26.
[32] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.
[33] A. Khandelwal et al., "Jiffy: Elastic far-memory for stateful serverless analytics," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 697–713.
[34] S. Shillaker and P. Pietzuch, "FAASM: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 419–433.
[35] Fudan SE Lab, "Serverless train ticket auto query," 2020. [Online]. Available: https://github.com/FudanSELab/train-ticket-auto-query
[36] "Bureau of transportation statistics," Accessed: Jul. 15, 2023. [Online]. Available: https://www.bts.gov/content/air-passenger-travel-departures-united-states-selected-foreign-countries-thousands
[37] M. Jesse, "Hotel booking demand," 2019, [Online]. Available: https://www.kaggle.com/datasets/jessemostipak/hotel-booking-demand
[38] 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.
[39] 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.
[40] B. Carl et al., "Locust: Hypothesis testing with Python," 2022. [Online]. Available: https://locust.io/
[41] A. Mohan et al., "Agile cold starts for scalable serverless," in *Proc. 11th USENIX Workshop Hot Topics Cloud Comput.*, 2019, pp. 1–6.
[42] "Hyper - make VM run like container," Accessed: Jul., 13, 2021. [Online]. Available: https://hypercontainer.io/
[43] "Google gVisor: Container runtime sandbox," Accessed: Nov. 13, 2021. [Online]. Available: https://github.com/google/gvisor/
[44] A. Klimovic et al., "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 427–444.
[45] A. Wang et al., "InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 267–281.
[46] D. Ustiugov et al., "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 559–572.

Zhaorui Wu received the MS degree in computer architecture from the Computer Science Department, Jinan University. He is currently working toward the PhD degree with Computer Science Department, Jinan University, China. His current research interests include parallel and distributed computing, computer architecture, and performance evaluation, etc.

Yuhui Deng received the PhD degree in computer science 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 with EMC Corporation as a senior research scientist from 2008 to 2009. He worked as a research officer with Cranfield University, in United Kingdom from 2005 to 2008. He was on the finallist of storage challenge in ACM/IEEE SC2007, and on the finallist 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, cloud computing, and information storage, etc.

Yi Zhou received the PhD degree in computer science from Auburn University, in 2018. He is currently an assistant professor with the TSYS School of Computer Science, Columbus State University. Prior to joining Columbus State University, in 2018, he has been a software engineer with Alcatel-Lucent Technologies (China) Company, Ltd. for four years from 2010 to 2014. His research interests include energy-saving techniques, database systems, and Big Data techniques and parallel computing.

Lin Cui received the PhD degree from the City University of Hong Kong, in 2013. He is currently a professor with the Department of Computer Science, Jinan University, Guangzhou, China. He has broad interests in networking and distributed systems, with focuses on software defined networking (SDN), programmable data plane, network function virtualization (NFV), congestion control, and so on.

Xiao Qin received the PhD 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, Auburn University. He won an NSF CAREER Award, in 2009. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation.