doi:10.3772/j. issn. 1006-6748. 2020. 01. 003

Exploring serverless computing for stream analytic[①]

Cheng Yingchao(成英超)[②] [*] [***], Hao Zhifeng [*] [**], Cai Ruichu [*]
([*] School of Computer Science and Technology, Guangdong University of Technology, Guangzhou 510006, P. R. China)
([**] School of Mathematics and Big Data, Foshan University, Foshan 528000, P. R. China)
([***] Department of Statistics, Texas A&M University, College Station 77840, USA)

Abstract

This work proposes ARS (FaaS) serverless framework scheduling and provisioning resources for streaming applications autonomously, which ensures real-time response on unpredictable and fluctuating streaming data. A HPC cloud platform is used as a de facto platform, on which serverless computing for stream analytic is explored. This work enables application developers to build and run steaming applications without worrying about servers, which means that the developers are able to focus on application features instead of scheduling and provisioning resources of the infrastructure. The serverless computing framework, ARS(FaaS), provides function-as-a-service to make the developers write code in discrete event-driven functions. ARS(FaaS) is capable of running and scaling the developer's code automatically, according to the throughput of streaming events. The major contribution of this serverless framework is effective and efficient autonomous resource scheduling for real-time streaming analytic, which enables the developers to build applications faster with autonomous resource scheduling. ARS (FaaS) framework is appropriate for real-time and stream analytic on event-driven data with spiky and variable compute requirements.

Key words: serverless, steam processing, HPC cloud, auto-scaling, function-as-a-service(FaaS)

0 Introduction

Serverless computing emerged as a key paradigm for processing streaming data in the cloud[1-3]. Traditional clouds (e. g., infrastructure-as-a-service) provide users with access to voluminous cloud resources, and resource elasticity is managed at the virtual machine level, which often results in overprovisioning of resources leading to increased hosting costs, or underprovisioning of resources leading to poor application performance[4], while the developing serverless computing is a compelling approach which hosts individual callable functions to provide function-as-a-service, i. e., the computation unit is a function. When a service request is received, the serverless platform allocates an ephemeral execution environment for the associated function to handle the request[5]. Serverless computing promises reduced hosting costs, high availability, fault tolerance, and dynamic elasticity through automatic provisioning and management of compute infrastructure[6]. Thus, the developers could focus on the application (business) logic, leaving the responsibilities of dynamic cloud resources managements to the provider while the cloud provider could improve the efficiency of their infrastructure resources. A more detailed evolutionary cloud computing story about serverless was illustrated by Ref. [7].

Serverless computing is also a form of utility computing. Its pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity[8]. The main commercial and open source serverless providers include AWS Lambda, Apache/IBM OpenWhisk, Google Cloud Functions, and Microsoft Azure Functions. Although serverless is a new paradigm, various serverless microservices are emerging in recent years, it is referred the reader to a comprehensive survey[1].

Case Study Numerous mobile apps have sprung up. By providing high-quality and innovative services, globally popular mobile apps, e. g., Instagram, WhatsApp, WeChat and AliPay, have been installed in billions of devices. Stable and high-volume streaming big data are generated by millions of daily active users. However, the overwhelming majority of mobile apps only have a very limited number of the users that are no

① Suported by the National Natural Science Foundation of China (No. 61472089, 61572143), NSFC-Guangdong Joint Found (No. U1501254) and China Scholarship Council (No. 201608440336).

② To whom correspondence should be addressed. E-mail: zfhao@fosu.edu.cn

Received on Dec. 19, 2018

larger than ten thousand. What's more, the user activity distribution is irregular. Thus, the non-mainstream mobile apps are dealing with couples of and/or thousands of small tasks (steams). They are stateless, short run times, and agile[6,9]. Most of the developers rent a certain amount of cloud resources for the non-mainstream applications. The amount of resources have to handle the peeks of user activity. However, these resources are idle for most of the time. To improve the resource efficiency, many developers change the resource configuration manually when necessary. With serverless computing, numerous non-mainstream mobile apps could achieve autonomous resource scheduling, it handles everything required to run and scale mobile apps with high availability.

In serverless computing, the application logic is composed of functions and executed in response to events. The events can be triggered from sources external to the cloud platform but also commonly occur internally between the cloud platform's service offerings, allowing developers to easily compose applications distributed across many services within a cloud[10]. Serverless computing is event-driven, where applications are defined by actions as well as events that trigger them. This character makes it very appropriate for stream analytic. Event-driven stream processing is one of the important research issues among the data stream researchers and has many applications[11-14]. Data stream management systems are creating extremely high-(volume, velocity, variety and value) streams to continuously process terabytes of data per hour from hundreds of thousands of sources such as IT logs, financial transactions, website click streams, social media feeds, and location-tracking events.

The remainder of the paper is organized as follows. Section 1 introduces the challenges of serverless computing. Section 2 gives the ARS(FaaS) serverless framework in detail. Section 3 describes the serverless stream analytic and the framework design on HPC cloud. Section 4 contains the empirical studies on real data. Finally, this paper concludes the work in future in Section 5.

1 Challenges on serverless

Even though serverless is perfect for event-driven stream processing, it is limited to short-running, stateless simple applications. It is good for micro-services, mobile backend, IoT and modest stream processing, but not well-suited for more complex services, e.g., deep learning training, Spark/Hadoop analytic, heavy-duty stream analytic and video streaming, especially when the application logic follows an execution path spanning multiple functions[5,15]. Consider a workflow of logo detection from a video stream, which executes five consecutive functions: split video into frames, extract features from each frame, measure L2 distance between the extracted features and pre-generated logo features, out-put matching pairs, and aggregate the matching pairs. This workflow is run using AWS Lambda[9], IBM Functions[10], Google Cloud Functions[11], and Azure Functions[12], all of which integrate multiple functions into a single service through different methods. On these commercial offerings, it is observed (through Fig. 1) that the total runtime is obviously longer than the execution time of application logic functions, i.e., the execution of such connected functions introduces considerable runtime overheads. Another observation is that existing serverless platforms do not support long-term function execution. To overcome it, the developer may choose making one function to invoke another to continue the process. However, the second and subsequent functions would need to start new containers, which increases overall overhead.

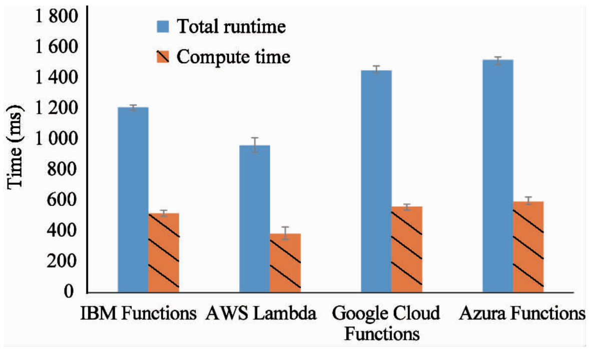

Fig. 1 Total runtime and compute time of the workflow of logo detection from a video stream. Results show the mean values with 95% confidence interval over 10 runs, after discarding the initial (cold) execution

Through expanding serverless to general stream analytic, it is found that this relatively new technology still has certain challenges.

The first challenge is cold start. The practical advantage of serverless is its ability of auto-scaling such that developers could reduce cloud costs and development costs. Meanwhile, cloud providers could improve resource management. However, existing serverless offerings usually execute each function within a separate container instance, which inevitably leads to increased invocation latency as a result of cold starts. When the request is received, the platform initializes a new, cold container instance to execute the associated function, and terminate the instance when the execution finishes.

Thus, it incurs long invocation latency for each request. Take the JVM-implemented function as an example, it could cost up to ten seconds to run the first invocation occasionally.

The second challenge is execution time and resource restrictions. In stream processing, traditional data stream management systems are best equipped to run one-time queries over finite stored data sets. However, the fluctuating streaming big data require not only one-time queries but also continuous queries over high-volume unbounded data in real-time. Most existing serverless offerings do not support continuous computation longer than 5/10 minutes. Serverless functions are limited in their execution time, but streaming applications may need more than that of short-running functions. In many scenarios, e. g. , financial or emergency applications, it is necessary to apply long-running logic. Besides function execution time limit, enforceable resource restrictions on a serverless function include memory, CPU usage and bandwidth. For instance, the maximum execution memory per invocation is 3 008 MB for AWS Lambda and 512 MB for IBM Functions. These resource restrictions are needed to make sure that the platform could deal with spikes, and withstand attacks. Additionally, there are aggregate resource restrictions that can be applied across a number of functions or across the entire platform[1].

To overcome the above challenges, a novel, high performance serverless framework, ARS(FaaS), is designed and prototyped, that is running on a HPC cloud platform. ARS (FaaS) enables self-characteristics (e. g. , self-optimizing and self-configuring) on HPC cloud, which promises increased elasticity and efficiency for scheduling, provisioning resources autonomously on demand. The main contributions of this work include: 1) The applicability of function-as-a-service and high performance serverless architecture are explored for general stream analytics. 2) A prototype that combines Tianhe-2 supercomputer[16] with Apache Storm and OpenWhisk is implemented. 3) Experiments with four streaming applications show the good scalability and low overheads of ARS(FaaS). 4) Performance analysis of ARS(FaaS) and state-of-the-art work (e. g. , AWS Lambda).

2 Serverless for streaming data

Serverless workloads today are very lived shorty, but serverless gives the illusion of unlimited resources to handle long running compute tasks [6]. To explore serverless for longer running tasks, one of the different facets of serverless is tried to address: streaming data. ARS(FaaS) serverless framework is designed to solve the scheduling issues by providing developers an efficient way of approaching general streaming applications. It tries to exclude the complexity of handling the resource scheduling work at all levels of the technology stack. The ARS(FaaS) framework is implemented with Apache Storm linked to Apache OpenWhisk which is responsible for basic process of the streaming data. The output of Storm is sent to a data stream and processed by an OpenWhisk function.

2.1 Pre-warm container

In ARS(FaaS) framework, the cloud operator is used to manage system resources so as to run the functions supplied by application developers. Specifically, ARS(FaaS) framework uses containers to handle this work, mapping each function into its own container. This mapping makes the function code portable. Thus, the operator is able to execute the function as long as there are enough resources in the HPC cloud. The containers are also able to isolate most of the faulty code execution by providing virtually isolated environments with name spaces which separate HPC cloud resources, e. g. , processors, networking.

When the ARS(FaaS) framework starts new containers to deal with the incoming requests, the initialization of the container may cost some time. And the container is responsible for its associated function code and the code's execution. Therefore, the initialization can court undesired start up latency to the function execution, which is known as cold start issue. To solve this issue, the reuse strategy is used in many practices, which means that the launched container is reused to handle future requests by keeping them 'warm', i. e. , running idle. Although the first function call still faces the clod start issue, the subsequent function calls are able to reuse the launched container. Therefore, the start up latency could be optimized for processing. Even though reducing the latency, this strategy comes at a cost of resource inefficiency, i. e. , the containers occupy cloud resources unnecessarily for the idle period. The ARS (FaaS) frame-work uses pre-warm technique, by which containers are launched before the arrival of requests.

2.2 Serverless for stream

A serverless stream processing model is provided to facilitate the serverless execution of streaming applications. In this model, the key idea is the conversion function which injects the compute logic to the stream. Thus, the framework puts the conversion functions into the applications' data processing topology.

The framework controls the deployment and execution of streams along with related functions by a high-level description. The description gives the developers fine-grained control over the run time mechanisms of the framework, including QoS requirements, scaling policies, etc. The description is divided into parts, in which each part describes a respective runtime mechanism. Take an instance, the QoS part describes the QoS requirements of the framework (e. g. , maximum stream latency, minimum stream through-put); The scaling part describes the elasticity strategies, specifying the framework to adapt to the varying workloads.

3 Framework design

As cloud computing has become a dominant computing paradigm, many HPC systems with tremendous computing power have enabled cloud features. A HPC cloud based serverless computing framework is developed to explore serverless implementation considerations on stream analytic, and provide a baseline for existing work comparison. The design of this framework aims to implement FaaS on the HPC cloud platform, enabling self-characteristics (e. g. , self-optimizing and self-configuring) on HPC cloud.

Fig. 2 shows an overview of the components of ARS (FaaS) framework. As we can see, the framework is running on the Tianhe-2 supercomputer platform with a cloud resource pool service provider Kylin cloud. This HPC cloud platform is chosen because it provides tremendous and highly scalable compute power that aligns well with the goals of this work.

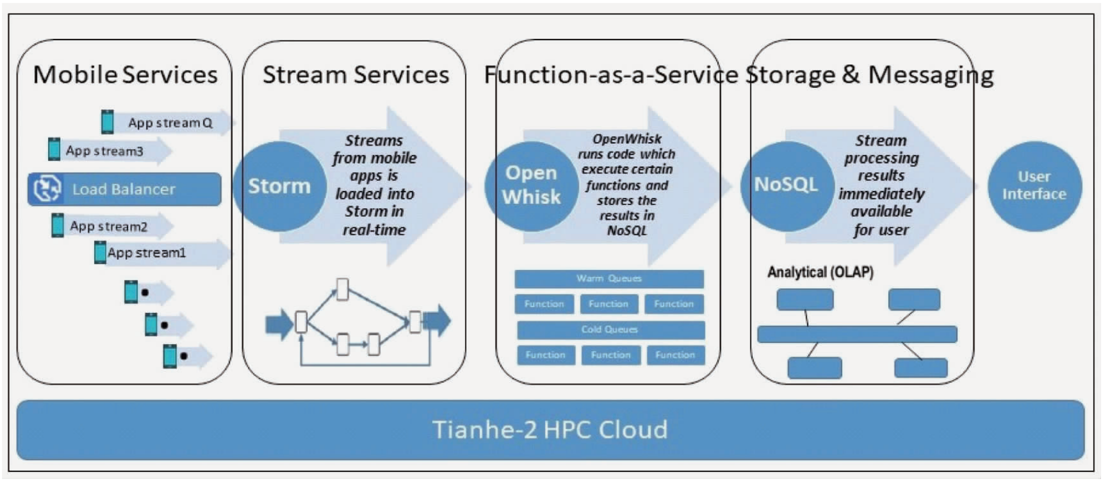

Fig. 2 Illustration of serverless computing on high – volume stream analytic

Real-time event data from numerous applications are sent to Apache Storm, which provides streaming services and allows multiple Apache OpenWhisk functions to process the same events. Function executions start from the services to receive the invocation calls by the events, and then retrieve function metadata from NoSQL. Services create the execution requests which contain function metadata and inputs. Subsequently, to process the execution requests, the available containers of the FaaS services are located by the services. An info-mechanism is designed to control the interactions between stream and FaaS service. For each function, the framework maintains a global cold queue and a warm queue. The warm queues contain information of available container, including the address of the Apache Storm worker instance and the name of the available container. The cold queues contain information indicating the workers with unallocated memory that they could start new containers. The services primarily check the warm queue of the function to find information of the available containers. If no information is found, the services check the cold queue of the function. The cold queue will send a message containing the URI to the FaaS services which will assign a new container to the function. Then the function will be executed and its outputs will be returned to the services to respond to the invocation call.

4 Empirical studies

Two sets of experiments are designed to test and demonstrate the execution performance of the framework, including function throughput and concurrency for streaming applications. Its performance is also compared with commercial offerings, including AWS Lambda, Google cloud Functions, IBM Functions, and Azure Functions. To evaluate the performance of ARS (FaaS) on stream analytics, the following streaming applications are implemented: real-time event detection[17], logo detection[18], frequent pattern detection[19] and density-based clustering for stream[20]. ARS(FaaS) framework is deployed on Tianhe-2 HPC cloud which provides IaaS. Tianhe-2 physical resources are virtualized as a virtual resource pool. Tianhe-2 HPC cloud provides users with a certain amount of resources in the form of virtual machines running Ubuntu Linux system. Each visual machine configuration could be customized. Since the serverless offerings have different resource restrictions (Table 1), five groups of experiments are executed to test the function throughput performance of the serverless offerings. The experiments in each group are executed with a same memory allocation which is the key resource configuration parameter in serverless computing.

Table 1 Feature comparison of serverless frameworks

Item	AWS Lambda	Azure Function	Google Function	IBM Function	ARS(FaaS)
Max Memory	3 008 MB	1 536 MB	2 048 MB	512 MB	128 GB
Trigger	18 triggers	6 triggers	3 triggers	3 triggers	3 triggers
Container OS	Linux	Windows NT	Debian GNU/Linux 8	Alpine Linux	Ubuntu Linux
Container CPU	2.9 GHz	1.4 GHz	2.2 GHz, 2 cores	2.1 GHz, 4 cores	2.2 GHz, 12 cores
Execution Timeout	5 min	10 min	9 min	10 min	n/a
Code Size Limit	50/250 MB	n/a	100/500 MB	48 MB	n/a
Runtime Language	5 languages	9 languages	1 language	7 languages	7 languages

4.1 Function throughput

Function throughput is an indicator of concurrent processing because it tells how many function instances are supplied to deal with extensive requests[21]. The first group of experiments are executed with a memory allocation of 512 MB. Thus, all the serverless offerings are tested and their performances are shown in Fig. 3. The second group of experiments are executed with a memory allocation of 1 536 MB. As the IBM Function has a memory restriction of 512 MB, there are four serverless frameworks tested in this group and the performance are shown in Fig. 4. In the third group, all experiments are executed with a memory allocation of 2 048 MB. IBM Function and Azure Function are not tested in this group also because of their memory restrictions. The experiment results of this group are shown in Fig. 5. For the same memory restriction, AWS Lambda and ARS(FaaS) are tested in the fourth group, with a memory allocation of 3 008 MB. The performance comparison is shown in Fig. 6. The best performance of ARS (FaaS) is also provided in Fig. 7. It causes the work to break through all the memory restrictions of serverless computing offerings and set a new record of 128 GB, it is the only serverless framework tested in the fifth group.

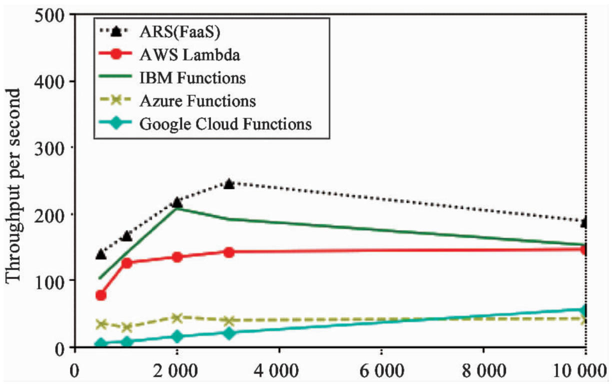

Fig. 3 The 1st experiment group of function throughput on concurrent invocations. Because of IBM Function's memory restriction, the experiments are executed with 512 MB memory allocation

Fig. 3 – Fig. 7 are showing the overview of the performance on function throughput of the serverless frameworks. In those figures, the x-axis presents the number of concurrent invocations from 500 to 10 000, and the y-axis presents the function throughput per second.

Fig. 3 shows the experiment results of the 1st group, where IBM Function (maximum memory: 512 MB) achieves its best performance (i. e., 207 throughputs per second in average) at 2 000 invocations. It's the best performance of the commercial serverless frameworks. While, with the 512 MB memory restriction, the performance of our ARS(FaaS) is better than IBM Function.

Fig. 4 shows the experiment results of the 2nd group, where Azure Function (maximum memory: 1 536 MB) achieves its maximum throughput (133) at 2 000 invocations, too. However, both AWS Lambda and our ARS(FaaS) performs better than Azure Function. What's more, all other serverless frameworks in this group have better performance than Azure Function at the level of 10 000 invocations.

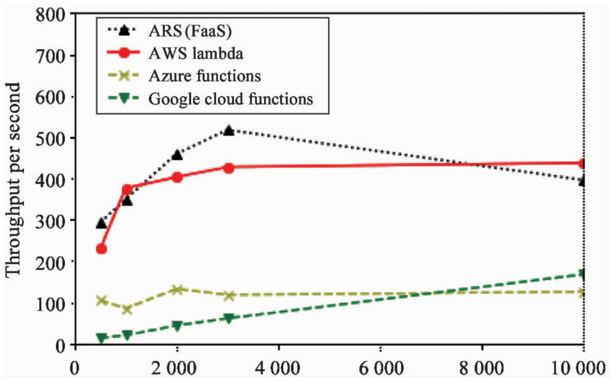

Fig. 4 The 2nd experiment group of function throughput on concurrent invocations. Because of Azure Function's memory restriction, the experiments are executed with 1 536 MB memory allocation

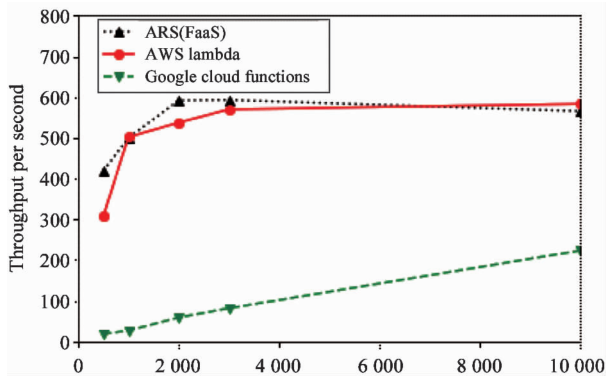

Fig. 5 The 3rd experiment group of function throughput on concurrent invocations. Because of Google Function's memory restriction, the experiments are executed with 2 048 MB memory allocation

Fig. 5 shows the experiment results of the 3rd group, where the function throughput of Google Function (maximum memory: 2 048 MB) increases continuously and reaches 223 throughputs per second at 10 000 invocations. In this group, Google Function takes the advantage of its maximum memory allocation, but still cannot compete with AWS Lambda and our ARS (FaaS).

Fig. 6 shows the experiment results of the 4th group, where AWS Lambda (maximum memory: 3 008 MB) achieves its best performance (i. e., 857 throughputs per second) at 3 000 invocations. Even though there is no front runner in this group, our ARS (FaaS) achieves the highest throughput at 2 000 invocations.

Fig. 7 shows the experiment result of the 5th group, where ARS (FaaS) (maximum memory: 128 GB) achieves its maximum throughput (about 53 000) at 100 000 invocations. It's the only serverless framework tested in this group, because the commercial serverless offerings, e. g., AWS Lambda, have performance bottlenecks (memory restrictions), while ARS (FaaS) technically could utilize the whole memory of Tianhe-2 supercomputer system (1. 408 PB). In this experiment, we are authorized to access 128 GB memory of Tianhe-2.

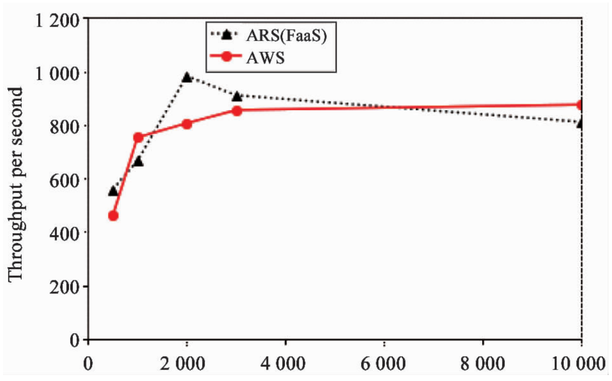

Fig. 6 The 4th experiment group of function throughput on concurrent invocations. Because of AWS Lambda's memory restriction, the experiments are executed with 3 008 MB memory allocation

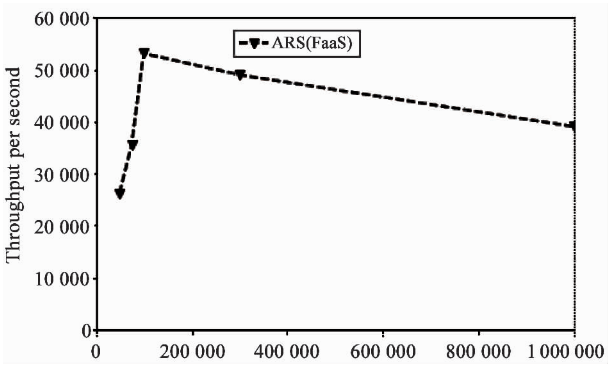

Fig. 7 The 5th experiment group of function throughput on concurrent invocations. The experiment is executed with 128 GB memory allocation

This set of experiments reveal that ARS (FaaS) performs much better than any other serverless frameworks on the function throughput with the same memory allocation. What's more, ARS (FaaS) also greatly increases the available memory of serverless computing. Thus, it has greatly expanded the application scope of serverless computing. The current serverless computing is not suited to some computing workloads, e. g., high-performance computing and high-volume stream analytic, because of the resource restrictions of the serverless frameworks. ARS (FaaS) has overcome its limitations and explored a new application scene for it. Next, the experimental exploration of serverless computing is given on high-volume stream analytic.

4.2 Concurrency for high-volume streaming application

Four streaming applications are used including real-time event detection[17], logo detection[18], frequent pattern detection[19] and density-based stream clustering[20] to form the high-volume data stream. The testing streaming applications are compute intensive, in which CPU resources are mainly consumed. The compute logic of each streaming application is represented by a storm topology with five sequential vertices which are defined serverless functions with execution time recorder. And the concurrency is evaluated with CPU intensive function.

To test the concurrency of the serverless frameworks, two groups of experiment are set in this part. In the first experiment group, all of the four streaming applications are executed parallelly, and the degree of parallelism of each application is set to 25. Thus, there are 100 streaming function topologies running parallelly. And 500 execution times are recorded with 100 concurrent invocations. And in the second experiment group (experimental control group), the four streaming applications are executed one by one. And this procedure is repeated 25 times. Thus, all of the 500 returned execution times are recorded with one concurrent invocations.

Fig. 8 – Fig. 12 are showing the CDF (cumulative distribution function) of execution time, which is the statistics of the 1 000 returned time records. The x-axis presents execution time, and y-axis presents the probability. The dotted lines show the execution time of the function with one concurrent invocation, and the solid lines show the execution time of the function with 100 concurrent invocations. The experiment results of the control group with non-parallel setting are consistent, while the results of the first group with 100 concurrent indicate the overhead of (23% – 4 606%) over the total execution time.

From the results, it is observed that ARS (FaaS) is the only serverless framework whose execution time of the function with 1 concurrent invocation is sub-second. For the function with 100 concurrent invocations,

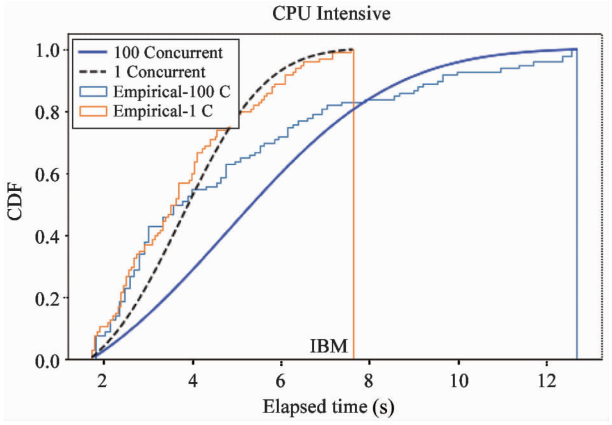

Fig. 8 The concurrency overhead with CPU intensive functions of IBM Function

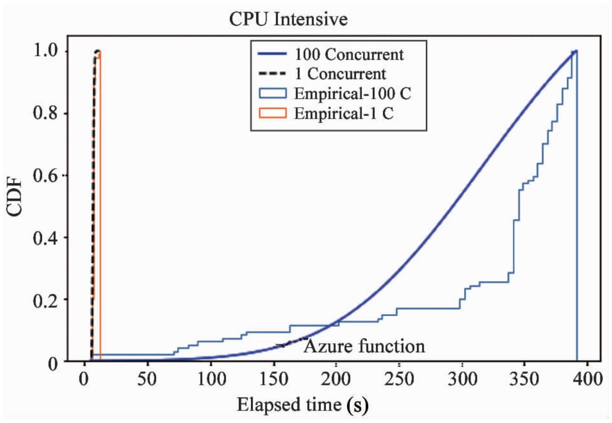

Fig. 9 The concurrency overhead with CPU intensive functions of Azure Function

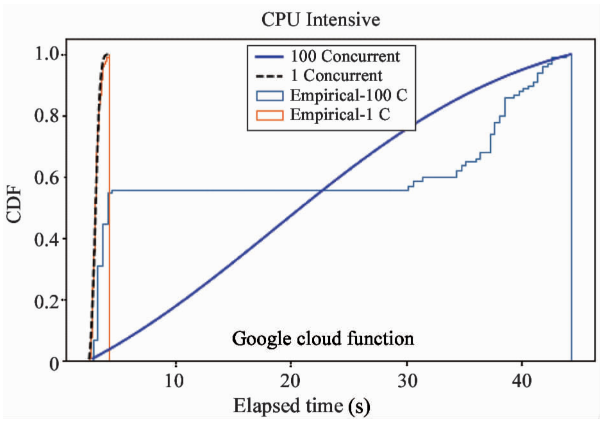

Fig. 10 The concurrency overhead with CPU intensive functions of Google Function

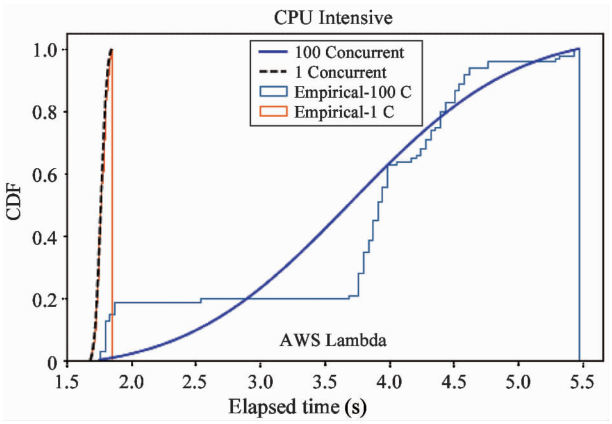

Fig. 11 The concurrency overhead with CPU intensive functions of AWS Lambda

ARS (FaaS) also achieves the best performance with the shortest execution time and lowest overhead. The observations also indicate that multiple invocations are allocated to a single function instance that may need to share compute resources. For example, there are four different streaming applications in this experiment, so four CPU intensive function invocations may take four times longer by sharing CPU time in quarter. This causes the concurrency overhead. The experiment results

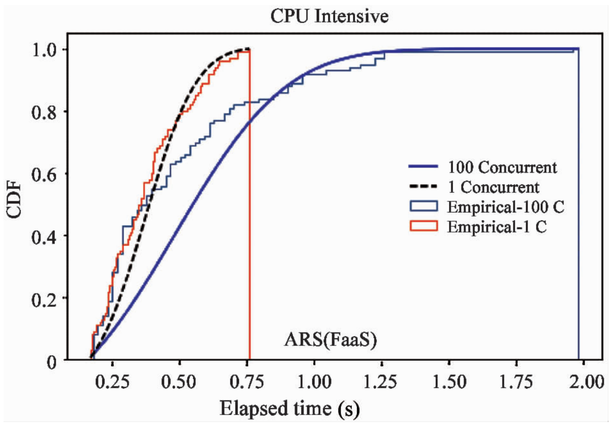

Fig. 12 The concurrency overhead with CPU intensive functions of ARS(FaaS)

indicate that ARS(FaaS) serverless framework performs much better than IBM Function, AWS Lambda, Azure Function and Google Function, on the concurrency for high-volume streaming application.

5 Conclusion

This paper introduces a novel serverless framework ARS(FaaS) which has accomplished both low latency and high resource efficiency. The design and implementation of ARS(FaaS) are presented. This framework has the advantage of a low or no initial cost to acquire computer resources, which is the key property of serverless. In this work, ARS(FaaS) framework supplies high-efficient HPC cloud resources, accelerating real-time and stream analytic on data. The experimental results have successfully validated the ARS(FaaS) framework. In the feature, much work will be done to enable autonomous resource scheduling for parallel programming, i. e. , extend the MapReduce utilization of function-as-a-service and to adapt this framework on the next generation supercomputers which reach exascale computing.

References

[1] Baldini I, Castro P, Chang K, et al. Serverless computing: current trends and open problems[J]. *arXiv*:1706.03178v1, 2017

[2] Hendrickson S, Sturdevant S, Harter T, et al. Serverless computation with openLambda[C]//Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), San Diego, USA, 2016: 33-39

[3] Yan M, Castro P, Cheng P, et al. Building a chatbot with serverless computing[C]// Proceedings of the 1st International Workshop on Mashups of Things and APIs, Trento, Italy, 2016: 5

[4] Lloyd W, Ramesh S, Chinthalapati S, et al. [C]//2018 IEEE International Conference on Cloud Engineering (IC2E 2018), Orlando, USA, 2018:159-169

[5] Akkus I E, Chen R, Rimac I, et al. {SAND}: towards high-performance serverless computing[C]//2018 USENIX Annual Technical Conference (USENIX ATC 2018), Boston, USA, 2018: 923-935

[6] Fox G C, Ishakian V, Muthusamy V, et al. Status of serverless computing and function-as-a-service (FAAS) in industry and research[J]. *arXiv*:1708.08028, 2017

[7] Alqaryouti O, Siyam N. Serverless computing and scheduling tasks on cloud: a review[J]. *American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)*, 2018, 40(1): 235-247

[8] Miller R. AWS Lambda makes serverless applications a reality [EB/OL]. https://techcrunch.com/: Verizon Media, 2015

[9] Hellerstein J M, Faleiro J, Gonzalez J E, et al. Serverless computing: one step forward, two steps back[J]. *arXiv*:1812.03651, 2018

[10] Mcgrath G, Brenner P R. Serverless computing: design, implementation, and performance[C]// IEEE International Conference on Distributed Computing Systems Workshops, Atlanta, USA, 2017: 405-410

[11] Etzion O. Event processing: past, present and futare[J]. *Proceedings of the VLDB Endowment*, 2010, 3(1-2): 1651-1652

[12] Jin X, Lee X, Kong N, et al. Efficient complex event processing over RFID data stream[C]//The 7th IEEE/ACIS International Conference on Computer and Information Science, Portland, USA, 2008: 75-81

[13] Shaikh S A, Watanabe Y, Wang Y, et al. Smart scheme: an efficient query execution scheme for event-driven stream processing[J]. *Knowledge and Information Systems*, 2019, 58(2): 341-370

[14] Cugola G, Margara A. Processing flows of information: from data stream to complex event processing[J]. *ACM Computing Surveys (CSUR)*, 2012, 44(3): 15

[15] Feng L, Kudva P, Da Silva D, et al. Exploring serverless computing for neural network training[C]// 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, USA, 2018: 334-341

[16] Liao X, Xiao L, Yang C, et al. MilkyWay-2 supercomputer: system and application[J]. *Frontiers of Computer Science*, 2014, 8(3): 345-356

[17] Nguyen D T, Jung J E. Real-time event detection for online behavioral analysis of big social data[J]. *Future Generation Computer Systems*, 2017, 66: 137-145

[18] Liu Y, Wang J, Li Z, et al. Efficient logo recognition by local feature groups[J]. *Multimedia Systems*, 2017, 23(3): 395-403

[19] Yun U. Mining lossless closed frequent patterns with weight constraints[J]. *Knowledge-Based Systems*, 2007, 20(1): 86-97

[20] Chen Y, Tu L. Density-based clustering for real-time stream data[C]//Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, USA, 2007: 133-142

[21] Lee H, Satyam K, Fox G. Evaluation of production serverless computing environments[C]// 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). San Francisco, USA, 2018: 442-450

Cheng Yingchao, born in 1989. He is a visiting scholar in Statistics Department of Texas A&M University, USA. He is also a Ph. D candidate in School of Computer Science and Technology of Guangdong University of Technology. He received his B. S. degree from Chongqing University of Arts and Sciences in 2013. His research interests include machine learning, data mining, high performance computing and service computing.