Enter your Paper Title Here

Author 1
School of Electrical and
Computer Engineering
University
Location
Email:

Author 2
University
Location
Email:

***Abstract*—Serverless computing and function execution on cloud computing services are getting lots of attentions from academia and industry. One of the reasons for the success of serverless computing is the straightforward interface for end-users while using the service. The simple interface lets users to control allocated RAM for a function run-time. However, this abstracts too much information, and users cannot expect how their applications will perform except CPU and memory. To address the issue, we thoroughly evaluate the network resource performance from various aspects. Different from people's general belief, the impact from the network resource to the overall serverless application is quite significant, and network resource isolation among concurrent execution is rarely provided from the service provider. Based on the result presented in this paper, we are sure that the network resource performance of function execution model should be more visible and expectable to expand applications of serverless computing.**

***Index Terms*—serverless computing, FaaS**

I. Introduction

Serverless computing is gaining popularity with Function-as-a-Service (FaaS) execution model. Without overheads of provisioning cloud instances and scaling them as compute demand changes, FaaS allows system developers to focus on core logic implementation. Many public cloud service vendors provide FaaS execution model with their own custom cloud services; e.g., block storage, database, messaging, event notification. One of the major benefits of the FaaS execution model is the straightforward usage that allows users to select a minimal set of configurations; the Lambda service by AWS, the first public FaaS provider, lets users to set the maximum memory size for function run-time, and CPU quota is allocated proportional to the maximum RAM size as well as the service charge.

Despite of the popularity, many of recent applications of FaaS execution models are limited to orchestration of multiple cloud services or functions by invoking other proprietary cloud services or custom function as a mean of passing input and output arguments. Bag-of-tasks type applications that do not impose dependencies among parallel jobs are also a good candidate for FaaS model. Processing large-scale datasets with a well-designed parallel algorithm in a cloud computing resources becomes norm, but such big-data applications do not fit well with the current FaaS execution model. Hellerstein et. al. [1] also insists that the data-friendly application should be natively supported by the FaaS execution model to widen the adoption of serverless computing in many related fields.

Furthermore, the abstracted resource and billing model of FaaS hide too much information of underlying compute resources, and users are likely to be ignorant of how the function will perform. To address the issue, Wang et. al. [2] thoroughly evaluated various public FaaS execution environments, and they uncovered many characters regarding service scalability, performance isolation, hardware heterogeneity, and the cold-start problem. Though the work discovered unknown attributes of various FaaS environments, the evaluation mainly focused on CPU and memory resources, and performance characters of network resources are barely covered; for network resource performance, they conducted only one experiment using *iperf3* system command to observe the resource isolation character by invoking multiple functions on a same host. They concluded that the aggregated network bandwidth does not differ from concurrent executions of various number of executions; hence, no network resource isolation among function invocations. However, we believe that the network performance in the FaaS runtime needs deeper investigation, because the network performance will take larger portion as the serverless computing broadens its application scenarios to data-friendly ones.

In order to better understand the network performance of FaaS using the container technology, we thoroughly measured various network related metrics (total aggregated bandwidth usage of a host machine, parallel download tasks end-to-end response time, total download time) of AWS Lambda service with a micro-benchmark using *iperf3* system command and a realistic workload that accesses a block storage service. We investigated the performance of both download and upload tasks from a function run-time. With the comprehensive experiments and analysis, we could uncover the following traits.

•

We are positive that the findings in this paper are essential when building data-intensive applications on FaaS environments with insights about the impact of allocated RAM and CPU resource to the network performance. The network performance evaluation from many concurrent download functions reveals that the end-to-end response time can be shortened due to increased parallelism, but the total aggregated download time across functions increases significantly that

results in increased bills for end-users. Though the current FaaS execution model gains popularity from the abstraction of complex resource provisioning and scheduling, users should be conscious about the impact from the concurrent executions on a same host and maximum memory allocation to avoid unexpected performance when deploying data-intensive applications on a FaaS environment.

The remainder of this paper is organized as follow. Chapter

II. Related Work

Many cloud service vendors provide FaaS execution models; Lambda by AWS, Functions by Azure, and Cloud Functions by Google. Different from other public cloud service providers, IBM open-sourced their function service implementation, Openwhisk[1]. OpenLambda [3] is another open source implementation of FaaS execution model. As a container orchestration tool, Kubernetes [4] is adopted in many industry applications that are built in the micro-service architecture form. To further extend the functionality of Kubernetes, many open source serverless platforms built on top of Kubernetes are actively under development; e.g., OpenFaaS, Kubeless, Knative. The depicted works have significant contribution to expand the adoption of FaaS execution model in the industry and academia, but they show unpredictable performance due to too much resource abstraction.

Wang et. al. [2] and Lee et. al. [5] compared public function execution environments. They mainly focused on evaluation of concurrent function throughput, service scalability, cold-start problem, and quantitative comparison. Their evaluation mainly focused on CPU and memory resource performance and did not cover network resource thoroughly as we do in this paper. As the FaaS execution environment becomes more data-heavy application friendly, we believe that the impact from the network resources becomes as important as CPU and memory.

Despite of the FaaS popularity, its applications are quite limited to orchestration of many cloud services. To extend FaaS applications, Kim et. al. [6] proposes to run data analysis jobs on Flint using serverless environment. Ishakian [7] runs a deep neural network model inference engine on a FaaS platform and compared the performance with dedicated machines. Feng et. al. [8] proposed an algorithm to run DNN training tasks using FaaS. Pywren [9] tries to run large-scale linear algebra and machine learning jobs on AWS Lambda. Recent efforts in the literature are in line with expanding FaaS applications to data-friendly jobs [1], and we are sure that the performance impact from network resources will be quite significant.

III. Function Execution Mechanism for Serverless Computing

Invoking N simultaneously creates N function instances and allows maximum scaling of function instances that can be measured at once. When the instance processes and terminates the request, the state changes to idle, which reduces the latency at startup. AWS Empty Packages. Cold start latency

[1]https://github.com/apache/incubator-openwhisk

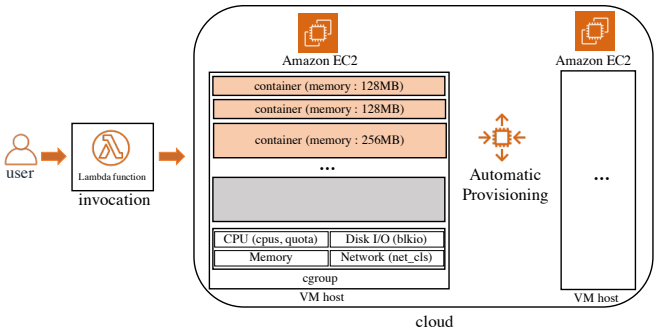

Fig. 1. aws lambda

decreases as the function memory increases. Cold start latency increases when multiple instances are running at the same Maximize VM host memory usage to reduce latency and maintain consistency with efficient instance scheduling. When a workload occurs, resources must be allocated automatically. As applications become more complex, there is a dynamic allocation of VM hosts, monitoring services. Monitors applications and automatically adjusts capacity to service at the lowest cost and maintain predictable performance. Services provide a simple yet powerful user interface. Scale by optimizing performance and cost and balancing them appropriately. The AWS lambda function runs in a container (sandbox) that isolates it from other functions and provides resources such as memory specified in the configuration of the function. When a function is created or the resource configuration is run for the first time, a new and updated container with the appropriate resources is created and loaded by the code. In a container or code. Lambda can reuse old containers without changing the code. Stop the process and re-enable the process the next time you call a function. The data is temporarily stored non-permanent and cleared when the function instance (Lambda = container sandbox) is deleted. When an instance is idle it is charged when it runs on consumption of resources without paying for consumed resources.

cgroup provides resource limits and reporting within the container space and provides detailed control over when and when host resources are allocated to the container. All containers use the same percentage of cpu cycles. This ratio can be modified by changing the weight of the cpu share in the container based on the weight of other running containers. The ratio only applies when the cpu concentration process is running, and when one container's task is idle, the other container can use the cpu remaining ratio, and the actual cpu ratio depends on the number of containers running on the system. The –cpu-qua flag uses a scheduler that restricts the use of cpu in containers and handles resource allocation for running a process called commletly fair schedular (CFS). The process in the container can also limit the amount of memory required and is limited by the memory hard limit. Memory reservation allows the container to use as much memory as it needs, detect memory contention or lack of memory, and limit the amount of memory it uses to the

reservation limit if memory sharing is allowed to be large. By default, all containers have the same percentage of block IO bandwidth (blkio). For adjusting the I/O of the container, use the weight setting. The net-cls in the network help the traffic controller prioritize the packet processing by tagging it to identify packets that occur in a particular cgroup.

To provide a simple interface to developers, public cloud vendors abstract complex details of infrastructure provisioning, such as the number of CPU cores, memory size, network/IO configurations. Instead, they provide straightforward metrics to help programmers easily understand performance and cost. For example, AWS Lambda allows users to configure only memory size. Compute resources are allocated relative to the configured RAM size, and the product of function execution time and allocated RAM size becomes the cost. However, this simple cost model and the abstraction of complex infrastructure might result in unexpected performance to users. al [2]. analyzed the behavior of various cloud vendors FaaS, and most services suffer from significant performance degradation due to co-location of containers in a same host.

IV. Performance Characters of Function Services

A. Experiment Methodology

To elaborate the network performance impact from containers co-location in a same host, we measured the available bandwidth with iperf3 library from AWS Lambda executions when they are invoked on a same host. We used the container-to-host mapping detection mechanism described in Wang et. al [2]. First, we profile self/cgroup file of the proc file-system to ensure that the Lambda operated on the same host. It provides the VM identifier which is started with "sandbox-root-" to verify that each Lambda invocation is executed on the same VM, and we can use this information to use simultaneous experiments. Next, we establish both a server and client to perform an iperf3 test. An iperf server started EC2 instance which is set up as a same VPC network with Lambda. The VM instance was selected by experimenting with one of the environments in which Lambda runs. The security group in EC2 specifies the 5201 port that the iperf uses. In EC2, run the server with the s option. Lambda grants VPC privileges (AWSLambdaVPCAccessExecutionRole) and sets the same VPC and subnets as set in EC2. The Lambda code uses the iperf3 library to create client experimental environments, set test parameters, and conduct experiments.

We found in our experiment that Lambda was running on five VM types which has different cpu model name and memory devices. Due to the differences in network bandwidth by VM Type, we chose one of the VM types for precise measurements, we run EC2 Instance (c3-large), and we installed Docker to run the containers which is similar to a sandbox. Docker runs the process in an isolated container, the process that runs on the EC2 instances. The container runs on Linux by default and shares the kernel of the host machine with other containers. Docker provides resource management to control how much memory, or CPU a container can use. We limited the resources of the container by given –memory(fixed maximum amount of memory the container can use) and –cpus(the fractional number of total CPU resources a container can use) options when executing the docker for resource allocation similar to the cpu allocation according to the memory of the Lambda. A fractional number of CPU is 0.06 for minimum 128 MB of container memory and 2 for up to maximum 3008 MB of container memory.

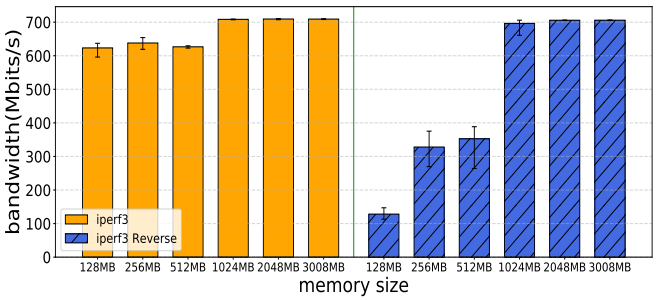

Fig. 2. Network bandwidths iperf3

B. Evaluation

Figure 2 shows network bandwidth across a given memory in lambda and client specific options reverse. In the iperf3 test without the reverse option, the client(Lambda) sends packet to the server(EC2). Normally, the test data is sent from the client to the server, and measures the upload speed of the client. Measuring the download speed from the server can be done by specifying the reverse option on the client. This causes data to be sent from the server to the client. It is same to Lambda downloading an object in the S3 Bucket. The horizontal axis shows the memory of Lambda, and the vertical axis shows Lambda's network bandwidth. The orange bar shows the network bandwidth without the reverse option and the blue bar present network bandwidth with the reverse option. We can observe that the total bandwidth is capped at 600 Mbps regardless of the memory. However, as the number of memory decreases in without reverse options, each Lambda has lower bandwidth allocated.

In Figure 3, the horizontal axis shows the number of concurrently running containers in a host, and the vertical axis shows the network bandwidth. We given the parallel options which is the parallel client streams to run. The red-solid line shows the aggregated bandwidths across all the containers, and the bars present per-container bandwidth. We can observe that the total bandwidth is capped at 800Mbps regardless of the number of containers running. However, as the number of concurrent containers increases, each container (or function) has lower bandwidth allocated.

To see the impact of limited network bandwidth per each container on a shared host, Figure 4 presents data preparation time from a function run-time. In the experiment, we generate a synthetic workload whose total input dataset size is 1000MB that is divided into 10MB blocks (total of 100 blocks) stored on AWS S3, and each Lambda run-time downloads a 10MB block for function execution. Though most of current AWS

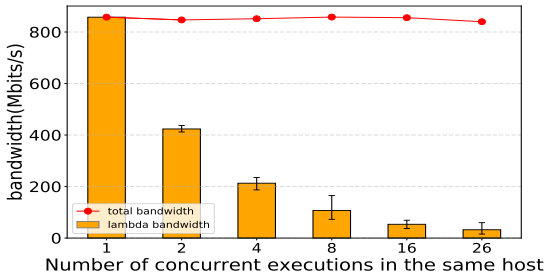

Fig. 3. Network bandwidths iperf3 parallel

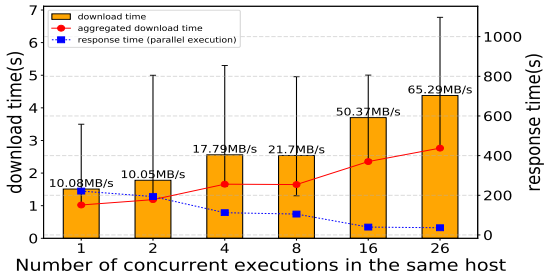

Fig. 4. download response time and aggregated time in parallel

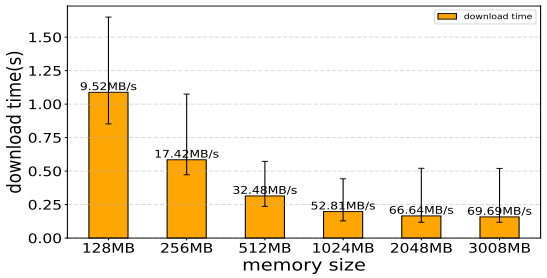

Fig. 5. download response time and aggregated time in memeory

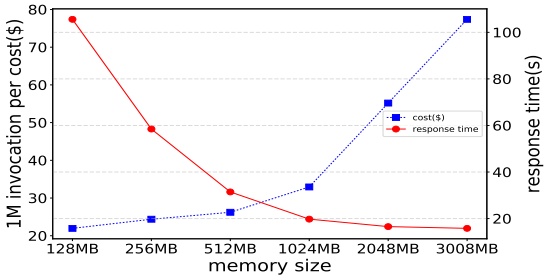

Fig. 6. download response time and Lambda 1M per costs

Lambda workloads are limited to simple orchestration functions or composite functions, Hellerstein et. al. [1] insist that the data-friendly applications should be natively supported by the FaaS. As presented in the synthetic workload, we believe that downloading input datasets from functions becomes norm. In the figure, the horizontal axis shows the number of concurrent execution in a same host. The rectangles with error bars represent per-container download time whose value is shown in the primary vertical axis. As the number of concurrent execution increases, the average/min/max download time increases. The blue-dotted line with square markers presents the response time to download all 100 blocks from parallel execution whose value is shown in the secondary vertical axis. As the degree of parallel execution increases, the response time decreases. The red-solid line with round markers shows the aggregated download time from all the parallel function executions, and it keeps increasing. The text above the bar measured network bandwidth as Mbps by profiling the amount of packets received and transmitted using net/dev in the proc file system. The result indicates that the larger number of concurrent function execution that utilizes network resources can improve the end-to-end response time, but the aggregated running time increases due to the network resource contention. Lambda execution is billed in the unit of 100ms usage, and larger aggregated download time due to too many function execution can cause cost increase.

In this Figure 5, the horizontal axis shows the number of memory in AWS Lambda and vertical axis shows the memory download time for 100 blocks in the environment similar to previous experiment. Increasing the memory of Lambda increases the network bandwidth. As a result, download time is shortened, and total time are reduced proportionally. From Lambda's memory of more than 1024MB, there is no significant difference in network bandwidth and the overall download time is similar. It doesn't have the time benefit of reducing download time, this results in a cost-effective waste to users as the cost of memory increases. The following figure illustrates the cost and total time of memory.

In this Figure 6, the horizontal axis shows the number of memory in Lambda. The blue-dotted line with square markers presents the 1 millions per lambda invocation to download all 100 blocks whose value is shown in the primary vertical axis. Lambda counts a invocation each billing time it starts executing in request. Billing time is calculated from the latency which is download time for s3 object. The cost depends on the amount of memory it allocate to Lambda function. As the amount of memory increases, the cost increases. The red-solid line with round markers shows the total response time, and it keeps decreasing.

To make sure the network bandwidth difference between reverse option and default option in iperf3 experiment, we compared uploads and download experiments. In the experiment, we upload 10MB blocks(total of 100 blocks) on AWS S3 bucket. In Figure 7, the vertical axis shows the upload time for 100blocks. Similar to previous download experiments, the average upload time increases as the number of concurrent execution increases. As the number of concurrent execution increase, the upload network bandwidth is increase.

In this Figure 8, the vertical axis shows the upload time. In contrast to the download experiment, the upload experiment do not differ significantly in performance based on memory. From the memory of Lambda over 512MB, there is no big difference between 3008MB and upload time. The error bar in the bar causes an interaction in which upload time is delayed

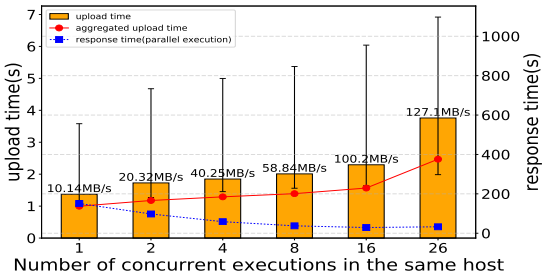

Fig. 7. upload response time and aggregated time in concurrent execution

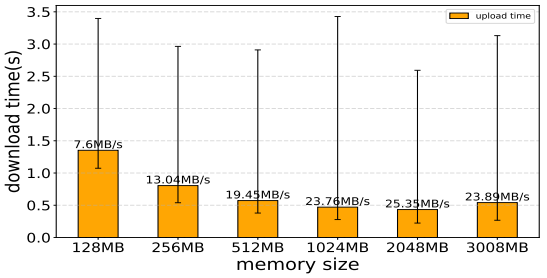

Fig. 8. upload response time and aggregated time in memory

somewhat later depending on the network situation of the AWS S3 service while uploading 100 blocks.

In this Figure 9, the horizontal axis shows the number of memory in Lambda. The blue-dotted line with square markers presents the 1 millions per lambda invocation to download all 100 blocks whose value is shown in the primary vertical axis. Lambda counts a invocation each billing time it starts executing in request. Billing time is calculated from the latency which is download time for s3 object. The cost depends on the amount of memory it allocate to Lambda function. As the amount of memory increases, the cost increases. The red-solid line with round markers shows the total response time, and it keeps decreasing.

V. Conclusion and Future Research Direction

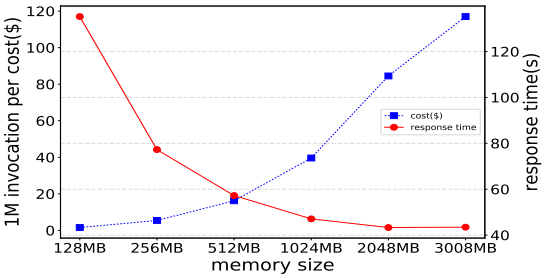

Fig. 9. upload response time and Lambda 1M per costs

References

[1] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[2] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[3] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[4] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[5] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 442–450. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00062

[6] Y. Kim and J. Lin, "Serverless data analytics with flint," *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 451–455, 2018.

[7] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," *CoRR*, vol. abs/1710.08460, 2017. [Online]. Available: http://arxiv.org/abs/1710.08460

[8] L. Feng, P. Kudva, D. D. Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2018, pp. 334–341. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00049

[9] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99Symposium on Cloud Computing, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 445–451. [Online]. Available: http://doi.acm.org/10.1145/3127479.3128601