Network Resource Isolation in Serverless Cloud Function Service

Jeongchul Kim
College of Computer Science
Kookmin University, South Korea
kjc5443@kookmin.ac.kr

Jungae Park
College of Computer Science
Kookmin University, South Korea
barkjungae@kookmin.ac.kr

Kyungyong Lee
College of Computer Science
Kookmin University, South Korea
leeky@kookmin.ac.kr

***Abstract*—Serverless computing and function execution using cloud computing services are currently the subject considerable attention from both academia and industry. One of the reasons for the success of serverless computing is its straightforward interface that allows users to control the size of the memory allocated for the run-time of a function. However, this approach may result in the abstraction of too much information, and users cannot predict how their applications will perform, especially for the network resource. To address this issue, we evaluated several aspects of network resource performance. Despite the general belief, the variation of serverless applications' network performance is quite significant, and the ability to isolate network resource allocation during concurrent execution is rarely provided by service providers. Based on the results presented in this paper, we insist that network resource performance of functional execution models should be more visible and predictable, in order to expand the applications of serverless computing.**

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

I. Introduction

Serverless computing is gaining popularity with the Function-as-a-Service (FaaS) execution model. Without incurring the overheads involved in provisioning Cloud instances and being able to scale them as needed, FaaS systems allow system developers to focus on the implementation of core logic. Many public Cloud service vendors provide an FaaS execution model with their own custom cloud services, such as block storage, database, messaging, and event notification. One of the major benefits of the FaaS is its straightforward interface, which allows users to select a minimal set of configurations. The Lambda service provided by AWS, the first public FaaS provider, lets users set the maximum memory size for function run-time, and the CPU quota is allocated proportionally to the RAM size, as is the service charge.

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

The abstracted resource and billing model of FaaS hides considerable information about underlying compute resources, and users are likely to be ignorant of how the function will perform. To address this issue, Wang et. al. [2] evaluated several public FaaS execution environments, and uncovered many issues regarding service scalability, performance isolation, hardware heterogeneity, and the cold-start problem. Although this work identified important attributes of various FaaS environments, the evaluation mainly focused on CPU and memory resources, and the performance characteristics of network resources are barely covered; for network resource performance, they conducted only one experiment using the *iperf3* system command to observe resource isolation characteristics by invoking multiple functions on a same host. They concluded that the aggregated network bandwidth does not differ from that of concurrent executions of various numbers of executions; hence, they did not identify any network resource isolation mechanisms amongst function invocations. However, we believed that the network performance in the FaaS runtime needed deeper investigation, because it will become more important as the serverless computing approach broadens its application scenarios to data-intensive applications.

In order to better understand the network performance of FaaS using container technology, we measured various network-related metrics: total aggregated bandwidth usage of a host machine; parallel download tasks end-to-end response time; and total download time of an AWS Lambda service with a micro-benchmark, using the *iperf3* system command and a realistic workload that accesses a block storage service. We investigated the performance of both download and upload tasks from the run-time of functions.

From the results of our experiments and subsequent analyses, we made the following observations:

- The *iperf3* system command, widely used as a network micro-benchmark in previous work, does not provide an accurate estimate of realistic network performance
- Quantitative evaluations reveal that the configuration of memory allocation for functions makes a difference in network bandwidth performance, even though they are not enforced by a service provider
- A response time and cost evaluation revealed that allocat-

978-1-7281-2406-3/19/$31.00 ©2019 IEEE
DOI 10.1109/FAS-W.2019.00051

ing more resources to function run-time does not always produce a proportional gain in performance, and costs can increase significantly

We believe that the findings in this paper will be valuable when building data-intensive applications in FaaS environments, by providing insights into the impact of the allocation of RAM and CPU resources to network performance. The comparison of networks from many concurrent download functions reveals that the end-to-end response time can be shortened with increased parallelism, but the total aggregated download time across functions increases significantly, resulting in increased bills for end users. Although the current FaaS execution model is popular because of the abstraction of complex resource provisioning and scheduling, users should be conscious of the impact of concurrent executions on a single host to avoid unexpected performance when deploying data-intensive applications in an FaaS environment.

The remainder of this paper is organized as follow. Section II discusses related work. Section III describes details about serverless computing and function execution environments. Section IV presents thorough evaluation of a function service regarding network resource performance, and Section V concludes the paper with future work.

II. Related Work

Many cloud service vendors provide FaaS execution models; Lambda by AWS; Functions by Azure; and Cloud Functions by Google. In contrast to other public cloud service providers, IBM open-sourced their function service implementation, Openwhisk. OpenLambda [3] is another open source implementation of FaaS. As a container orchestration tool, Kubernetes [4] has been adopted for many industry applications that are built using a micro-service architecture. To further extend the functionality of Kubernetes, many open source serverless platforms built on top of Kubernetes are actively under development, including OpenFaaS, Kubeless, and Knative. These applications have the potential to contribute significantly to the expansion of the adoption of FaaS in industry and academia, but they tend to show unpredictable performance because of a high level of resource abstraction.

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

Despite the popularity of FaaS, its applications are currently quite limited to the orchestration of multiple cloud services. To extend the scope of FaaS applications, Kim et. al. [6] propose running data analysis jobs on Flint using a serverless environment. Ishakian et. al. [7] ran a deep neural network model inference engine on an FaaS platform and compared its performance with those of dedicated machines. Feng et. al. [8] proposed an algorithm to run DNN training tasks using FaaS. Pywren [9] ran large-scale linear algebra and machine learning jobs on AWS Lambda. Son et. al. [10] presented an algorithm to build an optimal cloud environment to execute matrix multiplication tasks, and the proposed algorithm can be applied to a function environment. Recent efforts reported in the literature are tending towards expanding FaaS applications to data-heavy jobs [1], and we have demonstrated that the performance impacts from network resources can be significant.

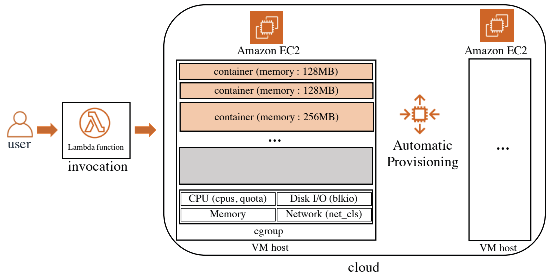


Fig. 1: Procedure of function invocation (AWS Lambda)

III. Function Execution Mechanism for Serverless Computing

Initial public cloud computing services offered virtualized instances, so that users could run an operating system image based on needs. From the initial offering, cloud computing services developed in the direction of hiding the complexities of infrastructure provisioning, operating system dependency, software installation, and automatic scaling as demands change. In the context of resource abstraction, serverless computing provides several services freeing users from the burden of server instance provisioning. For example, the Amazon API Gateway provides a public web endpoint service, usually with HTTP protocol support, alleviating the burdens of instance provisioning and the installation of necessary software, such as Apache web server or nginx. For database services without server provisioning, AWS provides DynamoDB (key-value storage) and Aurora (RDBMS), which are fully-managed by the service provider, allowing users to focus on core data management tasks.

In order to decrease challenges of users for the setup of compute-processing environments, many cloud service vendors provide a function service, called FaaS. With this service, a user implements the core functionality of an application and registers it with an FaaS. For the invocation of registered functions, an event-based approach is widely used. The sources of events are generally other services provided by the vendor. For example, in a Lambda, FaaS of AWS, functions can be invoked when an image file is uploaded to Amazon S3 to perform further registered actions, such as changing permissions for public access and file transcoding. With the abstraction of instance provisioning, a service provider can optimize resource utilization by packing as many functions as possible in a single host. In order to achieve this goal, a service provider uses container technology that has relatively lower management overhead and start time than virtualization [11].

In addition to the reduced overhead of server provisioning, most FaaS vendors provide a simple memory-based billing mechanism. In the function configuration step of AWS Lambda and Google Cloud Function, users have to decide upon the maximum memory size required by a function, and the service bill is calculated as the registered memory size times the duration of function execution. Other resources are known to be allocated in proportion to the configured memory size. In contrast to a virtualization technique that relies on a hypervisor for resource isolation among multiple tenants, container technology relies on cgroup, which provides per-resource isolation. In cgroup, the parameter *memory.limit_in_bytes* sets the maximum memory size that a container can use. There are many ways to control the maximum CPU usage of a container: container-to-core binding; share (priority)-based allocation, and absolute time allocation. When using absolute time allocation, *cpu.cfs_period_us* specifies the period of time that the CPU *quota* is reallocated. In combination with *cpu.cfs_period_us*, *cpu.cfs_quota_us* sets the amount of CPU time that a container can use during *cpu.cfs_period_us*. In FaaS execution model, CPU resources are allocated by setting the ratio of $\frac{cpu.cfs\_quota\_us}{cpu.cfs\_period\_us}$ to the ratio of configured memory size and the host machine's total memory size. For network resource allocation, cgroup's *net_prio* subsystem allows the setting of priorities among many containers, and can be used to differentiate network resource allocation. For block IO device allocation, *blkio* subsystem's *weight* option allows the user to set the relative importance of a containers IO usage. With its container technology and resource isolation mechanism, an FaaS vendor can provide performance guarantees with a simple billing mechanism. Based on the analysis of the FaaS vendors' performance, [2], [5] determined that CPU and memory resources are allocated proportionally to user payments, but these authors did not produce an in-depth discussion of network resource allocation and isolation. An FaaS working scenario is shown in Figure 1.

IV. Performance Characters of Function Services

Prior work [2], [5] has focused on evaluation and comparison among many FaaS providers, focusing mainly on memory allocation and the corresponding CPU performance. As FaaS applications become more data-friendly, it appears that network performance becomes crucial to the provision of predictable performance, so we focused on the evaluation of FaaS network performance with respect to memory allocation and concurrent execution.

A. Experiment Setup

The workload applications in the experiments mostly consider the network resources, and the impact from the run-time of a function should be minimal. As a micro-benchmark, we used the *iperf3* system call to measure available network bandwidth. To use the *iperf3* system command, we created a large-enough dedicated server using an Amazon EC2 *c4.8xlarge* instance with the *-s* option of *iperf3*, as function run-times do not support a direct connection [1]. The Lambda function run-time works as a client where the IP address and port of the server is passed as function arguments. Using the *iperf3* command, we can let a client (function run-time) work as either a data uploader (default option) or downloader (with *-R* option), and we present the result in both cases when necessary. To measure latency of data download and update, we use Amazon S3 as a source and destination from the Lambda run-time and utilize Amazon Fine Food Review text dataset[1] with *Python2.7* and *boto3* library. The different memory configuration of function run-time limits the maximum file size to be loaded in the memory, and we partition the dataset into chunks with 10, 20, 50, 100, and 200MB. All AWS resources in the experiments are deployed on N. Virginia region. We share the source codes used in the paper publicly[2] for reproducibility.

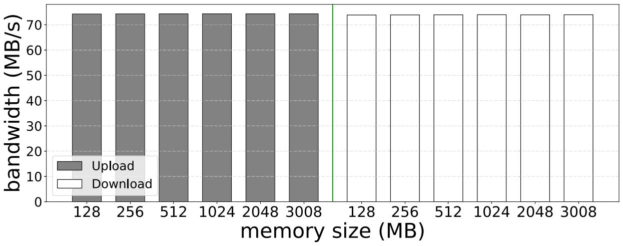
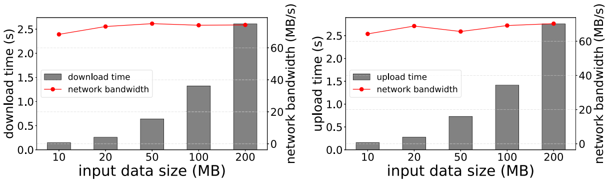


Fig. 2: Network bandwidths evaluation with iperf3



(a) Download time

(b) Upload time

Fig. 3: Response time with different file size

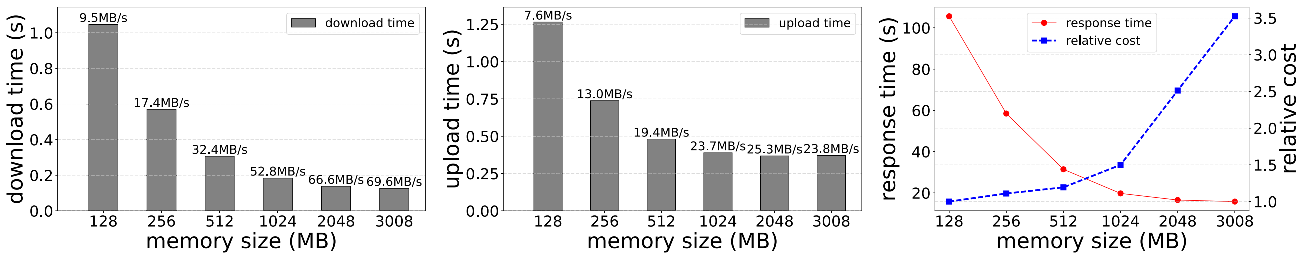
We present various metrics related to network performance. The *download time* is the time taken by a single function run-time to download a file. The *response time* measures the end-to-end latency when downloading files in parallel from multiple function executions. The large input dataset was partitioned into small chunks so that parallel download was implementable. The *aggregated download time* is the accumulated download time from multiple function executions. Unlike the response time, aggregated time does not consider function parallelism by adding up download time from each container, and it determines billing for the download service.

B. Impact of Memory Size Configuration

We first evaluated the network bandwidths available with AWS Lambda with different memory size configurations by using the *iperf3* system command. In Figure 2, the hori-

[1] https://snap.stanford.edu/data/web-FineFoods.html

[2] https://github.com/kmu-bigdata/faas-network



(a) Download time

(b) Upload time

(c) response time - cost

Fig. 4: The impact of function's configured memory size to the network performance and cost

zontal axis shows the memory size configured in Lambda. The leftmost six bars show the available bandwidth when a function run-time works as an uploader, and the rightmost six bars represent the available bandwidth when a function run-time works as a downloader, data obtained using the *iperf3 -R* option. Previous work [2] has investigated the bandwidth available in a function execution environment in the upload case, and the value obtained matches with Figure 2. From these experimental results, we can see that the service provider does not provide different levels of network quality based on the configured maximum memory size.

Evaluation with the *iperf3* system command provides an easy way to investigate the network bandwidths available for a function run-time, but it does not represent a realistic scenario for data applications that may downloading or uploading of files from a shared block storage. To understand the network performance of FaaS under a realistic scenario, we performed download and upload experiments using blocks of data of different sizes. Figure 3 shows the download (Figure 3a) and upload (Figre 3b) times of input files of different sizes on a Lambda run-time with 1024MB of memory configured. The primary vertical axis shows the response time taken to process a chunk, and the secondary vertical axis shows the network bandwidth consumed. The file size does not have a noticeable impact on network bandwidth use. We used a 10MB input chunk file size in the following experiments, unless otherwise noted, that is executable with the minimal memory configuration (128MB).

To evaluate the impact of memory size when real datasets are accessed from a function execution environment, we measured the download time, upload time, and the relationship between response time and cost (Figure 4). Figure 4a shows the download time for each function run-time when the total file size is 1 GB, divided into 10MB chunks. Thus, the total number of chunks to download is 100, equal to the total number of function executions required to process all of the chunks. In the figure, we show the median value across many invocations, in order to avoid impacts from unavoidable long-tail latency in a cloud environment [12]. In Figure 4a, the median download time decreases as the functions allocated memory size increases. This observation contradicts the results from the iperf3 experiment (Figure 2) which showed that function memory allocation does not have an impact on the network bandwidth performance. In addition, with respect to the amount of available bandwidth, access to S3 services from a function run-time exhibits much lower available bandwidth than the *iperf3* tests; for 128MB of configured memory for a function, *iperf3* shows about 70MB/s while the download from S3 shows 9.5MB/S. Amongst many possible reasons, we believe that differences in the experiment environment and scenarios are likely to be the most significant reason for this difference. In the *iperf3* experiment, we created a VPC in which a function run-time and EC2 instance can talk to each other via a fast local area network. However, access to S3 from a function run-time might include routing through the public Internet, even though the services exist in the same AWS region. Another difference is the execution environment: for the iperf3 experiment a system command is invoked, but download from S3 includes Python 2.7 with the boto3 library to actually access the S3 service. We also believe that the *iperf3* test involves lower memory usage and CPU utilization than does using Python with the boto3 library, and more intense resource usage of S3 downloads actually degraded the network performance.

Figure 4b shows the per-function upload time (median value) and available bandwidth. Similarly to the results shown in Figure 4a, the available upload bandwidth increases as the allocated memory size increases, and these findings are also contrary to the results shown in Figure 2. From the experimental results presented in Figures 4a, 4b, and 2, we can conclude that the widely-used *iperf3* benchmark does not accurately reflect the network performance of function environments. Obviously, in an FaaS application, the chance of accessing S3 is higher than using the *iperf3* command. Thus, to represent realistic data-intensive applications in a function environment evaluation, we have to consider using external data source in order to better understand the behavior of function execution environments. We can also conclude that although the service provider does not differentiate available network bandwidth of a function run-time based on the allocated memory size, the limited memory size and CPU usage quota impact the network performance, and functions are likely to use limited network bandwidth based on memory allocation.

AWS Lambda has a unique billing model that reflects the configured maximum memory size and running time of a function. To investigate the impact of function memory configuration and cost to download all the necessary input files in S3, we created a response time and cost map (Figure 4c). In the experiments, each function downloaded a chunk of size 10 MB. The total number of chunks downloaded was 100, and a new function invocation happened for each chunk. In the figure, the horizontal axis shows the configured memory size. The primary vertical axis shows response time, with values shown on the solid line with circle markers. The secondary vertical axis shows the normalized cost of running the entire set of functions, with values shown on the dotted line with square markers. From the figure, we can observe that the response time decreases as the configured memory size increases. However, the normalized cost increases with increasing memory size, because the increased memory does not result in linear improvement in the overall download response time. This non-linearity becomes noticeable as the memory size becomes much larger; for example, between 2048MB and 3008MB. If we increase memory size from 128MB to 256MB, the response time will be halved, with a marginal cost increase. However, a memory increase from 1024MB to 2048MB shortens the response time by about 25%, but the cost increases by about 68%.

C. Impact of Concurrent Execution

We evaluated the impact of configured memory size on overall network performance. We then investigate the impact of concurrent execution of functions on a single host.

Concurrent Execution Evaluation Methodology : In order to decide if function executions were conducted on a same host, Wang et. al. [2] proposed a function run-time and host mapping mechanism. For concurrent execution detection, we profiled self/cgroup file of the proc file-system from a function run-time. This file provides the VM identifier, which begins with “sandbox-root”, and if the VM identifier is the same, we assume that the function runs on a same host. We ran the concurrency tests on AWS Lambda as much as possible, but the function placement is not deterministic, a situation which caused difficulties in result verification. To overcome this issue, we created a function run-time environment using an AWS EC2 instance and Docker. Among the EC2 instance types, we used *c3-large*, which has two virtual cores and 3.75GB RAM, and is known to be widely used for function run-time [2]. Docker provides an easy way to stop, start, and deploy containers. Docker uses cgroup to isolate resources among containers. For memory allocation, we use *–memory* option to specify the maximum amount of memory that a container can use, and for CPU allocation, we used *–cpus* to specify the amount of CPU time that a container can use. The *–cpus* option uses *cgroup*’s *cpu.cfs_period_us* and *cpu.cfs_quota_us*. After fixing the memory size, we set the CPU ratio proportional to the memory size. For example, assuming the maximum memory size of Lambda to be 3008MB and a host machine has two virtual cores, the 128MB container will get a CPU allocation of 0.085 ($= 2.0 \times \frac{128}{3008}$), and the 3008MB container will get CPU allocation of 2.0. With this method, we created an function execution environment on EC2 and performed the memory-based. We cannot present the result due to space limitation, but it shows very similar pattern with the Lambda environment. With the confirmation, we perform experiments on the EC2 and Docker environments when we cannot make sure many functions run on a same host.

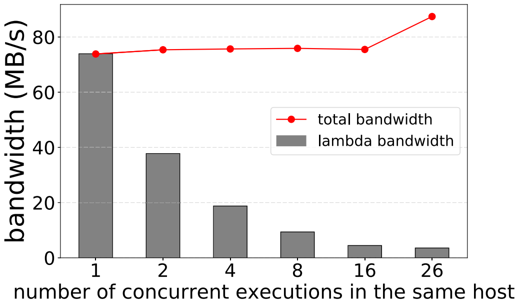


Fig. 5: Available network bandwidth with concurrent function execution measured with iperf3

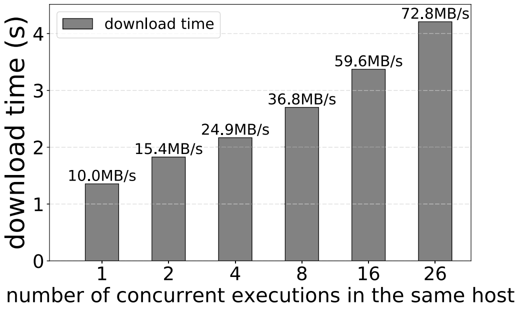


Fig. 6: download response time and aggregated bandwidth

Figure 5 shows the network bandwidths available when multiple functions run on the same host. To maximize the number of concurrently running functions on a host, we set the function memory size as 128MB, with up to 26 functions executed on the same host. We used the *iperf3* scenario to measure the network bandwidth. In the figure, the horizontal axis shows the number of functions running concurrently on the same host. The gray bar shows the median of available download bandwidths amongst the functions whose value is marked on the vertical axis. The solid line with round markers shows the aggregated network bandwidths across all concurrent functions. We do not show the upload bandwidth test result, because it is very similar to the download case. The Lambda service does not restrict network resources based on the configured memory size, and we could confirm this from the figure: as the number of concurrent executions increases, the allocated bandwidth per execution decreases, but the total aggregated bandwidth stays constant.

Figure 6 shows the download times when multiple functions are executed concurrently on the same host. In Figure 6, the number of concurrent executions is shown on the horizontal axis. The bars show the median download time required for multiple downloads to fetch all 100 chunks, each 10MB

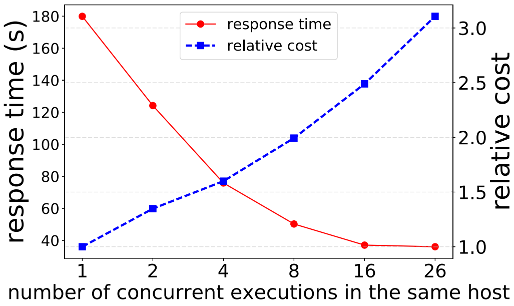


Fig. 7: response time and cost relation

in size, from Amazon S3. The numeric value on each bar shows the aggregated network bandwidths across concurrent executions. When there is one function running on a host, we can see that the download time is the shortest and the aggregated bandwidth is the smallest. As concurrency increases, the aggregated bandwidth increases, and the download time also increases, due to network resource contention on a same host.

Figure 7 shows the relationship between the response times of multiple concurrent function executions and cost, as we change the number of concurrent executions. The solid line with round markers shows the response time required to download all 100 chunks from S3 on the primary vertical axis. As the number of concurrent executions increases, the response time decreases, due to increased parallelism. The dotted line with square markers shows the normalized cost incurred with increased function concurrency, with values shown in the secondary vertical axis. Due to the increased parallelism and network resource contention, we can observe that having more functions does not always result in improved cost efficiency. When a small number of functions run concurrently, we can fetch necessary files faster by paying proportionally more. In the case of extreme concurrency, such as 16 and 26 instances, there is almost no response-time gain, but the cost increases by about 20%. Unfortunately, in a function execution environment, users do not have control over where submitted functions will execute or how many functions will execute on the same machine. From the service providers perspective, it is evident that packing as many functions as possible into a single host maximizes resource utilization. Thus, users have to be cautious about increasing the number of parallel tasks, as a specific strategy might not be optimal from the response time and cost perspective.

V. Conclusions and Future Work

In this paper, we presented the results of our investigations into the performance of network resources in data-intensive FaaS applications. First, we used the *iperf3* micro-benchmark to confirm that the AWS Lambda service does not differentiate the network resource allocation in proportion to configured memory size. We then carried out more realistic scenarios involving accessing a shared block storage service on the Internet. In the more realistic applications, we found that memory allocation strongly impacts network performance in an indirect manner, because the application involves large amounts of memory and CPU usage. If multiple functions run on the same host, network performance can degrade noticeably, and users can be charged in an unpredictable manner. We envision that the importance of network resources for FaaS applications will become significant as data-intensive applications are increasingly deployed in an FaaS environment, so the service middleware should schedule network resources proportionally to user costs.

Our ongoing work includes research into intelligent and fair scheduling of network resources in a FaaS environment. We are also conducting more experiments with other public FaaS providers to confirm the quality of network services for data-intensive applications.

Acknowledgements

This work is supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) (No. NRF-2015R1A5A7037615 and NRF-2016R1C1B2015135), the ICT R&D program of IITP (2017-0-00396), and the AWS Cloud Credits for Research program.

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 HotCloud 16*.

[4] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 2015.

[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," in *IEEE CLOUD 2018.*

[7] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform."

[8] L. Feng, P. Kudva, D. D. Silva, and J. Hu, "Exploring serverless computing for neural network training," in *IEEE CLOUD 2018.*

[9] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *ACM Symposium on Cloud Computing 17.*

[10] M. Son and K. Lee, "Distributed matrix multiplication performance estimator for machine learning jobs in cloud computing," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 638–645. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00088

[11] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *IEEE ISPASS 2015.*

[12] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun, "Metis: Robustly tuning tail latencies of cloud systems," in *USENIX ATC 2018.*