Noname manuscript No.
(will be inserted by the editor)

I/O Resource Isolation of Public Cloud Serverless Function Runtimes for Data-Intensive Applications

Jeongchul Kim · Kyungyong Lee

Received: date / Accepted: date

Abstract Serverless computing and a function execution model, Function-as-a-Service (FaaS), are currently receiving considerable attention from both academia and industry. One of the reasons for the success of serverless computing is its straightforward interface that abstracts complex internals of cloud computing resource usage and configurations. However, this approach may result in hiding too much information about how underlying cloud resources would work, entailing that users cannot predict how their applications will perform, especially for IO-heavy ones. To address this issue, we evaluate several aspects of network and disk IO performance with realistic workloads using public FaaS systems. Our analysis reveals that current public FaaS systems do not provide appropriate levels of IO performance differentiation, and the ability to isolate network resource allocation during concurrent execution is rarely offered by service providers. Based on the results presented in this paper, we insist that it must be mandatory for network and disk IO resource performance of FaaS to be more visible and predictable, as is the case for memory and CPU, in order to expand serverless computing applications to data-intensive ones.

Jeongchul Kim
Department of Computer Science. Kookmin University.
Seoul. South Korea.
E-mail: kjc5443@kookmin.ac.kr

Kyungyong Lee (*Corresponding Author*)
Department of Computer Science. Kookmin University.
Seoul. South Korea.
E-mail: leeky@kookmin.ac.kr

1 Introduction

Serverless computing is gaining popularity with the 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, databases, 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. For example, 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 their 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, which do not impose dependencies among parallel jobs, are also a good candidate for the FaaS model. Characteristics of current prevalent FaaS applications are stateless, and they require minimal interaction among function run-times [1]. The hardware and instance configuration of FaaS run-times reflect such characteristics: no direct communication among function run-times support, no deterministic scheduling support, and a small amount of attached disk storage.

Using cloud computing resources for processing large-scale datasets with well-designed parallel algorithms is becoming the norm, but such big-data applications do not fit well with the current FaaS execution model.

Hellerstein et. al. [2] 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 high-level abstraction of the resource and billing model of current FaaS hides considerable information about underlying compute resources, meaning that users are likely to be ignorant of how a given function will perform. To address this issue, Wang et. al. [3] and Lee et. al. [4] 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 that are closely related and dependent performance metrics in FaaS, barely covering the performance characteristics of network and disk IO resources; for network resource performance, they conducted an experiment using the *iperf3* system command to observe resource isolation characteristics by invoking multiple functions on the same host. They concluded that the aggregated network bandwidth does not differ from that of various numbers of concurrent executions; hence, they did not identify any network resource isolation mechanisms amongst function invocations. For disk IO operation, they used the *dd* system command to identify performance. However, we believe that the IO performance in the FaaS runtime needs deeper investigation, because it will become more important as the serverless computing approach is broadened to data-intensive applications.

In order to better understand the network and disk IO performance of FaaS using container technology, we performed thorough experiments that heavily utilize disk and network resources with practical application workloads. From the results of our experiments and subsequent analyses, we made the following observations:

- I/O device micro-benchmarks that exclusively stress specific hardware do not provide an accurate estimate of realistic network and disk performance
- Quantitative evaluations reveal that the configuration of memory allocation for functions makes a difference in network and disk IO performance, even though they are not enforced by a service provider
- A response time and cost evaluation revealed that allocating more resources to function run-time does not always produce a proportional gain in performance, and the cost can increase significantly
- Fine-grained measurement of IO overhead while running data-intensive applications on public FaaS reveals the consequence of not limiting IO devices, which is unfavourable to function run-times with larger RAM configurations.

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 maximum memory size allocation to network and disk IO 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. Furthermore, we could observe that not limiting IO device performance can adversely affect CPU performance especially when the RAM configuration is small. Although the current FaaS execution model is popular because of the abstraction of complex resource provisioning and scheduling, users, when deploying data-intensive applications in an FaaS environment, should be conscious of the impact of concurrent executions on a single host to avoid unexpected performance.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes details about serverless computing and function execution environments. Section 4 presents thorough evaluation of a function service regarding network and disk IO resource performance, and Section 6 concludes the paper by proposing future work.

2 Related Work

Many cloud service vendors provide FaaS execution models, including 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 [5] is another open source implementation of FaaS. As a container orchestration tool, Kubernetes [6] 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 being actively developed, including OpenFaaS, Kubeless, and Knative. These applications have the potential to contribute significantly to expanding 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. [3] and Lee et. al. [4] compared public function execution environments. They 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 and disk IO resources 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. [7] proposed running data analysis jobs on Flint using a serverless environment. Ishakian et. al. [8] ran a deep neural network model inference engine on an FaaS platform and compared its performance with those of dedicated machines. Feng et. al. [9] proposed an algorithm to run DNN training tasks using FaaS. Pywren [10] ran large-scale linear algebra and machine learning jobs on AWS Lambda. Kim et. al. [11] 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. Kim et. al. [12, 13] proposed a suite of data-intensive FaaS workloads to expand the application scenario of serverless computing. There is a recent trend in the literature towards expanding FaaS applications to data-heavy jobs [2], and it has been demonstrated that the performance impacts from network and IO device resources can be significant.

There are attempts to overcome limitations of FaaS to support diverse applications. Pocket [14] and Locus [15] proposed external ephemeral storage service by using high performance key-value storage engines, such as NVMe SSD or Redis. Crucial [16] proposed a shared-state machine that acts as a global variable storage service for Java run-time. Though the current FaaS applications are limited to embarrassingly parallel ones, with efforts from academia, we believe that FaaS will gradually support data-intensive MapReduce [17] type applications natively and that network and disk IO performance will become crucial.

3 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

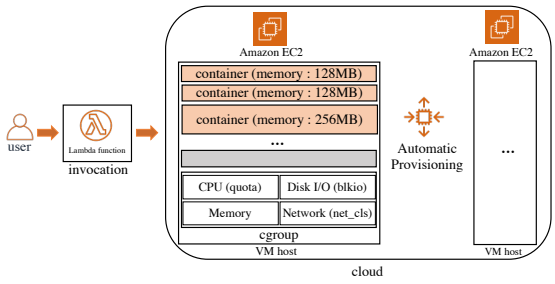


Fig. 1: FaaS environment using container technology

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 computation environments, many cloud service vendors provide a serverless function service. With this service, a user implements the core functionality of an application and registers it with the 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 the 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, the service provider uses container technology that has relatively lower management overhead and start time than virtualization [18].

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 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 an 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, the *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, [3, 4] determined that CPU and memory resources are allocated proportionally to user payments, but these authors did not produce an in-depth discussion of network and disk IO resources allocation and isolation. An FaaS working scenario is shown in Figure 1.

4 IO Performance Characteristics of FaaS

Prior work [3,4] has focused on evaluation and comparison among many FaaS providers mainly on memory allocation and the corresponding CPU performance. As FaaS applications become more data-friendly, it appears that the impact from the IO performance becomes crucial for providing reliable and predictable performance. In order to understand the impact of various FaaS configurations to the disk and network performance, we thoroughly investigate the performance quantitatively with respect to memory allocation and concurrent executions. We use FunctionBench [12] to evaluate the IO performance of FaaS with realistic applications. The FunctionBench is composed of micro and application benchmarks. The micro-benchmarks consist of a few system commands that stress either CPU, memory, disk, or network exclusively. The application benchmarks contain many data-intensive scenarios, such as image/video processing, text data featurization (tf-idf) followed by sentiment analysis, DNN serving, and MapReduce tasks [13]. The authors provide source codes that are ready to be executed for public cloud services AWS, Azure, and Google, and we use them without modification.

4.1 Characteristics of Network Performance

To exclusively measure the performance of network resources of FaaS, we used the *iperf3* system call from the FunctionBench [12] 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 [2]. The Lambda function run-time works as a client where the IP address and port of the server are 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 the Amazon Fine Food Review text dataset[1] with *Python2.7* and the *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 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 download time does not consider function parallelism by adding up download time from each container, and it determines billing for the download service.

4.1.1 Impact of Memory Size Configuration

We first evaluated the network bandwidth available with AWS Lambda with different memory size configurations by using the *iperf3* system command. In Figure 2, the horizontal 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, with data obtained using the *iperf3 -R* option. Previous work [3] has investigated the bandwidth available in a function execution environment in the upload case, and their findings match the values given in Figure 2.

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

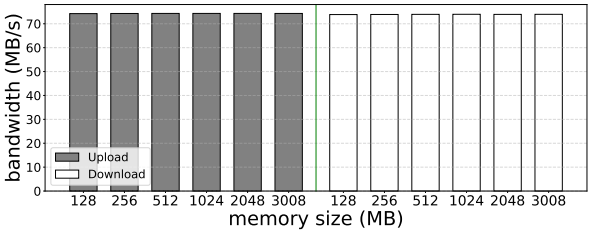


Fig. 2: Network bandwidth evaluation with iperf3

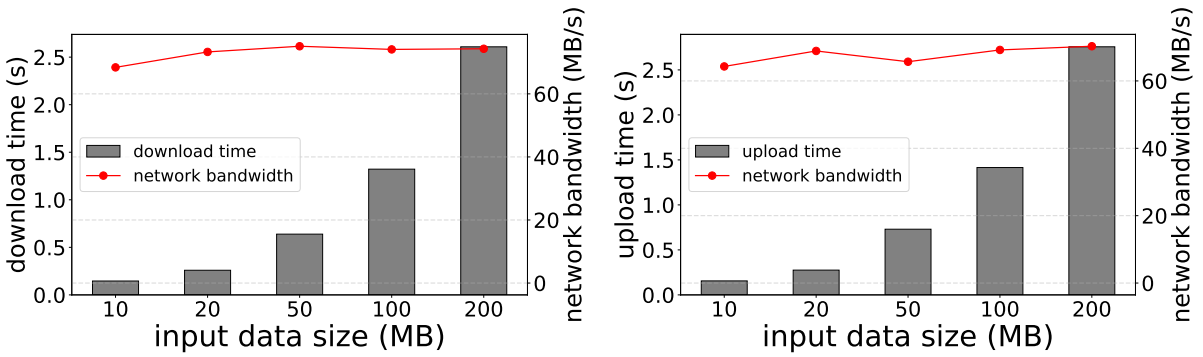
From these experimental results, we can see that the service provider does not provide different levels of network quality based on the configured maximum memory size.

Evaluation with the *iperf3* system command provides an easy way to investigate the network bandwidth available for a function run-time, but it does not represent a realistic scenario for data applications that may download or upload files from a shared block storage. To understand the network performance of an 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 (Figure 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, which 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 effects of unavoidable long-tail latency in a cloud environment [19]. 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.

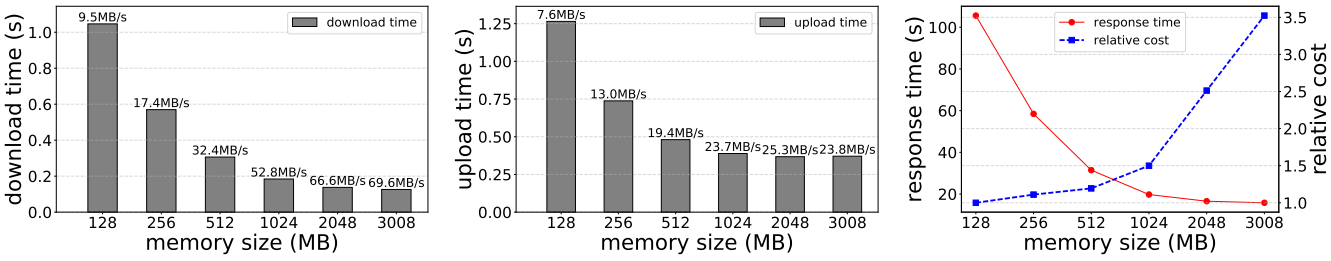
Figure 4b shows the per-function upload time (median value) and available bandwidth. Similar to the results shown in Figure 4a, the available upload bandwidth increases as the allocated memory size increases, and these findings are also contrary to the results shown in Figure 2. From the experimental 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 use of external data sources 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 its proportional CPU usage quota to configured RAM size negatively affect 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 mem-



(a) Download time (b) Upload time

Fig. 3: Response time with different file size



(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

ory size; the primary vertical axis shows response time, with values shown on the solid line with circle markers; and 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. The figure shows 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%.

4.1.2 Impact of Concurrent Execution

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

Concurrent Execution Evaluation Methodology: In order to decide if function executions were conducted on the same host, Wang et. al. [3] 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 the same host. We ran the concurrency tests on AWS Lambda as much as possible, but the function placement is not deterministic, which entailed 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 [3]. 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*

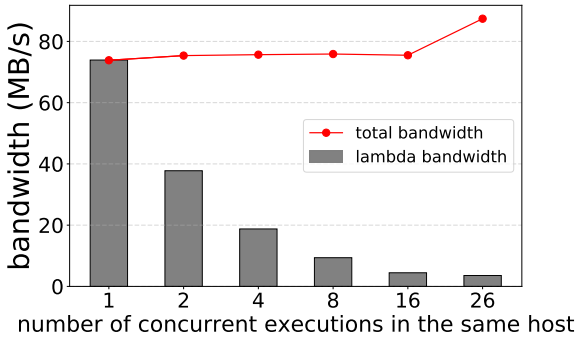


Fig. 5: Available network bandwidth with concurrent function execution measured with iperf3

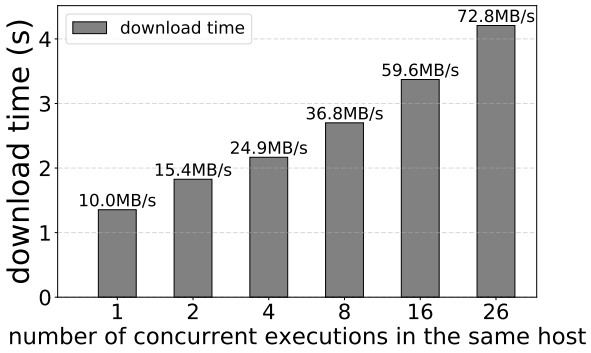


Fig. 6: download response time and aggregated bandwidth

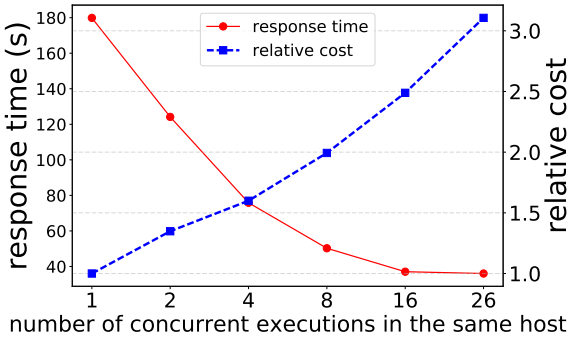


Fig. 7: response time and cost relation

and *cpu.cfs_quota_us*. After fixing the memory size, we set the CPU ratio as 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 a function execution environment using EC2 instances and performed experiment. It shows very similar result pattern with the Lambda environment. With the confirmation, we perform experiments on the EC2 and Docker environments when we cannot ensure that many functions run on the same host.

Figure 5 shows the network bandwidth 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 bandwidth amongst the functions whose value is marked on the vertical axis; and the solid line with round markers shows the aggregated network bandwidth 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, as evidenced by the figure: as the number of concurrent executions increases, the allocated bandwidth per execution decreases, but the total aggregated bandwidth remains 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 in size, from Amazon S3; and the numeric value on each bar shows the aggregated network bandwidth 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 with the number of concurrent executions shown in the horizontal axis. 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 a response time and cost perspective.

4.2 Characteristics of File IO Performance

For most of current FaaS applications, using local storage to store intermediate result is rare [2], and public FaaS systems provide limited local storage for FaaS runtimes. However, data-intensive applications can generate huge intermediate outcomes that are generally temporary stored on a local machine's storage [20, 21], and the importance of disk IO performance will become significant as big-data applications become deployed on FaaS. To better understand the disk IO performance of public FaaS systems for data-oriented applications, we thoroughly investigate bandwidth and latency with respect to the different configured memory sizes and concurrencies.

Wang et. al. [3] lightly evaluated IO performance of public FaaS using *dd* system command. However, such a micro-benchmark cannot capture complex access patterns of realistic data-oriented applications [22], and in this study we use a suite of IO benchmarks provided in the FunctionBench [13] that uses *fread/fwrite* in the Python for both sequential and random access.

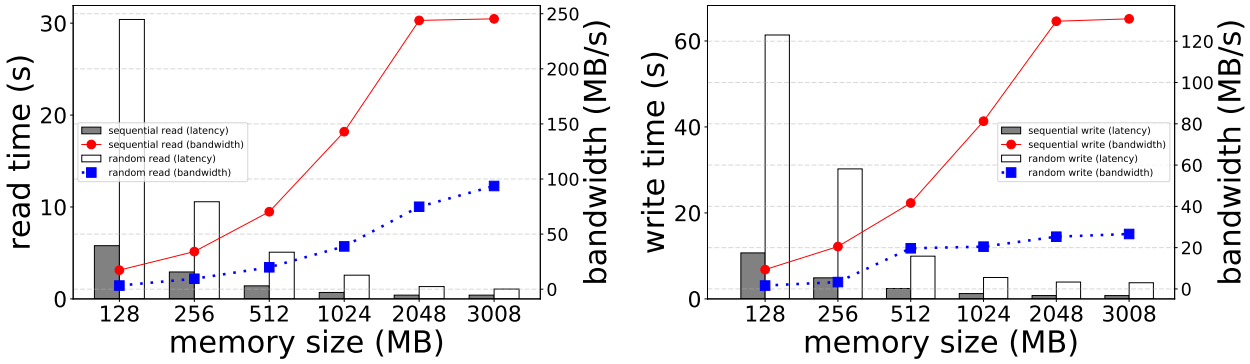
4.2.1 Impact of Memory Size Configuration

Figure 8 shows the response time and corresponding bandwidth for processing 100MB files from AWS Lambda. We conducted the same set of experiments with different file sizes, and it shows similar result. Figure 8a shows the read performance, and Figure 8b shows the write performance. In each figure, the gray bar and white bar show the latency of sequential and random access, respectively, in the primary vertical axis. The solid line with round markers and dotted line with square markers show the available bandwidth of sequential and random access, respectively, in the secondary vertical axis. For both read and write operations, it shows proportional performance as we increase the memory allocation. When a function run-time has over 2GB of memory allocated, the available disk IO bandwidth reaches the maximum. When the allocated RAM is small, the available disk bandwidth is very little. Thus, users should be cautious in the function run-time configuration, even if a function requires a small amount of memory, as it can affect various aspect of resources.

Similar to the network resource, it is known that public FaaS vendors do not differentiate disk IO resource allocation [3]. However, controlling available processing capacity by limiting the RAM size and CPU quota proportionally impacts the disk IO performance indirectly, and users experience different disk IO performance proportional to the allocated memory. In the experiments, the evaluation of read happens right after a write operation has finished. Thus, it is likely that the read operation performance has benefit from using cached output from the write operation. In realistic FaaS applications, this scenario is more likely than accessing files stored in disk without caching, because the current FaaS execution model is stateless, and a randomly assigned function run-time does not store a file locally for function execution.

4.2.2 Impact of Concurrent Execution

The disk IO bandwidth can get impact from concurrent function executions on a same host. To quantitatively measure the impact, we perform the *fread/fwrite* experiments on a controlled environment presented in Section 4.1.2. The result of write operation is shown in Figure 9. The horizontal axis shows the degree of concurrent executions on a host. The primary vertical axis shows the latency to complete an operation of 100MB input file, and the secondary vertical axis shows the corresponding available bandwidth. The gray bar indicates the latency of sequential write, and the white bar shows the latency of random write. The allocated memory size of each function run-time is set as 128MB. The solid line with round markers shows the bandwidth of sequential write, and the dotted line with square markers shows that of random write. In the experiment, we could not complete random write operation within 1,000 seconds when the number of concurrent executions is 26, and the white bar and blue square marker in the setting are missing. Both random and sequential write show degradation due to the contention in a same host. Similar to the network resources, users should be conscious about the performance degradation of disk IO operation due to the concurrent execution even if they cannot control the function placement.



(a) File read (b) File write

Fig. 8: Response time and available bandwidth with different memory size (AWS Lambda)

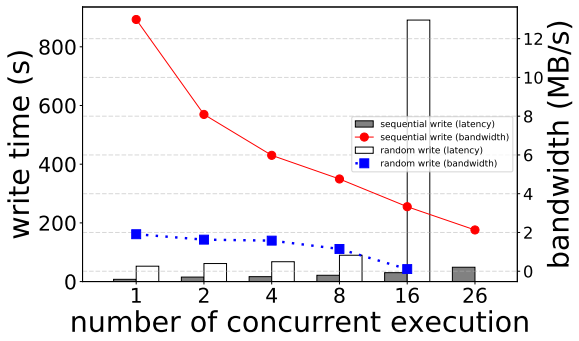


Fig. 9: Available disk bandwidth with concurrent function execution

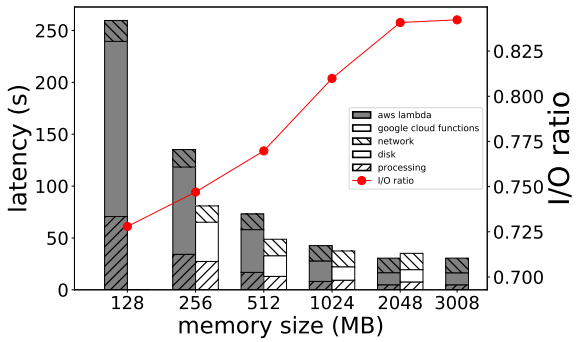


Fig. 10: Image processing workloads with AWS Lambda and Google Cloud Functions

5 IO Performance of Public FaaS

It is important to understand the impact from IO resources to overall performance of data-intensive applications on public FaaS systems. To represent data analytics scenarios, we ran MapReduce and Image Processing applications from FunctionBench [12, 13] and present compute, disk IO, and network time on AWS Lambda and Google Cloud Function service while allocating different memory sizes.

Image Processing : In the image processing workload, we assume that a batch of images (five images of 1MB each) is stored in a shared cloud object storage. A FaaS function fetches images and apply several effects, such as flip, rotate, blurring, contour filtering, resize, and gray scale, using Python Pillow library. As there are many effects applied for each image, the intermediate outcome should be stored in a local disk of each function run-time. After all the tasks complete, the outcomes stored in a local storage is uploaded to a shared cloud object storage.

Figure 10 shows the latency of Image Processing with different memory size configured. The dark gray bar shows AWS Lambda performance, and the white bar shows Google cloud Functions. Missing bars mean that the given experiment could not complete with the configured memory size. We separate each bar into network (upper left diagonal), disk IO (plain), and processing (right upper diagonal) time that is shown in the primary vertical axis. The secondary vertical axis shows the ratio of network and disk IO to the total completion time. As more resources are allocated, we can see that both AWS and Google function show improved performance. The ratio of network and disk IO keep increasing as the configured memory size increases. Current FaaS systems do not provide isolation of IO resources proportional to the configured memory size [3], and this experiment result implies that handling IO device overheads can negatively impact the CPU performance especially when the memory configuration and the corresponding CPU allocation are small. As more memory and CPU quota are allocated to function runtime, the improvement ratio of compute is much higher

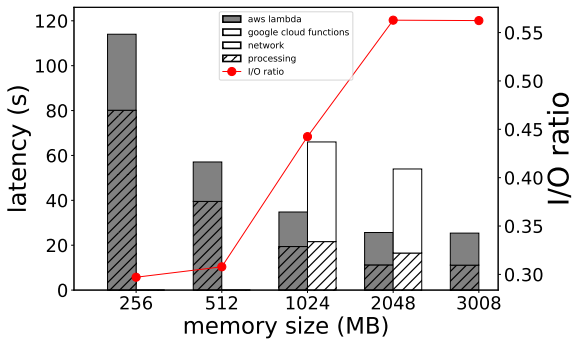


Fig. 11: MapReduce workloads with AWS Lambda and Google Cloud Functions

than that of IO device. The larger memory and CPU allocation result in enough capacity for processing the compute job. However there is not much room for improvement for IO devices as they do not get restrictions based on the configured memory size.

MapReduce : The MapReduce workload implements parallel execution of Map and Reduce primitives [17] using FaaS run-times. The workload receives the Wikipedia dataset in the programming language category, and it counts the number of occurrences of a programming language to infer the popularity. In the experiment, we prepared 1GB of total input dataset that is partitioned into ten blocks that are stored in a shared cloud object storage service. Map tasks are invoked in parallel with the number of blocks. Each Map task fetches an input block from a storage and count the number of target programming language. The outputs from Map tasks are uploaded to a shared object storage, and a Reduce task aggregates Map outputs.

Figure 11 shows the MapReduce experiment result. The gray bars show the performance from AWS Lambda, and white bars shows results from Google Cloud Functions. In each bar, right upper diagonal parts express the compute time, and the plain parts represent network latency. The MapReduce task in the experiment does not involve a local disk IO operation. Similar to the Image Processing experiment result, larger memory size results in shorter execution time, as it has more CPU allocated. The IO ratio increases as more memory is allocated to function run-times because no limitation of IO resource results in better relative performance when the configured memory size is small.

From the micro-benchmark experiment result that relies on data download, upload, file write, and read, we could observe that disk and IO performance improve proportionally with the configured memory size, though explicit IO resource limitation is not enforced. However, with experiments of realistic applications, we could uncover that the ratio of IO changes significantly with different configured memory size; the impact from IO device becomes noticeable as the configured memory and CPU allocation become large, and it eventually hinders performance improvement even with a larger memory and CPU allocation in FaaS. Because such behavior is not easily expectable from users' perspective, public FaaS providers should provide a mechanism to let users control degree of IO resource allocation. The needs will become more significant as data-intensive applications are deployed using FaaS systems.

6 Conclusions

In this paper, we presented the results of our investigations into the performance of network and disk IO resources in data-intensive FaaS applications. First, we measured network performance by using the *iperf3* micro-benchmark and accessing a file shared on a block storage service. We confirmed that the AWS Lambda service does not differentiate the network resource allocation in proportion to configured memory size, but we also found out 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. For disk IO resources, we confirmed that the different memory size configuration results in proportional random and sequential read/write bandwidth. We envision the importance of network and disk IO resource performance isolation for realistic data-intensive FaaS applications (e.g., MapReduce and image processing). Not providing different level of IO resource limitation results in favorable IO performance with the lower memory size configuration. Thus, as larger memory size is allocated for a function run-time, the impact from IO device becomes more significant in a way that is not straightforward for end-users. To make the FaaS system widen application scenarios to data-heavy ones, the service level of IO resources should be more distinguishable as users set FaaS run-times with a different degree of expectations.

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. E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019. [Online]. Available: http://arxiv.org/abs/1902.03383
2. 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
3. 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
4. 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
5. 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*.
6. D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 2015.
7. Y. Kim and J. Lin, "Serverless data analytics with flint," in *IEEE CLOUD 2018*.
8. V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform."
9. L. Feng, P. Kudva, D. D. Silva, and J. Hu, "Exploring serverless computing for neural network training," in *IEEE CLOUD 2018*.
10. 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*.
11. J. Kim, M. Son, and K. Lee, "Mpec: Distributed matrix multiplication performance modeling on a scale-out cloud environment for data mining jobs," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.
12. J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, pp. 502–504.
13. J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: ACM, 2019.
14. A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic
15. Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu
16. D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures," in *ACM/IFIP Middleware'19*, 2019, p. To appear.
17. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264
18. W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *IEEE ISPASS 2015*.
19. 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*.
20. A. S. Foundation, "Apache hadoop," 2004. [Online]. Available: http://hadoop.apache.org/
21. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28.
22. J. Kim, J. Park, and K. Lee, "Network resource isolation in serverless cloud function service," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, June 2019, pp. 182–187.