Completion Time Prediction of Open Source FaaS Functions

David Balla, Markosz Maliosz, and Csaba Simon

Abstract—**Function as a Service (FaaS) is the latest stage of application virtualization in the cloud. It enables to deploy small code pieces – functions – in the cloud. FaaS focuses on event-driven functions in response to triggers from different sources. The functions run in ephemeral virtual environments. This means that the user is charged on the basis of the time the function is busy serving the invocation requests. With the advent of Industry 4.0 the need has arisen to run applications on Edge Computing nodes. FaaS is a promising solution for serving industrial applications that require predictable latency while meeting the demands of edge computing, which operates on a limited resource base. Therefore, knowing the completion time of the invocation requests is of key importance.**

In this paper, we introduce a function runtime design for open-source FaaS implementations that achieves a lower deviation in request completion times compared to default runtimes by regulating the function's access to host CPU cores. We present the implementation details of our proposed function runtime design for Python, Go and Node.js. We also introduce a simulation framework that is able to estimate the completion time distribution of the incoming invocation requests. We validate the results of our simulation framework using real measurement data.

Index Terms—**Communications Society, IEEE, IEEEtran, journal, LATEX, paper, template.**

I. Introduction

The granularity of application virtualization technologies is continuously increasing in the past decades, virtual machines have been partially replaced by light-weight container virtualization solutions, however, in both cases complete applications are hosted in the virtualized environment. In contrast to Virtual Machines (VMs) and containers, FaaS makes it available to host only a single building block of a distributed application in the virtualized environment. FaaS implementations are based on virtualization solutions, such as containers and lightweight VMs, such as microVMs [1], [2] or unikernels [3]. The FaaS framework spins up an instance for the incoming requests and keeps it running for a given time period and they are evicted if no requests are sent to them during this time. FaaS shows a change in the paradigm, a new way of application orchestration, rather than introducing a new implementation for virtualization. Functions can be organized into function chains, where one function invokes the next, thus implementing a complex application.

Functions are invoked for incoming events and they use the compute resources only during the execution. The corresponding billing model is based on the time the function keeps the compute resources allocated to serve the incoming requests. In contrast, using VMs and containers, the user pays for the up-time of the virtualized environment even if the hosted applications are idle.

The importance of FaaS can be seen by the fact that not only the major public cloud providers have implemented their FaaS platforms (e.g. Google Cloud, Amazon Web Services, Microsoft Azure, Alibaba Cloud), but the open source community has also embraced the technology. Several open-source FaaS implementations are available on GitHub, such as OpenFaaS [4], Fission [5], Kubeless [6] or Nuclio [7].

FaaS systems provide predefined function runtimes to execute the functions. Functions implemented by the users are encapsulated into the provided function runtimes, helping the users to focus on the business logic and sparing the manual integration work. However, these runtimes can majorly influence the performance of the functions.

Knowing the completion time of the requests has a key importance, as the billing model of FaaS systems is based on the combination of time the function is busy serving the requests as well as the allocated resources for the functions. Cloud providers can define different strategies for cost calculation. In AWS Lambda [8] and Microsoft Azure Functions [9], users are charged based on a per-millisecond rate, whereas users of Google Cloud Functions [10] and Alibaba Function Compute [11] are billed in 100-millisecond increments. The amount of allocated resources determines the price of each time increment. Therefore we exclusively focus on the completion time of the function invocations, as the cost of an invocation can be derived from the completion time and the amount of allocated resources.

In this paper we introduce a function runtime design that is able to provide stable completion times by regulating the function's access to the host CPU cores, thereby preventing bottlenecks and achieving higher QoS. We also introduce a method to estimate the completion time of function invocations. We present our results by using compute intensive functions implemented in Python, Go and Node.js.

We base our work on our previous paper [12] in which we have introduced our function runtime design as well as an algorithm that is able to estimate the completion time distribution of the function invocations. However, in [12] we have implemented our function runtime design exclusively in Python, and we designed our algorithm to estimate the completion time of functions that support parallel request processing. In this paper, we introduce the implementation details and

David Balla, Markosz Maliosz, Csaba Simon are with the High Speed Networks Laboratory Department of Telecommunications and Artificial Intelligence Budapest University of Technology and Economics, Budapest, Hungary. (E-mail: balla/maliosz/simon@tmit.bme.hu)

DOI: 10.36244/ICJ.2025.2.7

performance characteristics of our function runtime design for Python, Go and Node.js. We also introduce an additional algorithm that is designed to estimate the completion time distributions of invocation requests sent to function instances supporting sequential request processing. The introduced algorithms are taking into account the resource assignment strategies as well as the expected load that is sent to the examined function instance.

In this paper we focus on open source FaaS systems, as in this case we have access to the source code of the whole ecosystem in contrast to public cloud providers' implementations.

Open source FaaS systems are almost exclusively implemented on top of Kubernetes. Kubernetes is an open source cloud orchestration platform for containers. The basic architecture of an open source FaaS system is depicted in Fig. 1. The gateway component is the entry point of the FaaS system. It works as a proxy and directs the requests to the appropriate functions. Function instances run in containers that are encapsulated by Kubernetes pods. Function instances of the same type are hidden by a Kubernetes service. The gateway component addresses the service of the given function and forwards the requests to the actual function instances.

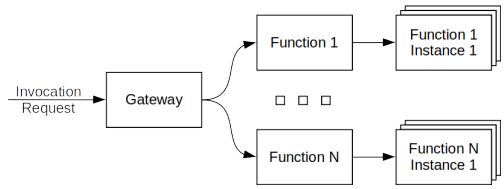

Fig. 1. Generic open source FaaS architecture

This paper is structured as follows. In Section II, we review the related work. Section III introduces our proposed function run-time design, while in Section IV we discuss the details of our function completion time prediction methods. In Sections V and VI we introduce our measurement environment and discuss our findings. Finally, Section VII concludes the paper.

II. Related Work

The benchmarking of FaaS implementations has a key importance, as different implementations have different performance characteristics [13], [14], [15]. M. Grambow et al. introduce BeFaaS [16], a framework to benchmark the major public providers' as well as open source FaaS platforms. BeFaaS comes with a built-in benchmark scenario that is an e-commerce application. BeFaaS can be extended by new benchmarks and can be created and added to the framework by using the BeFaaS programming library. In addition to showing the completion time of the functions, BeFaaS provides a drill-down analysis that helps to understand the particularities of the behavior of the benchmarked FaaS frameworks. P. Maissen et al. implemented FaaSdom, a tool to benchmark major public FaaS frameworks. FaaSdom [18] supports the benchmarking of different language runtimes, as well as it provides the ability to calculate the costs of the users. SeBS [17] provides a framework to benchmark public FaaS platforms. SeBS takes the following metrics into consideration, CPU and Memmory utilization, Response time, Code Size and Network IO. Costs can also be benchmarked by SeBS. The authors discuss the additional costs for the user that originates from the billing model of a particular FaaS provider.

Simulation frameworks can significantly reduce the costs of developing applications in a FaaS ecosystem, while also identifying factors that influence performance. SimFaaS [19] is a simulation framework for FaaS systems that enables the prediction of several performance metrics of a FaaS system, such as average response time, the probability of cold starts, and the average number of function instances. The authors validate the results of SimFaaS by comparing it to real usage data from AWS Lambda. M. Hanaforoosh et al. introduce MFS [20], a serverless FaaS simulator based on Apache Open Whisk. MFS calculates the reports of several metrics on top of the ones supported by SimFaaS, e.g., the number of functions that can or cannot be scheduled on any of the physical machines, the number of requests that can or cannot finish before a given deadline, and unlike SimFaaS, it handles containers and functions separately, reporting the number of containers used. SimLess [21] is a framework to simulate function choreographies in major public FaaS providers' ecosystems. SimLess considers various overheads such as network, concurrency and, authentication when estimating the round-trip time of the functions. J. Manner et al. introduces a methodology [22] to enable the comparison of the local and the cloud function execution and to map the local profiling data to the cloud platform. Their effort can significantly reduce development time as developers can work with their local tools that they are familiar with.

Using ephemeral, event-driven FaaS functions is a promising approach to implement services hosted on top of Edge computing devices, as such environments are equipped with a limited amount of compute resources. LambdaContSim [23] is a simulator designed to evaluate custom strategies to place functions on edge nodes to meet various requirements. It measures different metrics, such as placement success and failure, energy consumption, and service time. F. Filippini et al. present a simulation framework [24] for evaluating load-balancing algorithms in decentralized FaaS environments. The framework assesses performance using metrics such as success rate, power consumption, and the number of rejected requests.

Based on this review, our contribution differs from previous efforts, as our work focuses on the behavior of function runtimes under various compute resource assignment strategies, by using our proposed simulation framework.

III. Function Runtimes

Function runtimes are key components of FaaS systems. Function runtimes are encapsulating the user-defined functions, and by this lifting the burden of the integration of the functions to the distributed FaaS environment off the shoulders of the users. Function runtimes are implented as lightweight web-servers, that wrap the user-defined function. They also define an endpoint to respond to periodic health-check messages initiated by the FaaS framework. We have examined runtimes of several open source FaaS systems, namely

OpenFaaS [26], Fission [27], and Kubeless [28]. According to our investigations, the majority of the supported runtimes are capable of parallel request processing by starting a new worker thread for each of the incoming requests. However, these runtimes do not necessarily take into consideration the number of physical CPU cores that the host computer has. In case of starting more worker threads than the number of host CPU cores can lead to variable completion times. To overcome this issue, we propose a function runtime, that sets the limit the number of concurrently processed requests to the number of host CPU cores [12]. In the following, we introduce function runtimes for Python, Go and Node.js.

A. Python

Python runtimes are implemented by using different libraries in case of the examined FaaS frameworks. OpenFaaS and Fission implement their Python runtimes by using the Flask web framework, while Kubeless is using the Bottle library.

For the sake of simplicity, in the case of Python, we implemented our proposed runtime by using the Flask library. Flask supports parallel request processing by starting new Python threads or forking new Python processes. Python threads are sharing the Python interpreter's Global Interpreter Lock (GIL), which can lead to a serious performance bottleneck in the case of CPU intensive tasks [29]. To overcome this issue, we implemented our runtime by using Python's multiprocessing library, that starts a new Python interpreter with the user-defined function for each of the incoming requests. In Fig. 2, we show our measurement results performed on a computer having 16 physical CPU cores. We performed our measurements by sending function invocation requests to the examined function instance at different concurrency levels, which were integer multiples of the number of physical CPU cores. It can be seen that the tail latency is lower when limiting the number of worker threads to the number of available CPU cores.

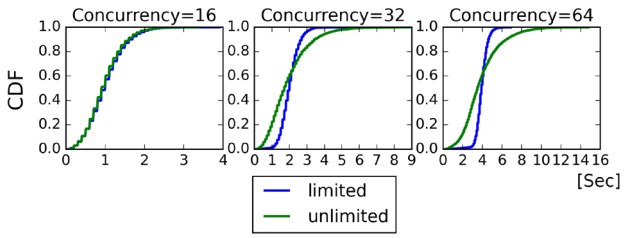

Fig. 2. Completion time differences - Limited vs. Unlimited number of worker threads - Python

B. Go

The Go runtimes of the examined open source FaaS implementations are all using the built-in HTTP server implemented in the Net library of the Go language that starts a new worker thread (Goroutine) for each of the incoming requests.

We extended the basic Go runtime design by adding a counting semaphore (implemented by Go buffered channels) to limit the number of simultaneously running worker threads. Fig. 3 depicts the performance difference of request processing between the proposed and the default function runtime designs. It can be seen that the proposed runtime design shows lower tail completion time values when the requests concurrency exceeds the number of CPU cores.

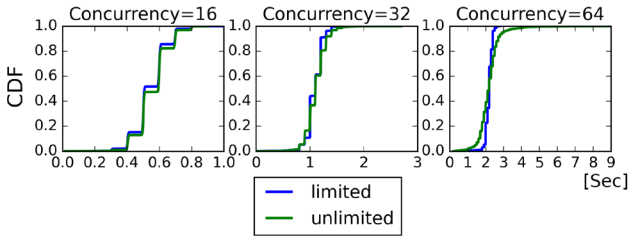

Fig. 3. Completion time differences - Limited vs. Unlimited number of worker threads - Go

C. Node.js

The Node.js runtimes are single threaded and are not capable of using multiple CPU cores simultaneously to concurrently serve CPU intensive tasks. By exploiting the Cluster or Worker threads library, we can achieve parallel processing of the incoming requests with Node.js. We have implemented a Node.js function runtime, that enables parallel request processing, by using the Worker threads library. To limit the number of concurrently running worker threads, we implemented our runtime to store all the incoming requests in a queue and use a counter to limit the number of requests processed concurrently. However, Node.js adds an extra layer of scheduling as it uses an event loop that allows to run tasks in an asynchronous way, therefore, the behavior of this function runtime differs from that runtimes where threads are directly scheduled by the operating system's task scheduler. Fig. 4 shows the results of using our proposed Node.js runtime with setting limited and unlimited number of worker threads to process the invocation requests.

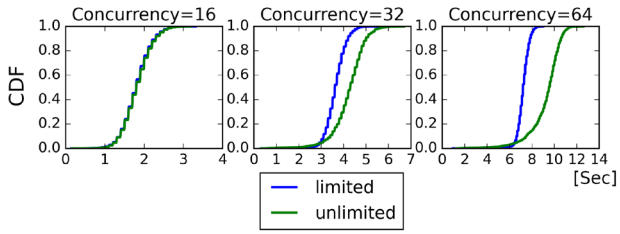

Fig. 4. Completion time differences - Limited vs. Unlimited number of worker threads - Node.js

IV. COMPLETION TIME PREDICTION

Being able to predict the completion time of function invocations is crucial. It makes available for the users to estimate their costs as the billing model of FaaS is based on the time the function spends on serving the incoming requests,

as well as it helps to design latency-sensitive workflows in a FaaS ecosystem.

In open source FaaS systems the function instances are running in containers encapsulated in Kubernetes Pods. Containerized applications are assigned to different Linux Control Groups (cgroups). Cgroups control the associated amount of resources to the application, e.g., CPU, memory, network. In open source FaaS frameworks, the users can specify the assigned amount of CPU and memory resources to the functions. These values are configured in the cgroups of the function instance's container.

The amount of CPU resources assigned to the container determines the amount of CPU time the applications in the container can consume in a single scheduling window. This value can be higher than 100% in the case of multi-core systems. For example, if the user assigns 200% CPU resources to the containerized application that has 4 threads and is running on a computer with 4 cores, then each of the threads can run in 50% of the scheduling window. The Completely Fair Scheduler (CFS) of Linux takes into account the CPU resources allocated for the applications through the cgroups.

We propose two algorithms to estimate the completion time distribution of the function invocations. Both algorithms model the CFS scheduler in a simplified way, by using round-robin scheduling. Cgroups are organized in a hierarchical way, which results in a hierarchical way of scheduling. However, the round-robin scheduling can be used, as the cgroup of the function container has no sub-cgroups, and the function container hosts only the user-defined function, that is, a small and simple single or multi-threaded application.

A. Multi-Threaded Runtimes

Our proposed algorithm for multi-threaded function runtimes [12] takes into account the number of CPU cores, the time (translated from CPU ticks) required for the function to run on the CPU to serve the invocation request (onCPU time), the assigned amount of CPU resources, and the concurrency of the incoming load. The on-CPU time is not necessarily equal to the response time, as the worker thread can be preempted while serving the request. The algorithm assumes that the function instance uses our proposed function runtime and that the user specifies the desired amount of CPU resources to be allocated to the function.

Our proposed function runtimes have a main thread that starts a new worker thread for each of the incoming requests until the number of worker threads reaches the CPU core limit. The proposed algorithm distributes the worker threads over the CPU cores. The algorithm runs the tasks in the scheduling window and decreases the tasks' onCPU time by the minimum time-slice as well as the CPU quota for the current scheduling window. If the CPU quota has decreased to less than the minimum time-slice, the algorithm starts a new scheduling window (see Algorithm 1).

According to our investigations, the minimum time-slice is 4 milliseconds, while the length of the scheduling window is 100ms. Our estimate for the time required to fork a new worker thread is 1 ms. If a task does not need the whole time-slice to finish, the rest of the time-slice can be used by the rest of the worker threads.

Algorithm 1: Completion time prediction for simultaneous request processing [12]

```
Function AddNewTask(startTime):
    actualWindow -= TimeOfFork
    tasks.append({timeOnCpu, startTime, endTime=-1})
    parallelTasks -= 1
    numTasks += 1

Function RunTask(task):
    task.timeOnCPU -= minTimeSlice
    actualWindow -= minTimeSlice

loops=1
timeFrame = 100ms
actualWindow = CPUThrottle
minTimeSlice = 4ms
timeOfFork = 1ms
numTasks = 0
parallelTasks = numCPUCores
tasks, finishedTasks = []
while len(FinishedTasks) != N do
    while actualWindow > minTimeSlice do
        task = tasks.GetNextTask()
        if task is mainTask then
            if numTasks < concurrency then
                // first batch of requests
                if parallelTasks < numCPUs then
                    AddNewTask(0)
                end
            else
                // at least one task of the first batch
                   has finished
                if newTasks > 0 and parallelTasks < numCPUs then
                    AddNewTask(finishedTasks[-newTasks].endTime+
                      ElapsedTimeTillNewRequest)
                    newTasks -= 1
                end
            end
        else
            RunTask(task)
            if task.timeOnCPU <= 0 then
                task.endTime = loops*timeFrame
                SaveTaskRuntime(task)
                finishedTasks.append(Task)
                tasks.Remove(Task)
                parallelTasks -= 1
                newTasks += 1
            end
        end
    end
    loops++
    actualWindow = cpuQuota
end
```

B. Single Threaded Runtimes

Some of the function runtimes only support sequential request processing, therefore, we propose another algorithm to estimate the completion time distribution of such functions. Initially, the algorithm starts tasks equal to the concurrency level and stores them in a list and starts to run the first task. The simulator decreases the on-CPU time and the CPU quota for the actual scheduling window. If the on-CPU time of the task is less than or equal to zero, then the simulator starts to run a new task as well as it adds a new task to the end of the task list. If the CPU quota reaches a value that is less than the minimum time slice, the simulator starts a new scheduling window. If the CPU quota is not zero but is less than the minimum time slice, it can be used in the next scheduling window. (see Algorithm 2)

Algorithm 2: Completion time prediction for sequential request processing

```
currTime = 0
for i in [1..Concurrency] do
    tasks.Append(Task(startTime=currTime))
end
actualWindow = CPUThrottle
task = tasks.popFirst()
while len(FinishedTasks) != N do
    while task.onCPU > 0 and actualWindow > 0 do
        task.onCPU -= minRuntime
        actualWindow -= minRuntime
        currTime += minRuntime
    end
    if task.onCPU <= 0 then
        FinishedTasks.append(currTime - task.startTime)
        task = tasks.popFirst()
        tasks.Append(Task(startTime = currTime))
    end
    if actualWindow <= min_runtime then
        if actualWindow > 0 then
            actualWindow = CPUThrottle + actualWindow
        else
            actualWindow = CPUThrottle
            currTime += schedWindow - CPUThrottle
        end
    end
end
```

V. TESTBED

A. Test environment

We deployed our measurement environments by installing a two-node Kubernetes cluster on Cloudlab [25] c220g1 compute nodes equipped with 2 Intel E5-2630 v3 8-core and 128 GB RAM and 10 Gb network interfaces. The Linux CFS operates with time values instead of CPU ticks. However, if the CPU frequency scaling is turned on, the number of CPU ticks and the number of operations under a given time period are undefined. Therefore, we turned off the CPU frequency scaling and locked the CPU frequency to 2.2GHz. We have also turned off the logical CPU cores given by the HyperThreading capability, to avoid situations when tasks are scheduled to the same physical core.

B. Test Functions

We implemented our test functions in Python, Go, and Node.js programming languages. Each of the functions calculates the estimated value of π by using the Leibniz formula. The Leibniz formula takes a parameter that sets the accuracy of the estimated value of π. This parameter is the input value of the function. The higher the value of the parameter, the more accurate the estimated value of π. For our measurements, we selected the input values from a normal distribution.

C. Load Generator

For our measurements, we used Hey, an HTTP load generator that is able to generate HTTP requests with a given concurrency level. To maintain a stable concurrency level, Hey initially starts client threads equal to the desired concurrency level. First, all the clients send out their requests. After that, a client can only send a new request once a response has been received for the current request.

Hey is only able to work with the same request content during load generation. Therefore, we modified the codebase of Hey to be able to send different input values to the functions.

VI. EVALUATION

We have implemented the proposed algorithms as function completion time simulator software modules. We validate the accuracy of the proposed simulators by comparing their outputs with real measurement results. We performed our measurements by using function runtimes that support parallel and sequential request processing.

To show the ability of our simulator software to predict the completion time distribution even in the case of variable input, we performed our measurements by selecting the input variables from a normal distribution. In this case, the configuration of the simulator requires two measurements. To be able to calculate the values of a normal distribution, we need to know the mean and deviation values. Thus, we performed two measurements using the median and deviation of the input values, to get the corresponding onCPU times.

The onCPU times can be acquired by using the Linux Perf tool. However, running Perf requires system administrator rights that are not necessarily available in all cases. In such situations, we can estimate the onCPU time by the response time of the function invocation as shown by eq. (1). When using eq. (1) to estimate the onCPU value, we suppose that the user-defined function does not start any further threads. However, this calculated value includes the additional latency of the network transport.

$$OnCPU \approx \lfloor \frac{Comp.T}{Sch.W} \rfloor * CPU_{fn} + Comp.T - \lfloor \frac{Comp.T}{Sch.W} \rfloor * Sch.W \quad (1)$$

Where:

- $Sch.W$ = 100ms, scheduling window,
- $CPU_{fn} \in [1..100]$, amount of CPU resources assigned to the function,
- $Comp.T$: completion time of a single function invocation.

A. Parallel Request Processing

To show the efficiency of our simulator, we present our results of different scenarios covering cases with different CPU and concurrency settings. The performance of the examined language runtimes are different, therefore, we adjusted the function invocation parameters accordingly.

Fig. 5 shows our measured and simulated results in the case of runtimes that are able to process multiple requests simultaneously. It can be seen that the measured and simulated values are very close to each other. In Table I we summarized the absolute differences between the measured and simulated results, for each of the language runtimes, related to the median and the 95th percentile.

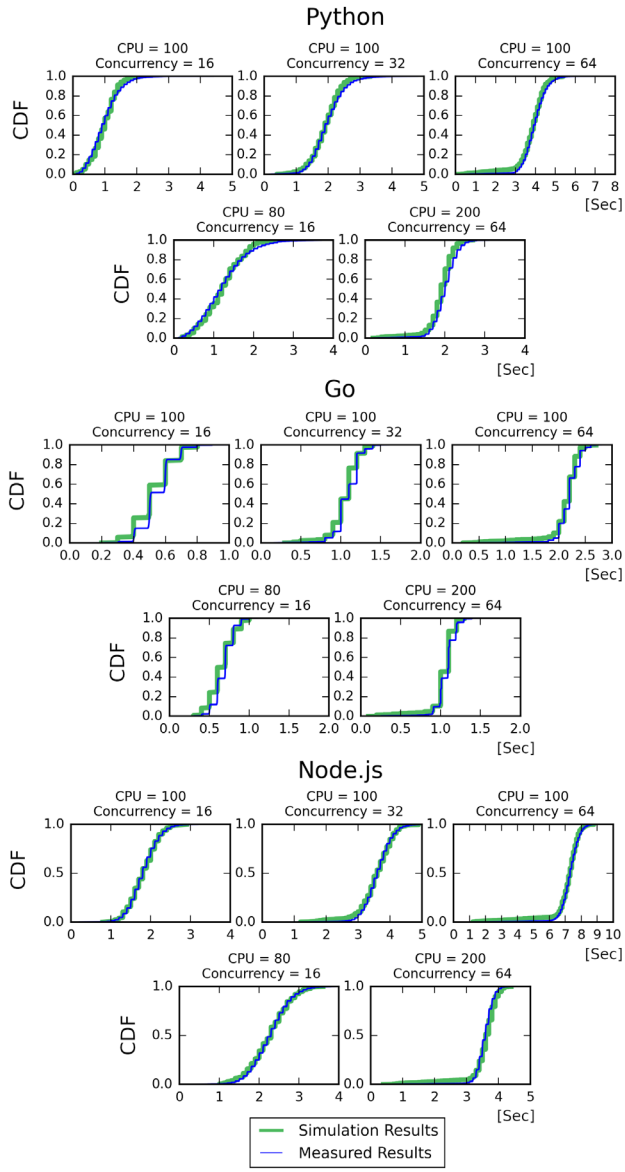

Fig. 5. Real vs Simulation completion time distribution in case of parallel request processing

TABLE I
MEDIAN AND 95^{th} %ILE ERRORS [SEC]
PARALLEL REQUEST PROCESSING

CPU	Conc.	Percentile	Python	Go	Node.js
80%	16	50th	0.0047	0.0038	0.0038
		95th	0.1974	0.0023	0.0078
100%	16	50th	0.0056	0.0081	0.0002
		95th	0.1987	0.0000	0.0022
	32	50th	0.0949	0.0999	0.0007
		95th	0.1007	0.0003	0.0008
	64	50th	0.0089	0.0000	0.0002
		95th	0.1953	0.0998	0.0014
200%	64	50th	0.0998	0.0003	0.1046
		95th	0.1907	0.0101	0.1104

B. *Sequential Request Processing*

Fig. 6 shows the results of our measured and simulated completion time distributions for each of the examined function runtimes. In this case, we selected the input values to generate jobs of which onCPU times are less than the time the function is scheduled to run in a given scheduling window. This leads to a scenario where some of the function invocation completion times are significantly longer than the rest, as it can happen that the function is preempted by the scheduler while it is processing an invocation request. In this case, not only the completion time of the preempted request is influenced, but all the completion times of the requests that arrived after the preempted request. We experienced an undefined behavior in the case of Node.js over the request concurrency level of 32, therefore, in Fig. 6 we only show our results for the concurrency levels of 16 and 32. We suppose that the experienced undefined behavior over the concurrency level of 32 is due to the asynchronous event-loop of the Node.js runtime. We show the absolute differences between the measured and simulation results related to the median and 95^{th} percentile in Table II.

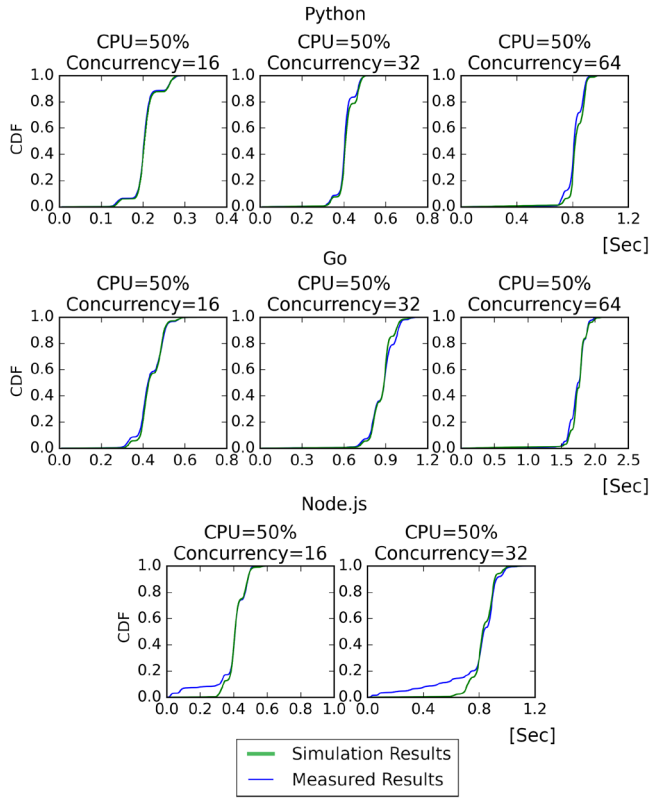

Fig. 6. Real vs Simulation completion time distribution in case of sequential request processing

C. *Differences of Sequential and Parallel Request Processing*

We investigated the completion time differences in the cases where function runtimes supporting parallel request processing

TABLE II
MEDIAN AND 95^{th} %ILE ERRORS [SEC]
SEQUENTIAL REQUEST PROCESSING

CPU	Conc.	Percentile	Python	Go	Node.js
50%	16	50th	0.0005	0.0079	0.0031
		95th	0.0022	0.0083	0.0052
	32	50th	0.0010	0.0015	0.0172
		95th	0.0019	0.0156	0.0384
	64	50th	0.0039	0.0240	-
		95th	0.0070	0.0015	-

and those where sequential processing runtimes were used. In Fig. 7 we show the differences between the completion time distributions related to the examined cases. In this scenario, we used our Python and Go language runtimes and assigned 100% of CPU resources to them. It can be seen that, in the case of Python, using a function runtime that supports simultaneous request processing results in higher completion time values in general. This phenomenon can be explained by the additional time that is caused by the starting a new Python interpreter for each of the incoming requests. In case of Go, the completion time distributions for the two cases are very close to each other. Also, our parallel processing Go runtime starts threads that are not as expensive as starting new processes. Parallel function runtimes can gain extra performance with more than 100% of assigned CPU resources.

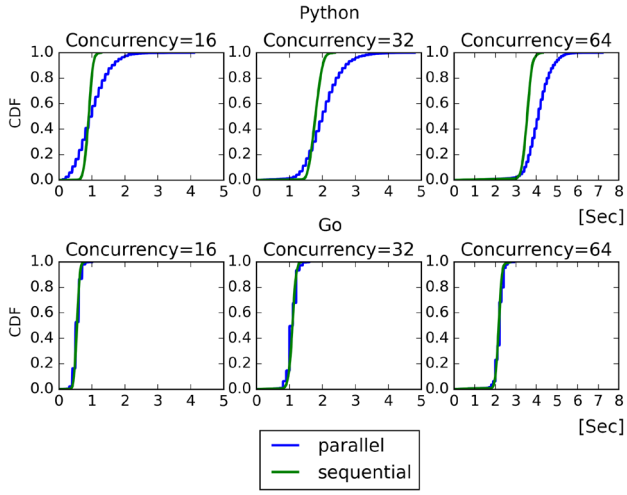

Fig. 7. Distribution of function completion times in case of parallel and sequential request processing runtimes

D. Variable CPU frequency

We performed our measurements by turning off the CPU frequency scaler. However, in a real-life scenario, the lack of enabling the CPU frequency scaler results in high energy consumption even if the compute node is idle. Therefore, we investigated the effects of the CPU frequency scaling on the accuracy of our measurements. We performed our measurements with the CPU frequency scaling turned on and off, as well as investigated the effects of CPU utilization generated by other CPU heavy loads. For our measurements, we used our function implemented in Go, with 50% of CPU assigned, to also have an idle period in the scheduling window. In this idle period the frequency scaler will scale down the CPU frequency short after the application is scheduled out even if the application remains on the same CPU core. To mitigate any unpredictable behavior on the software side, we generated load using a fixed input value instead of varying the input parameters. Fig. 8 shows the results of our measurements. It can be seen that the results are influenced by the variable CPU frequency caused by the CPU frequency scaler. In the case of a fully utilized compute node, the CPU clock runs on the maximal frequency, which results in more predictable completion times that are very close to the results of the scenario where the CPU frequency scaling is turned off. We can also observe that the background load does not influence the performance of the function. This can be explained by the assigned amount of CPU resources that is guaranteed for the function instance at the level of the scheduler of the OS.

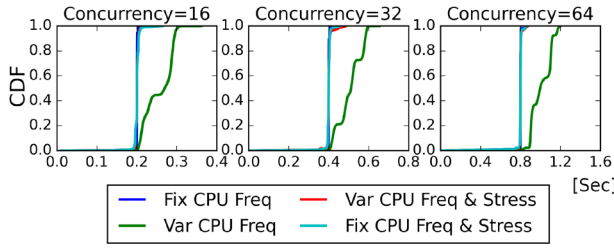

Fig. 8. Effects of CPU frequency scaling on function completion times

VII. CONCLUSION

In this paper we introduced a function runtime design that is able to process multiple requests at the same time but it limits the number of simultaneously processed requests to the number of available CPU cores. We showed that using our runtime design the tail completion times can be reduced significantly. We implemented the proposed function runtime design for Python, Go and Node.js.

We have also proposed an algorithm to predict the completion time distribution of the invocation requests sent to a FaaS function that are using our proposed function runtimes. However, not all the function runtimes support simultaneous request processing. Therefore, we proposed another algorithm that is able to forecast the completion time distribution of invocation requests sent to functions that are only capable of sequential request processing. We implemented both proposed algorithms as function runtime simulators and validated their results using real measurement data from scenarios with varying inbound request concurrency and CPU resource allocations for the functions.

Knowing the function completion time in advance can highly facilitate the design phase of software architectures based on FaaS considering the billing model of FaaS. FaaS operates with ephemeral function instances; therefore, FaaS is a promising model to implement services hosted by edge computing nodes. Knowing the completion times for a given load, our work helps engineers to better utilize the limited resource base of edge computing devices.

REFERENCES

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., & Popa, D. M. (2020). Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)* (pp. 419–434).

[2] Randazzo, A., & Tinnirello, I. (2019, October). Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (pp. 209–214). IEEE.

[3] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., ... & Huici, F. (2017, October). My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 218–233).

[4] https://openfaas.com

[5] https://fission.io/

[6] https://github.com/vmware-archive/kubeless

[7] https://nuclio.io/

[8] AWS Lambda pricing, https://aws.amazon.com/lambda/pricing

[9] Azure Functions pricing, https://azure.microsoft.com/pricing/details/functions/

[10] Pricing Overview, https://cloud.google.com/functions/pricing-overview

[11] Function Compute Pricing, https://www.alibabacloud.com/product/function-compute/pricing

[12] Balla, D., Maliosz, M., & Simon, C. (2021, November). Estimating function completion time distribution in open source FaaS. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)* (pp. 65–71). IEEE.

[13] Motta, M. A. C., Carvalho, L. R., Rosa, M. J. F., & Araujo, A. P. F. (2022). Comparison of faas platform performance in private clouds. *Proceedings of the 12th CLOSER*, 109–120.

[14] Ma, P., Shi, P., & Yi, G. (2023, October). Feature and Performance Comparison of FaaS Platforms. In *2023 IEEE 14th International Conference on Software Engineering and Service Science (ICSESS)* (pp. 239–243). IEEE.

[15] Barcelona-Pons, D., & García-López, P. (2021). Benchmarking parallelism in FaaS platforms. *Future Generation Computer Systems*, 124, 268–284.

[16] Grambow, M., Pfandzelter, T., Burchard, L., Schubert, C., Zhao, M., & Bermbach, D. (2021, October). Befaas: An application-centric benchmarking framework for faas platforms. In *2021 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 1–8). IEEE.

[17] Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., & Hoefler, T. (2021, November). Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference* (pp. 64–78).

[18] Maissen, P., Felber, P., Kropf, P., & Schiavoni, V. (2020, July). Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM international conference on distributed and event-based systems* (pp. 73–84).

[19] Mahmoudi, N., & Khazaei, H. (2021). Simfaas: A performance simulator for serverless computing platforms. *arXiv preprint arXiv:2102.08904*.

[20] Hanaforoosh, M., Ashtiani, M., & Azgomi, M. A. (2023, May). MFS: A serverless FaaS simulator. In *2023 9th International Conference on Web Research (ICWR)* (pp. 81–86). IEEE.

[21] Ristov, S., Hautz, M., Hollaus, C., & Prodan, R. (2022, November). SimLess: simulate serverless workflows and their twins and siblings in federated FaaS. In *Proceedings of the 13th Symposium on Cloud Computing* (pp. 323–339).

[22] Manner, J., Endreß, M., Böhm, S., & Wirtz, G. (2021, September). Optimizing cloud function configuration via local simulations. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)* (pp. 168–178). IEEE.

[23] Matricardi, A., Bocci, A., Forti, S., & Brogi, A. (2023,August). Simulating FaaS Orchestrations In The Cloud-Edge Continuum. In *Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge* (pp. 19–26).

[24] Filippini, F., Calmi, N., Cavenaghi, L., Petriglia, E., Savi, M., & Ciavotta, M. (2024, June). Analysis and Evaluation of Load Management Strategies in a Decentralized FaaS Environment: A Simulation-Based Framework. In *Proceedings of the 1st Workshop on Serverless at the Edge* (pp. 1–8).

[25] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., ... & Mishra, P. (2019). The design and operation of CloudLab. In *2019 USENIX annual technical conference (USENIX ATC 19)* (pp. 1–14).

[26] https://github.com/openfaas/templates

[27] https://github.com/fission/environments

[28] https://github.com/vmware-archive/runtimes

[29] Python GlobalInterpreterLock https://wiki.python.org/moin/GlobalInterpreterLock

David Balla is a PhD candidate at the Budapest University of Technology and Economics (BME), Department of Telecommunications and Artificial Intelligence, High Speed Networks Laboratory. His research interests include low-latency and real-time networking and cloud computing solutions. His current research focuses on cloud-native software architectures and real-time FaaS systems.

Markosz Maliosz is an associate professor at the Budapest University of Technology and Economics (BME), Department of Telecommunications and Artificial Intelligence, High Speed Networks Laboratory. He received his PhD (2006) and MSc (1998) degrees in Computer Science in the field of communication systems from BME. He worked as guest researcher and consultant at telecommunication in the areas of network performance evaluation and time-sensitive networks. His research interests include network architectures and design, optimization techniques and traffic engineering, his current research activity focuses on network virtualization and optimization, and cloud networking. He is member of the Scientific Association for Infocommunications (HTE), Hungary.

Csaba Simon obtained his PhD degree at Budapest University of Technology and Economics, Department of Telecommunications and Artificial Intelligence and he is working at the same Department since 2001. His research interests are mostly related to 5G and 6G Systems, virtualization, cloud native applications and network and service management.