Implications of Serverless Compute Framework on End Users

Ishaan Thakur
it233@cornell.edu
Cornell University
School of Electrical and Computer Engineering
Ithaca, New York, USA

Yixie Chen
yc2636@cornell.edu
Cornell University
School of Electrical and Computer Engineering
Ithaca, New York, USA

***Abstract*—Serverless computing is a cloud technology model that is increasingly becoming popular among developers. Serverless cloud vendors provide fine-grained resource provisioning that scales dynamically in response to user demand. This serverless paradigm is followed by Function-as-a-Service (FaaS) models, in which the users provide their application logic as a series of functions that are performed in response to a user / system-generated trigger event. The functions executed in a serverless model are meant to be short-lived, stateless applications that are wrapped into containers / VMs, which are very cheap to launch. However, due to the short span of execution for these functions, there are several overheads introduced owing to spinning up of new containers as well as inter-process interference that may impact the performance of this compute model. In this paper, we explore how varying invocation patterns and concurrency of serverless functions can impact latency for various function executions. We then dive into exploring several performance and resource tradeoffs between function invocation over a serverless platform as opposed to native execution.**

Keywords-Cloud Computing, Serverless, Apache OpenWhisk, Datacenter, Function as a Service, Latency, Profiling, QoS

I. Introduction

Serverless Compute is a popular cloud service model provided by several large I.T. firms like Amazon, Google, and Microsoft under numerous brand names. Although these services might have varying degree of features, the underlying concept is nearly identical: serverless aims to accomplish auto-scaling while delivering more inexpensive compute services by transforming processing to a pay-as-you-go model.

In contrast to standard cloud computing notions, serverless computing is distinguished by the fact that the infrastructure and platforms on which the services are delivered are not visible to the users. Customers are only concerned with the desired functionality of their application under this method, and the rest is left to the service provider. This has motivated the emergence of Function as a service (FaaS) compute model, which provides end users the ability to only develop code for the application logic while the cloud providers handle provisioning of the servers, resource management and so on.

This concept has been successfully implemented in the business world with the introduction of Lambda [1] by Amazon in November 2014, followed by Microsoft Azure Function App [3] and Google's Cloud Function [4]. In 2016, IBM introduced OpenWhisk [12], a serverless platform that allows users to locally deploy and evaluate their functions' execution. This has sparked several research conducted on advantages [7] and drawbacks [9] of this service.

However, most of the research conducted has been centered around platform management aspects of serverless framework such as application development, scheduling policies and so on. In this paper, we decided to dive into the impact serverless may have on the hardware (servers) on which the application runs on. In order to mimic this behavior, we decided to deploy a local instance of a FaaS platform (OpenWhisk) and analyze performance of serverless functions with varying invocation patterns and tradeoffs that developers may experience in comparison to native execution.

II. Background and Theory

The inception of serverless has given birth to the field of FaaS. This has sparked a debate over the tradeoffs one may experience if they switch from traditional cloud models such as IaaS (Infrastructure as a Service) and PaaS (Platform as a Service). The differences that the customers may experience in terms of application development are discussed in the following section.

A. Serverless vs Traditional Cloud Models

Briefly, the serverless framework differs from other serverless cloud models in the following ways:

a) Code Development without the need for managing different resources

Instead of requesting resources, the user would write the logic of the application, and the cloud vendors would automatically allocate resources to run the respective code on.

b) Pay-as-you go model

Rather than dependent on a dimension of the fundamental cloud platform, such as size and number of VMs assigned, billing is based on the aspects related to the execution of functions, such as execution time.

c) Separation of Compute and Storage

Storage and compute scale independently, and they are provided and charged separately. In general, the computing is stateless, and the storage is provided by a separate cloud service.

B. Impact of Serverless on several stakeholders

Unlike native functions, it is hard to find a suitable performance criteria for serverless compute since it has a lot of stakeholders with different expectations. For local deployment, low execution time is a great performance criteria. However, in the serverless compute model, the stakeholders' demands can compete with each other. There are three main stakeholders, namely the cloud service vendors, application developers and the customers. For investment purposes, the vendors desire high throughput per server hence, concurrently run applications in order to have higher return on investment for the infrastructure. The developers, on the other hand, want low execution time in order to reduce computation costs on their end. Finally, the customers prefer low latency for running their applications smoothly.

Due to the different expectations of each of these key members, an economic quandary is raised for the cloud vendor and leaves us questions about the effects of the pricing model for serverless computing on the developer and end user.

III. Apache OpenWhisk Platform

Our group required insight to understand system components and separate software from architectural overheads in order to thoroughly assess implications of serverless framework. This was possible through Apache OpenWhisk, an open-source FaaS platform that is the basis for all serverless functionalities offered by IBM.

A. *Overview of OpenWhisk Platform*

This serverless platform can run functions in response to events at any scale and can automatically handle services, infrastructure, and application scalability using Docker Containers [6]. Apart from this, OpenWhisk also offers a REST API based CLI (command line interface) called *"wsk"* that is able to provide catalog services, manage packaging and offers several popular container deployment choices.

Because its components are built using containers, OpenWhisk may be deployed and configured on a wide range of platforms. This enables it to offer a wide range of deployment choices, both locally and inside a cloud architecture. The deployment of OpenWhisk could be done on several container frameworks such as Mesos [2], Kubernetes [10], OpenShift [11] and Compose [5].

The OpenWhisk programming model is divided into three different modules: Action, Trigger, and Rules. Action is a stateless function that executes arbitrary pieces of code; Trigger is a class of events that can originate from a number of sources (called Events); and Rules allows one to associate a Trigger with an Action. The interaction between these components is shown in Figure 1. Moreover, OpenWhisk allows grouping of actions together to make a sequence. Currently, this serverless platform supports several programming languages for Action development such as Go, Python, JavaScript, Java, Swift and many more.

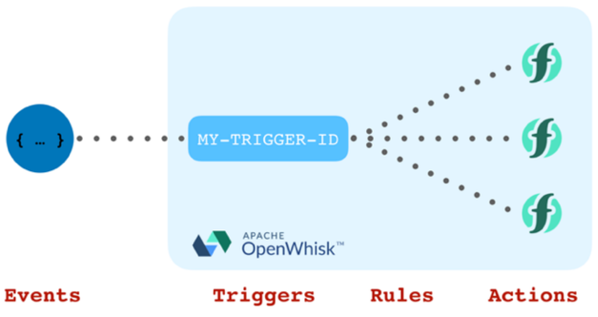


Figure 1: Dependency of Triggers, Rules and Actions in OpenWhisk.

Next, let us look over the OpenWhisk architecture and how the information flows through it. For our example, let us assume that there is a temperature sensor that generates a trigger event responsible for sending an HTTP request to our OpenWhisk function when temperature rises above a certain value. The steps taken for the message to flow inside OpenWhisk is outlined below:

- The request to the corresponding OpenWhisk action travels to an NGINX reverse proxy that is responsible for forwarding the message to the controller component.
- The controller is then responsible for authentication as well as authorization of received requests via a call to a CouchDB database instance. This could involve checking whether the HTTP message has the correct Connection String for the corresponding OpenWhisk instance.
- After the message is authenticated, the controller verifies the state of the invokers.
- Next, the load balancer component of the controller chooses a healthy invoker to execute the corresponding action on.
- The HTTP message is then passed from the Controller to an Invoker instance via an Apache Kafka publish subscribe messaging system.
- After receiving the message, the Invoker executes the action as a docker container, whose configuration could be defined by the developer or generated as a language-specific container that packages the developer's code.
- After execution of the action completes, OpenWhisk checkpoints the results of the function calls in the built-in CouchDB instance. In addition to the temperature information, the system also saves metadata, execution time and several other invocation results.

In Figure 2, we can see how OpenWhisk's internal message processing flow is structured.

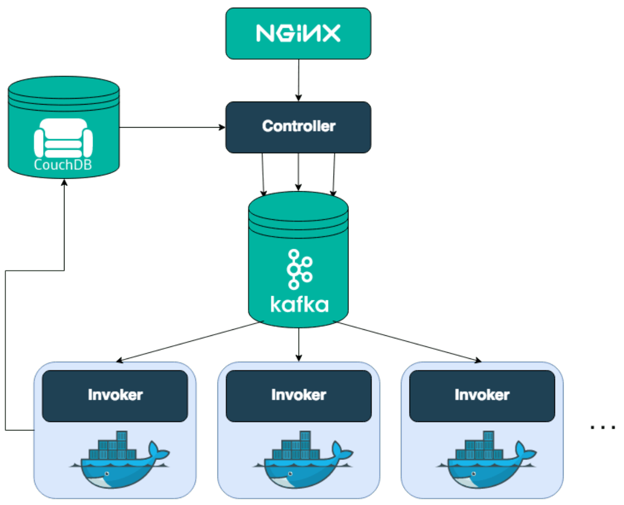


Figure 2: Apache OpenWhisk's Internal Message Processing Flow.

B. *Deployment of OpenWhisk*

The OpenWhisk instance was locally deployed on a Linux device. The process involved initially cloning the OpenWhisk package from GitHub and building the local deployment instance of this platform via Gradle using Java 10. After building the application, an executable jar file was created that could then be used for launching an OpenWhisk instance. The recommended method for launching a CouchDB instance is via Ansible. However, due to package dependency issues with installation of OpenWhisk's built-in CouchDB instance, we decided to install a separate CouchDB module that could then connect with our OpenWhisk platform and be used for logging in the execution results. After launching the OpenWhisk application, we were not only able to write actions that could then be hosted on our deployed OpenWhisk platform, but we also had access to an online playground UI that allowed easy debugging for single file action instances.

IV. FaaSProfiler

Now, one of the challenges we faced in analyzing our locally deployed OpenWhisk instance was a reliable tool for testing and profiling several hardware-level and execution results of actions that would run on this serverless platform.

For this, we came across an Open-Source Software called FaaSProfiler [8] that was developed by Princeton University's Parallel Research Group for analyzing hardware architecture implications of serverless functions [13].

As discussed in [13], FaaSProfiler stands out due to its user-friendly interface that allows developers to profile serverless function behavior that can not only have varying invocation patterns and activity windows but also invoke a large number of individual functions in a single instance. Moreover, the FaaSprofiler software provides developers with a vast amount of profiling data such as execution time, latency and wait time along with resource profiling metrics such as branch misses, page faults and so on. The next section discusses the internal working of FaaSProfiler.

```
{
    "test_name": "example_test_for_paper",
    "test_duration_in_seconds": 60,
    "random_seed": 120,
    "blocking_cli": false,
    "instances":{
        "instance1":{
            "application": "app_name",
            "distribution": "Poisson",
            "rate": 25,
            "activity_window": [5, 10]
        }
    },
    "perf_monitoring":{
    "runtime_script":
"monitoring/RunTimeMonitoring.sh",
        "post_script": null
    }
}
```

Figure 3: JSON Input File for Description of Action Invocation Pattern via FaaSProfiler.

V. Development with FaaSProfiler

The first step involved cloning the software from the GitHub repository [8] and running an initialization script to install all required frameworks / packages for development. In the FaaSProfiler, a user can specify features of an action to be executed using a JSON file. An instance of a typical JSON file input is shown in Figure 3. As we can see, there are fields in the beginning to specify the name of the test, duration the test will run for, a random seed and an option to specify whether all actions will execute in a blocking or non-blocking manner. Next, the user can specify a bunch of actions that can have varying distribution and invocation rates along with the activity window these functions will execute in.

Moreover, there is an option regarding the script we want to run that will monitor different parameters of the execution of these actions. For our evaluation we modified the provided RuntimeMonitoring.sh file to record perf monitoring and profiling data that would then be saved in the CouchDB instance of OpenWhisk. Apart from this, there is an option in the config file to include a postscript for monitoring other features of the FaaS platform in addition to the profiling function specified in the runtime script field.

After one has defined the JSON file comprising the desired execution pattern of actions, they would then have to specify the URL path for the locally deployed CouchDB instance in the Workload Invoker function of FaaSProfiler. The next step would involve saving the authentication details for the OpenWhisk in an environment variable via the wsk CLI, which the FaaSprofiler can automatically infer from. One can then call the Workload Invoker function that will execute the specified actions by sending an HTTP request to the OpenWhisk, which will finally checkpoint the results after execution of actions have finished to the deployed CouchDB instance. Next, the Workload Analyzer function can be invoked for sending an HTTP request to the CouchDB instance for retrieving the invocation data for plotting the results as well as storing the profiling data as a pickle file for future analysis, such as comparative plotting between different execution cycles.

In Figure 4, we can see how the FaaSProfiler can interact with an OpenWhisk instance by sending an HTTP request.

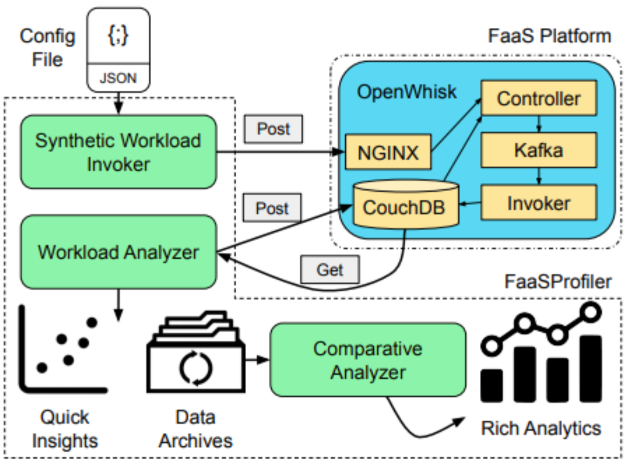


Figure 4: Interaction of FaaSProfiler with an OpenWhisk Instance.

VI. Testing Design

For the next step, we created several functions to evaluate the performance of the OpenWhisk platform as well as identifying the overheads. These functions include CouchDBAction, SmartConvAction, MoodAction, LeftPad, and APIAction. These are used as benchmark applications for testing purposes. CouchDBAction, LeftPad, and APIAction are written in NodeJS and deployed via Serverless framework [14] on our locally deployed OpenWhisk instance. On the other hand, SmartConvAction and MoodAction are developed in Python and deployed via wsk CLI. The tasks each of these functions accomplish are listed in Table 1.

Given these five benchmark functions, we then conducted four major tests to observe the tradeoffs for running FaaS functions with OpenWhisk as opposed to native execution. There are three major parameters that we modified for each test to obtain results under different circumstances. These include action distribution pattern, invocation rate and activity window.

Test A is responsible for comparing the results between different distribution patterns of invocation for serverless functions. We first deploy all five action instances concurrently on OpenWhisk with Poisson distribution and an invocation rate of 1 corresponding to an under-invoked scenario while keeping the same activity window. Then, we increase the invocation rate for all the functions to 50 while recording the performance. This process is repeated for Uniform distribution with the same parameters. By comparing the results we obtained from these two different distributions, we successfully observed key differences regarding the performance of serverless deployment.

Test B is conducted to observe how invocation rates impact the serverless performance. In this section, we initialized all five functions to run concurrently using OpenWhisk with the same activity windows and Uniform distribution but varying invocation rates to 1 ips, 20 ips, and 100 ips, respectively. These rates represent the under-invoked, balanced and over-invoked status of the system. By recording the performance pattern for each of them, we noticed several interesting facts regarding the performance of serverless.

Test C is used to compare the performance differences between running applications using serverless and native execution. We ran all five benchmark functions separately with the same activity windows and Uniform distribution but different invocation rates corresponding to under-invoked, balanced, and over-invoked scenarios using OpenWhisk at first. The same functions were then locally invoked with the same invocation rates. The execution time was measured for these native invocations via time command in Linux. Notice that by running the functions separately, we were able to eliminate the overhead that may be introduced to programming languages.

Test D focuses on the CPU performance impact the benchmark functions have when they run using OpenWhisk. Using the Linux Perf tool, we analyzed the page-faults, branch-misses, and context switches while running the functions with different invocation rates (1, 20 and 100 ips). These functions were then executed natively with the same parameters for comparison purposes.

All the tests were performed on an Ubuntu 18.04 machine with Intel i7, 16GB DDR3L, 1600 GHz RAM. The results and observations are discussed in the next section.

Application	Description of the Application	Programming Language
CouchDBAction	Queries JSON data from a separate CouchDB instance and displays the results.	NodeJS
SmartConvAction	Renders an input Markdown text to HTML format.	Python
MoodAction	Performs Sentiment Analysis of a given text. This involves outputting results of whether the given text is positive, negative or neutral.	Python
LeftPad	Padding the input text by a specified amount from the left.	NodeJS
APIAction	Displays the JSON data obtained from a get request to an API containing several statistical information about the population of the US.	NodeJS

Table 1: The list and description of Testing Applications that are used as benchmarks.

VII. Results

In this section, we analyze the results we obtained from the proposed testing and discuss the intuitions behind the given phenomenon.

A. Test A

In the first section, we analyzed the impact two different distributions could have on the Serverless compute platform. For this we invoked all the five functions concurrently twice, the first time with all of them having a Poisson distribution and the second time with Uniform distribution. Each of these concurrent invocations happened with all the functions invocation rate varying from 1, 20 and 50 ips and an activity window of [0, 5], [10, 15] and [15, 20] seconds respectively. The results are shown in Figure 5 and 6.

We observed that the Poisson distribution tends to be much more variable with shorter run time. This could be because of a randomized invocation pattern, thus resulting in functions to be executed outside of the grace period for a container and hence, causing in more cold start events. Moreover, the latency for this distribution is linearly proportional to the invocation rates. As the rates increase, the latency for function execution also increases owing to the queuing effect of multiple functions wanting to execute in a limited number of containers.

Uniform distribution, on the other hand, is far more stabilized compared to the Poisson. However, we can see a relatively high latency at the beginning of Uniform's invocation. The initiation time for rate 1 is much higher owing to spinning up of new containers for the cold start event of new functions. Moreover, an increase in the invocation rate results in a higher warm start event of containers for new functions, thus, resulting in reduced initiation time. We also notice throughout the experiment that the Uniform distribution has smaller latency and lesser variability in run time. This could be attributed to Uniform distribution's constant burst of function invocation in a particular activity window thus, resulting in functions to execute in containers that are still warm.

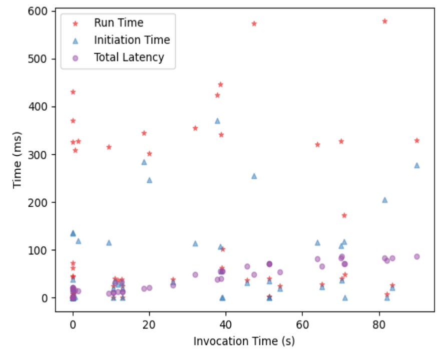


Figure 5. Runtime, Total Latency Results for Poisson Distribution.

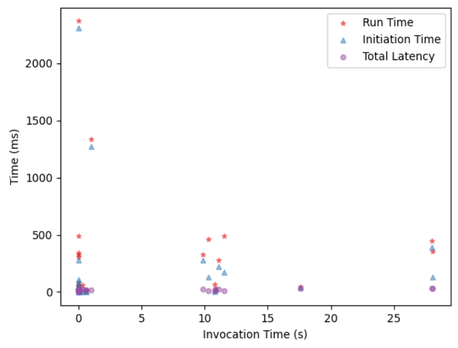


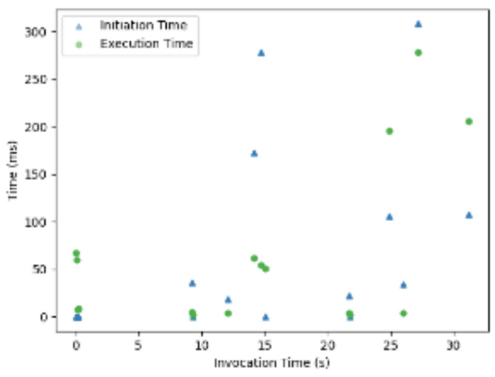
Figure 6. Runtime, Total Latency Results for Uniform Distribution.

B. Test B

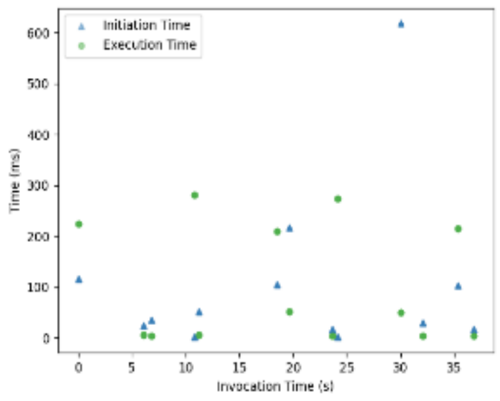
In this section, we evaluate the influence invocation rates have on serverless computing. All five functions were concurrently invoked in a uniform distribution pattern with varying invocation rates to 1, 20, and 100 ips. We recorded execution time as our performance metric for an activity window from 0 to 20 seconds. Execution time can be defined as the time taken for a function to run after it has been loaded into a container. It is an essential factor for users since several cloud vendors charge their customers based on execution time. In order to minimize the impact of execution time, the system has to reduce cold start events of containers and minimize queueing time for invocations.

However, from the results we obtained, which is shown in Figure 7, we noticed that running OpenWhisk with balanced and over-invoked invocation rates can cause a relatively high cold start. This is possibly due to the spinning up of new containers outside of their grace period.

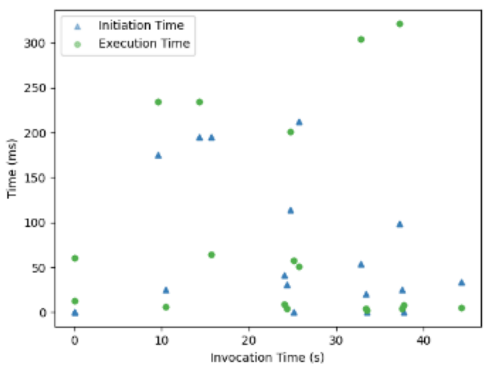
Moreover, a highly variable pattern of execution time is observed for balanced, and over-invoked models. We believe that the reason for this variation might be because of an increase in concurrency due to involuntary context switches as well as high overhead for spinning up new Docker containers.



1 ips



20 ips



100 ips

Figure 7. Execution time for different invocation rates while running all 5 functions concurrently.

C. *Test C*

In Test C, we compared the profiling data of each of the function invocations on the serverless platform with native execution. We executed all the five benchmark functions independently with varying invocation rates of 1, 20 and 100 ips but maintaining the same activity window of five seconds and uniformly distributed execution pattern. The results we obtained from the FaaSProfiler for each of the functions exhibit similar patterns across all the respective invocation rates. The sample profiling plots for SmartConvAction (developed in Python) is shown in Figure 8.

From these figures, we can notice that for a rate of 1 ips, corresponding to an under-invoked event, the initiation time is relatively high throughout the activity window of 0 to 5 seconds. This could be attributed to the cold start event of containers for executing new function invocations. However, for a rate of 20 and 100 ips, corresponding to balanced and over-invoked cases, the initiation time is high during the initial phase, and reduces for later phase of invocations in the activity window. This could be attributed to a large function invocation rate that keep the containers alive, thus, resulting in only observing the cold start events in the beginning. Note that we removed the high initiation time (around 2000 msec) points for 20 and 100 ips for the 0th second to have much better visual representation of other latency features.

Next, we notice a varying wait time pattern for rate 1 ips plot. This could be attributed to the OpenWhisk's Controller randomly initializing and killing a set of containers for a small invocation rate. Moreover, as the rate increases to 20 ips, we notice that an emptying queuing pattern for wait time is formed. This is because initially the wait time increases due to the containers initializing for invocation of functions and subsequently being killed after execution of the function terminates. Next, for a rate of 100 ips, we observe a queuing pattern formation for the wait time. This is due to the fact that initially wait time increases very fast due to the cold startup of containers for invoking a large set of functions. Then the rate at which wait time increases slowly starts to decrease (while maintaining a linear pattern) due to the functions waiting to run on the limited set of containers that are being utilized by the previous set of function invocations.

Finally, the execution time, which is expressed as the difference between run time and initiation time, has much more variability for balanced and over-invoked scenarios as compared to the under-invoked case. This could be attributed to a high amount of context switches happening at the system level due to a large number of invocation rates. Moreover, this variability could also be attributed to the interference between docker containers used for executing respective function invocations. This behavior of the serverless platform could be a huge downside as several cloud vendors charge end users based on execution time of the serverless functions. Having a high variability for different invocation rates can be problematic, as the end users would be charged a huge amount even though they had a smaller scale of function invocations.

The results we obtained for native execution of each function with varying invocation rate, is shown in Table 2. The general trend that we can see is that as the function execution rate increases, the latency also increases. This is in accordance with the fact that more compute resources would be required for performing a large number of function executions. Apart from this, we also notice that the overall latency for separate invocation rates varies across different kinds of functions. This differs from our serverless testing, where all the functions, irrespective of the programming language they have been developed in and the type of actions they accomplish, had similar wait, initiation and execution time pattern based on the specified invocation rate. This shows that serverless could be suitable for running a wide range of tasks without causing a huge impact on the server hardware on which these functions run as opposed to native function execution. Moreover, as the OpenWhisk platform can automatically scale the amount of docker containers for running a particular number of functions, serverless shows advantage in scalability as opposed to native execution of functions that use default configuration of the programming language and the system for running tasks. This may be a downside as it will involve a lot of fine tuning to find optimal specifications for running a particular set of actions.

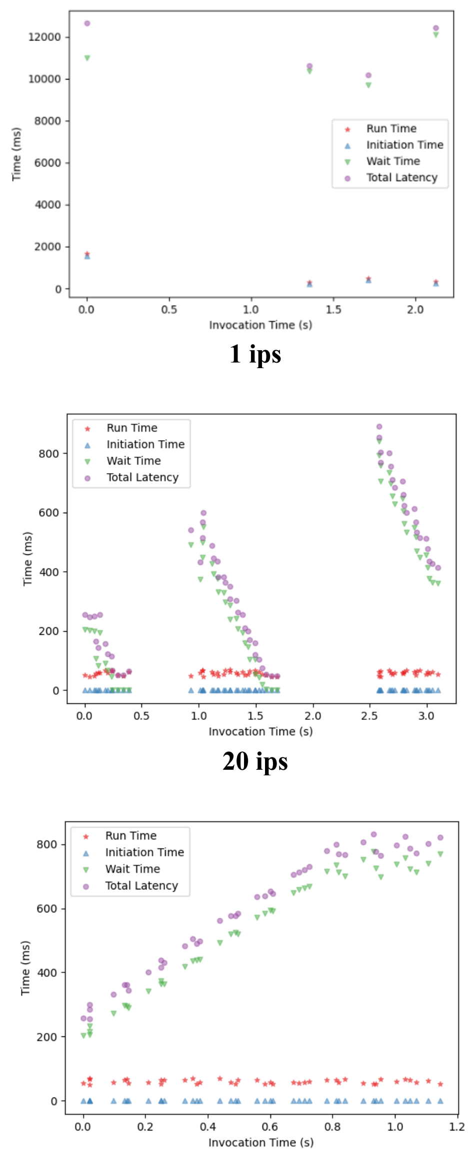


Figure 8. SmartConvAction function results for 1, 20 and 100 ips.

Serial No.	Functions	Rate 1(s)	Rate 20 (s)	Rate 100 (s)
1	CouchDBAction	0.329	1.524	7.597
2	SmartConvAction	0.208	1.098	5.275
3	MoodAction	0.677	3.959	17.180
4	LeftPad	0.324	1.485	7.608
5	APIAction	0.558	2.189	10.725

Table 2 Test C results for native functions with 1, 20, and 100 ips.

D. Test D

Finally, we record the hardware performance of each function for both serverless invocation and local execution. Using Linux performance analyzer tool Perf, we focused on three main performance metrics which are branch-misses, page faults and context-switches. Branch-misses record the number of unsuccessful branch predictions during the run time. It can cause significant delay if the percentages are too high. A page fault occurs when a process accesses a page that is mapped in the virtual address space, but not loaded in the physical memory. High page faults can impact the system and result in performance issues. Finally, a context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU (central processing unit) from one process or thread to another. High context-switches indicate that many processes are competing for processor time and can hence, result in system-level problems.

For comparison purposes, we recorded the performance for each function using under-invoked, balanced, and over-invoked execution strategies for OpenWhisk. The sample results for CouchDBAction are shown in Figure 9. We used 1 ips from activity windows of 0 to 5, 20 ips from 10 to 15, and 100 ips from 20 to 25 seconds for obtaining much better visual results. Table 3 shows the results for running these functions locally with the same invocation parameters.

For page faults, we observed unique data patterns when using serverless deployment. The data identifies that the number of page-faults per kilo-instruction increases dramatically for balanced invocation with a high variation. However, for under invoked, the data points are far more stabilized with a lower value. Finally, the number of page faults reaches the lowest value among the three experiments for the over-invoked case. We observed similar pattern throughout all the five benchmark functions. A sample plot of the results for CouchDBAction is shown in Figure 9.

Table 3 provides the page faults with native function execution. With the increase of concurrency, we notice that the number of page faults per kilo-instructions decreases. This observation stands valid for most of the testing functions except for MoodAction. The number of page faults seems to stay roughly the same across functions despite each of them having different programming language overhead and task accomplished.

Combining the results that we obtained, we noticed several intuitive findings. First, the spike of page-faults for balanced invocation for serverless computing might be due to the fact that the containers are spawning too frequently without taking the advantages of the grace period. The rate is not high enough for keeping the containers warm in OpenWhisk hence, the system has to keep initializing new containers which leads to huge overheads. As the function invocation rate increases, this issue is solved since the containers are still warm for executing function instances hence, reducing the overhead of constantly spawning separate containers. Second, despite the high number of page faults for balanced invocation, we obtained a very similar result for under-invoked and over-invoked experiments. This implies a similar hardware performance between running OpenWhisk locally and native functions. This pattern can be found disregarding the type of the functions we run.

Branch-misses, which is the second performance criteria in Test D, provided us with identical results. While using OpenWhisk, we observed high and varied branch-misses for balanced invocation rate and relatively low rate for under and over-invoked scenarios. However, the number of branch-misses for over-invoked is higher than under-invoked for all five functions. Figure 9 illustrates an example of the pattern for CouchDBAction.

The local performance of branch-misses also shares a very similar pattern across functions with the exception of MoodAction. Results can be found in Table 3.

Since a high number of branch-misses can indicate performance issues in the system just as page fault does, we believe that the reasoning for the spike in the balanced invocation model follows similar explanation from page fault results for such function invocation patterns. However, one difference we notice is that the branch mispredictions increase for over-invoked data compared with the under-invoked result. This might be due to large number of containers that were running in the experiment.

Finally, context-switches are the third resource metric we analyzed for our testing. Without surprise, we found a pattern similar to the results obtained from branch misses and page faults. Moreover, we also noticed that the under-invoked data seemed much more variable compared to the over-invoked result, which stabilizes throughout the running time. The sample results we obtained for CouchDBAction are presented in Figure 9.

The local results indicate that as more native functions run concurrently, the number of context-switches increases. The results justify the intuition that high concurrence results in high CPU switches. Same pattern is observed disregarding the type of the functions. Table 3 justifies this observation.

By comparing what we obtained, we argue that the constant spawning of new containers are still the reasons for high context-switches numbers in the balanced and over-invoked execution tests. The overheads of creating new containers lead to such results.

Function No. and Rate	Branch Mispredictions (M/sec)	Context Switches (K/s)	Page Faults (M/s)
F1 rate1	35.279	0.073	0.023
F1 rate20	23.349	0.488	0.013
F1 rate100	23.302	0.641	0.013
F2 rate1	24.55	0.068	0.023
F2 rate20	15.183	0.407	0.011
F2 rate100	15.203	0.546	0.011

F3 rate1	10.814	0.533	0.008
F3 rate20	14.723	126	0.011
F3 rate100	15.723	53	0.013
F4 rate1	34.39	0.074	0.022
F4 rate20	22.945	0.569	0.013
F4 rate100	23.261	0.602	0.013
F5 rate1	30.994	0.268	0.022
F5 rate20	21.273	0.631	0.013
F5 rate100	21.49	0.701	0.013

Table 3 Test D results for native function execution with 1, 20, and 100 ips.

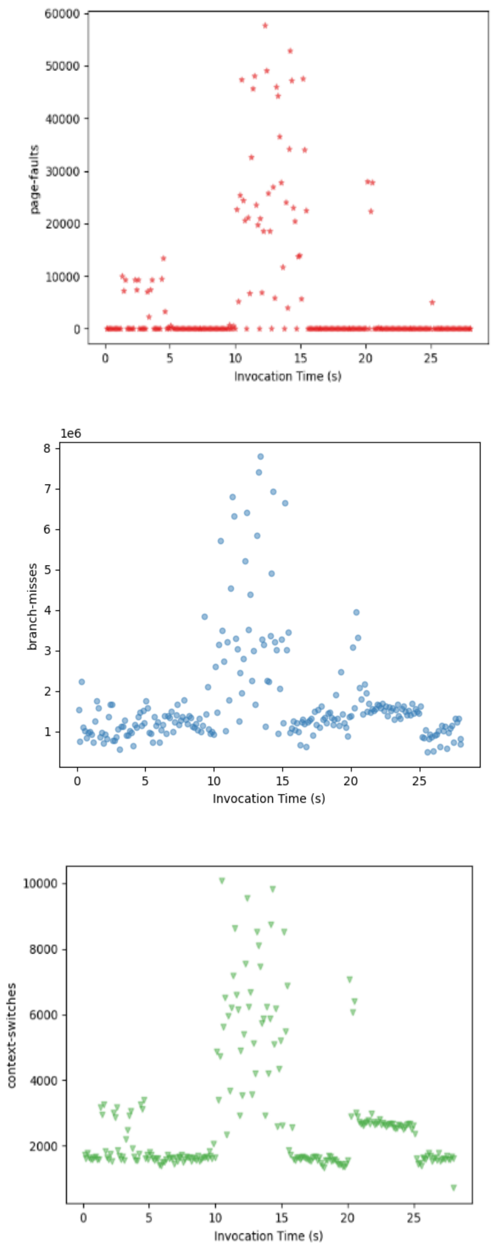


Figure 9. CouchDBAction result for page faults, branch mispredictions, and context-switches.

VIII. Future Work

The grace period of Docker containers in OpenWhisk can have a significant impact on the performance for serverless computing. Therefore, if the users can adjust the grace period for the containers based on their job requirements and specifications, the performance might significantly improve. One of the future studies that we aim to focus on is how to adjust the grace period based on the jobs to achieve better performance using OpenWhisk. Similar idea can also be applied to other cloud serverless services such as AWS Lambda, Azure Functions and so on.

We could also explore serverless function development in other programming languages in order to evaluate and better understand the language induced overheads. We additionally aim to test the impact these functions may have on performance results across various cloud services such as AWS, Azure and so on.

Finally, we aim to search for more FaaS profiling tools or even create our own for better performance analysis. We faced several challenges while using FaaSprofiler, such as mislabeling of several parameters while plotting and bugs faced while running the software that we had to manually fix during the research process. New updates to the OpenWhisk platform might have been the reason for these issues. Moreover, the profiler only supports a certain version number of programming languages and operation systems configuration to build which also causes several issues while deployment. We aim to improve this situation in the future.

IX. Conclusion

A prominent upcoming cloud service approach that several large IT companies are providing is serverless compute. Due to the ease of just writing the code and vendors taking care of provisioning the servers and resources for the function to run on, serverless has gained huge popularity among developers. However, there are several downsides of running serverless functions over the cloud that the end user must be aware of.

Through this paper, we have analyzed the impact serverless functions have on latency with varying invocation patterns and concurrency. Moreover, we also studied how different invocations of separate functions deployed in different programming languages can influence performance and resource metrics over serverless as opposed to executing them natively. There are several future related tasks that can be explored to better understand the internal workings of serverless and development of new tools to address current challenges that this cloud model is facing.

Acknowledgment

We would like to thank Professor Christina Delimitrou for providing us an opportunity to work on this project as part of the ECE 5710: Datacenter Computing course. This project has been instrumental in helping us learn applications of serverless compute frameworks and analyzing its tradeoffs with native execution. We additionally appreciate her efforts in creating an extremely engaging environment for discussing project ideas as well as providing invaluable advice despite having meetings online due to the Covid-19 pandemic.

REFERENCES

[1] "Amazon lambda," https://aws.amazon.com/lambda/, accessed: 2021-04-01.

[2] Apache Mesos. URL: http://mesos.apache.org/, accessed: 2021-04-05.

[3] "Azure functions," https://azure.microsoft.com/en-us/services/functions/, accessed: 2021-04-01.

[4] "Cloud functions," https://cloud.google.com/functions/, accessed: 2021-04-05.

[5] Compose. URL: https://docs.docker.com/compose/, accessed: 2021-04-10.

[6] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," Linux J., vol. 2014, Mar. 2014.

[7] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al., "Cloud programming simplified: a berkeley view on serverless computing," arXiv preprint arXiv:1902.03383, 2019.

[8] FaaSProfiler. URL: https://github.com/PrincetonUniversity/faas-profiler, accessed: 2021-04-13.

[9] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," arXiv preprint arXiv:1812.03651, 2018.

[10] Kubernetes. URL: https : / / kubernetes . io/, accessed: 2021-04-07.

[11] OpenShift. URL: https://www.openshift.com/, accessed: 2021-04-08.

[12] "Openwhisk," https://openwhisk.apache.org/, accessed: 2021-04-05.

[13] Shahrad, M., Balkind, J., &amp; Wentzlaff, D. (2019). Architectural Implications of Function-as-a-Service Computing. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. https://doi.org/10.1145/3352460.3358296.

[14] "The serverless application framework," https://www.serverless.com/, accessed: 2021-04-15.