Benchmarking Heterogeneous Cloud Functions

Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima

AGH University of Science and Technology
Department of Computer Science,
Krakow, Poland
{malawski, kfigiela}@agh.edu.pl

Abstract. Cloud Functions, often called Function-as-a-Service (FaaS), pioneered by AWS Lambda, are an increasingly popular method of running distributed applications. As in other cloud offerings, cloud functions are heterogeneous, due to different underlying hardware, runtime systems, as well as resource management and billing models. In this paper, we focus on performance evaluation of cloud functions, taking into account heterogeneity aspects. We developed a cloud function benchmarking framework, consisting of one suite based on Serverless Framework, and one based on HyperFlow. We deployed the CPU-intensive benchmarks: Mersenne Twister and Linpack, and evaluated all the major cloud function providers: AWS Lambda, Azure Functions, Google Cloud Functions and IBM OpenWhisk. We make our results available online and continuously updated. We report on the initial results of the performance evaluation and we discuss the discovered insights on the resource allocation policies.

Keywords: Cloud computing, FaaS, Cloud Functions, Performance Evaluation

1 Introduction

Cloud Functions, pioneered by AWS Lambda, are becoming an increasingly popular method of running distributed applications. They form a new paradigm, often called Function-as-a-Service (FaaS) or serverless computing. Cloud functions allow the developers to deploy their code in the form of a function to the cloud provider, and the infrastructure is responsible for the execution, resource provisioning and automatic scaling of the runtime environment. Resource usage is usually metered with millisecond accuracy and the billing is per every 100 ms of CPU time used. Cloud functions are typically executed in a Node.js environment, but they also allow running custom binary code, which gives an opportunity for using them not only for Web or event-driven applications, but also for some compute-intensive tasks, as presented in our earlier work [5].

As in other cloud offerings, cloud functions are heterogeneous in nature, due to various underlying hardware, different underlying runtime systems, as well as resource management and billing models. For example, most providers use Linux as a hosting OS, but Azure functions run on Windows. This heterogeneity is in principle hidden from the developer by using the common Node.js environment, which is platform-independent, but again various providers have different versions of Node (as of May 2017: for AWS Lambda – Node 6.10, for Google Cloud Functions – Node 6.9.1, for IBM Bluemix – Node 6.9.1, for Azure – 6.5.0.). Moreover, even though there is a common "function" abstraction for all the providers, there is no single standard API.

In this paper, we focus on performance evaluation of cloud functions and we show how we faced various heterogeneity challenges. We have developed a framework for performance evaluation of cloud functions and applied it to all the major cloud function providers: AWS Lambda, Azure Functions, Google Cloud Functions (GCF) and IBM OpenWhisk. Moreover, we used our existing scientific workflow engine HyperFlow [1] which has been recently extended to support cloud functions [5] to run parallel workflow benchmarks. We report on the initial results of the performance evaluation and we discuss the discovered insights on the resource allocation policies.

The paper is organized as follows. Section 2 discusses the related work on cloud benchmarking. In Section 3, we outline our framework, while in Section 4, we give the details of the experiment setup. Section 5 presents and discusses the results, while Section 6 gives a summary and outlines the future work.

2 Related Work

Cloud performance evaluation, including heterogeneous infrastructures has been subject of previous research. An excellent example is in [2], where multiple clouds are compared from the perspective of many-task computing applications. Several hypotheses regarding performance of public clouds are discussed in a comprehensive study presented in [3]. More recent studies focus e.g. on burstable [4] instances, which are cheaper than regular instances but have varying performance and reliability characteristics. Performance of alternative cloud solutions such as Platform-as-a-Service (PaaS) has also been analyzed. E.g. [6, 8] focused on Google App Engine from the perspective of CPU-intensive scientific applications.

A detailed performance and cost comparison of traditional clouds with microservices and the AWS Lambda serverless architecture is presented in [10], using an enterprise application. Similarly, in [11] the authors discuss the advantages of using cloud services and AWS Lambda for systems that require higher resilience. An interesting discussion of serverless paradigm is given in [7], where the case studies are blogging and media management application. An example of price and performance of cloud functions is provided also in [9], describing Snafu, a new implementation of FaaS model, which can be deployed in a Docker cluster on AWS. Its performance and cost is compared with AWS Lambda using a recursive Fibonacci cloud function benchmark.

Up to our knowledge, heterogeneous cloud functions have not been comprehensively studied yet, which motivates this research.

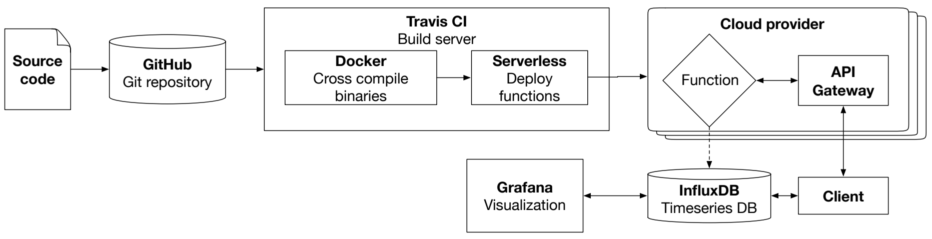

Fig. 1. Architecure of the cloud functions benchmarking framework based on Serverless Framework

3 Benchmarking Framework for Cloud Functions

For benchmarking cloud function providers, we used two frameworks. The first one is our new suite, designed specifically for this research, based on Serverless Framework. The second one uses our HyperFlow workflow engine [1, 5].

3.1 Suite Based on Serverless Framework

The objective of this benchmarking suite is to execute and gather performance results of heterogeneous cloud function benchmarks over a long period of time. The suite has to run as a permanent service and execute selected benchmarks periodically. The results are then stored and available for examination. Our goal was to automate functions deployment as much as possible to improve results reproducibility. The architecture of the suite is shown in Fig. 1.

In order to deploy our benchmark suite we have used the Serverless Framework[1]. It provides a uniform way of setting up cloud functions deployment and supports, at the time of writing, AWS Lambda, IBM OpenWhisk and Azure Functions natively and Google Cloud Functions through an official plugin. In order to streamline our data taking process, we automated code deployment even further by setting up project on Travis continuous integration (CI), so that the code is automatically deployed on each cloud whenever we push new code to the Git repository. This also simplified security credentials management, since we do not need to distribute deployment credentials for each provider.

To address the heterogeneity of runtime environments underlying cloud functions, we have created dedicated wrappers for native binary that was executed by the function. We have used Docker to build binaries compatible with target environments. For Linux based environments, we use amazonlinux image to build a static binary that is compatible with AWS Lambda, Google Cloud Functions and IBM OpenWhisk. Azure Functions run in a Windows-based environment, thus it requires a separate binary. We used Dockcross[2] project that provides a suite of Docker images with cross-compilers, which includes a Windows target.

[1] `https://serverless.com`
[2] `https://github.com/dockcross/dockcross`

The Serverless Framework is able to deploy functions with all the necessary companion services (e.g. HTTP endpoint). However, we still had to adapt our code slightly for each provider, since the required API is different. For instance, AWS Lambda requires a callback when a function result is ready, while IBM OpenWhisk requires to return a Promise for asynchronous functions. The cloud platforms also differ in how $PATH and current working directory are handled.

The benchmarks results are sent to the InfluxDB time series database. We have also setup Grafana for convenient access to benchmark results. We implemented our suite in Elixir and Node.js. The source code is available on GitHub[3].

3.2 Suite Based on HyperFlow

For running parallel benchmarking experiments we adapted HyperFlow [1] workflow engine. HyperFlow was earlier integrated with GCF [5], and for this work it was extended to support AWS Lambda. HyperFlow is a lightweight workflow engine based on Node.js and it can orchestrate complex large-scale scientific workflows, including directed acyclic graphs (DAG).

For the purpose of running the benchmarks, we used a set of pre-generated DAGs of the fork-join pattern: the first task is a fork task which does not perform any job, it is followed by N identical parallel children of benchmark tasks running the actual computation, which in turn are followed by a single join task which plays the role of a final synchronization barrier. Such graphs are typical for scientific workflows, which often include such parallel stages (bag of tasks), and moreover are convenient for execution of multiple benchmark runs in parallel.

In the case of HyperFlow, the cloud function running on the provider side is a JavaScript wrapper (HyperFlow executor), which runs the actual benchmark, measures the time and sends the results to the cloud storage, such as S3 or Cloud Storage, depending on the cloud provider.

4 Experiment Setup

We configured our frameworks with two types of CPU-intensive benchmarks, one focused on integer and the other on floating-point performance.

4.1 Configuration of the Serverless Benchmarking Suite

In this experiment we used a random number generator, as an example of an integer-based CPU-intensive benchmark. Such generators are key in many scientific appliations, such as Monte Carlo methods, which are good potential candidates for running as cloud functions.

Specifically, the cloud function is a JavaScript wrapper around the binary benchmark, which is a program written in C. We used a popular Mersenne Twister (MT19937) random number generator algorithm. The benchmark runs

[3] `https://github.com/kfigiela/cloud-functions`

approximately 16.7 milion iterations of the algorithm using a fixed seed number during each run and provides reproducible load.

We measure the execution time t_b of the binary benchmark from within the JavaScript wrapper that is running on serverless infrastructure, and the total request processing time t_r on the client side. We decided to deploy our client outside the clouds that were subject to examination. The client was deployed on a bare-metal ARM machine hosted in Scaleway cloud in Paris datacenter. The benchmark was executed for each provider every 5 minutes. We took multiple measurements for different memory sizes available: for AWS Lambda – 128, 256, 512, 1024, 1536 MB, for Google Cloud Functions – 128, 256, 512, 1024, 2048 MB, for IBM OpenWhisk – 128, 256, 512 MB. Azure Functions do not provide a choice on function size and the memory is allocated dynamically. The measurements: binary execution time t_b and request processing time t_r were sent to InfluxDB by the client. Since the API Gateway used in conjunction with AWS Lambda restricts request processing time to 30 seconds, we were not able to measure t_r for 128 MB Lambdas. Although the requests timeout on the API Gateway, the function completes execution. In this case, the function reports t_b time directly to InfluxDB.

On AWS Lambda functions were deployed in eu-west-1 region, on GCF functions were deployed in us-central1 region, on IBM OpenWhisk functions were deployed in US South region and on Azure function was deployed in US West region. Such setup results from the fact that not all of the providers offer cloud functions in all their regions yet.

We started collecting data on April 18, 2017, and the data used in this paper include the values collected till May 11, 2017.

4.2 Configuration of HyperFlow Suite

As a benchmark we used the HPL Linpack[4], which is probably the most popular CPU-intensive benchmark focusing on the floating point performance. It solves a dense linear system of equations in double precision and returns the results in GFlops. To deploy the Linpack on multiple cloud functions, we used the binary distribution from Intel MKL[5], version mklb_p_2017.3, which has binaries for Linux and Windows.

As benchmark workflows we generated a set of fork-join DAGs, with parallelism $N = [10, 20, ..., 100]$, thus it allowed us to run up to 100 Linpack tasks in parallel. Please note that in this setup all the Linpack benchmarks run independently, since cloud functions cannot communicate with each other, so this configuration differs from the typical Linpack runs in HPC centers which use MPI. Our goal is to measure the performance of individual cloud functions and the potential overheads interference between parallel executions.

The Linpack was configured to run using the problem size (number of equations) of $s \in \{1000, 1500, 2000, 3000, 4000, 5000, 6000, 8000, 10000, 12000, 15000\}$.

[4] http://www.netlib.org/benchmark/hpl/

[5] https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite

Not all of these sizes are possible to run on functions with smaller memory, e.g. $4000 \times 4000 \times 8 Bytes = 128 MB$, so the benchmark stops when it cannot allocate enough memory, reporting the best performance p_f achieved (in GFlops).

We run the Linpack workflows for each N on all the possible memory sizes available on GCF (128, 256, 512, 1024, 2048 MB) and on AWS Lambda on sizes from 128 to 1536 with increments of 64 MB.

On AWS Lambda functions were deployed in eu-west-1 region, on GCF functions were deployed in us-central1 region.

5 Performance Evaluation Results

Our benchmarks from the serverless suite run permanently and the original unfiltered data as well as current values are available publicly on our website[6]. They include also selected summary statistics and basic histograms. The data can be exported in CSV format, and we included the data in the GitHub repository.

Selected results are presented in the following subsections (5.1 and 5.2), while the results of the Linpack runs using HyperFlow are given in section 5.3.

5.1 Integer Performance Evaluation

The results of the integer benchmarks using Mersenne Twister random generator are presented in Fig. 2. They are shown as histograms, grouped by providers and function size. They give us interesting observations about the resource allocation policies of cloud providers.

Firstly, the performance of AWS Lambda is fairly consistent, and agrees with the documentation which states that the CPU allocation is proportional to the function size (memory). On the other hand, Google cloud functions execution time have multimodal distributions with higher dispersion. For example, for the 256 MB function, the execution time is most often around 27 s, but there is another peak around 20 s, coinciding with the faster 512 MB function. Similarly, the distribution for the slowest 128 MB function has multiple peaks, overlapping with faster functions and reaching even the performance of the fastest 2048 MB function. This suggests that GCF does not enforce strictly the performance limits, and opportunistically invokes smaller functions using the faster resources.

Regarding IBM Bluemix, the performance does not depend on the function size, and the distribution is quite narrow, as in the case of AWS. On the other hand, the performance of Azure has much wider distribution, and the average execution times are relatively slower. This can be attributed to different hardware, but also to the underlying operating system (Windows) and virtualization.

5.2 Overheads Evaluation

By measuring the binary execution time t_b inside the functions as well as the request processing time t_r (as seen from the client), we can also obtain a rough

[6] `http://cloud-functions.icsr.agh.edu.pl`

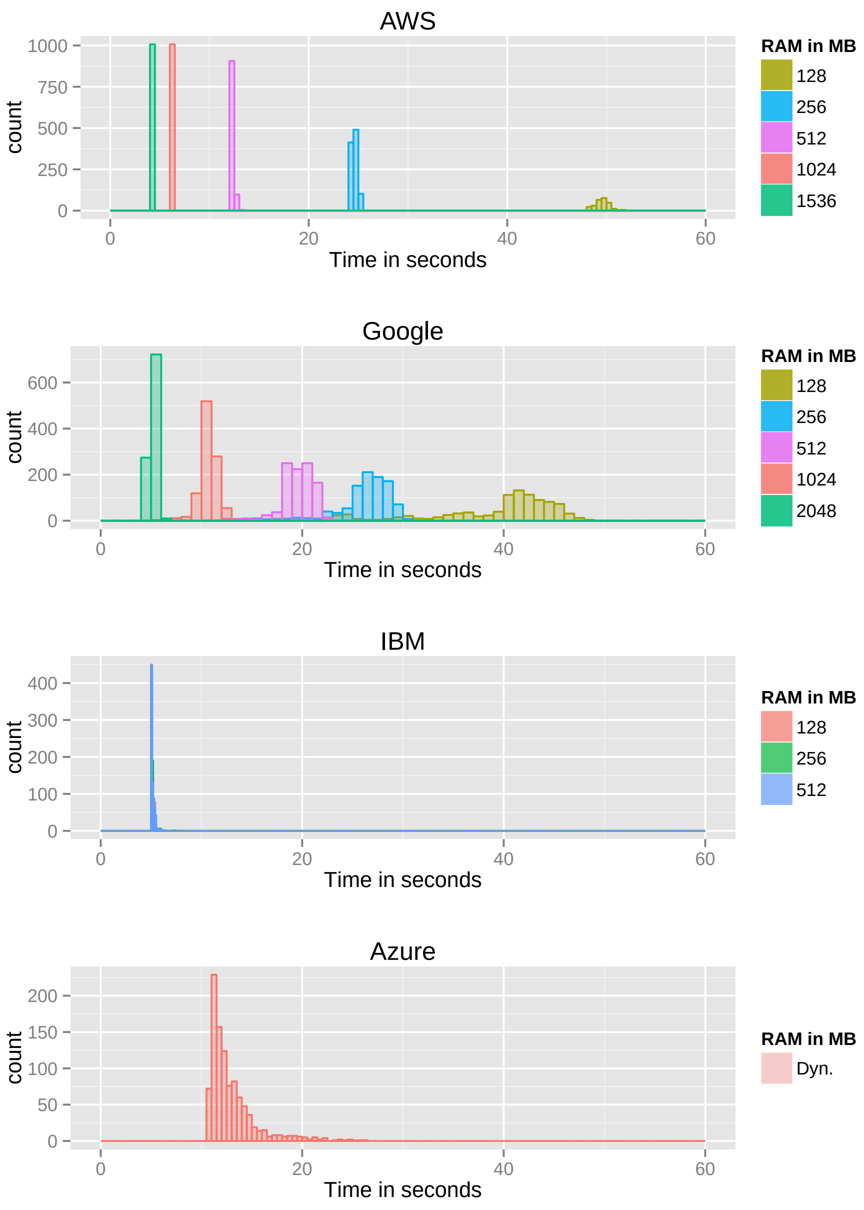

Fig. 2. Histograms of integer-based MT random number generator benchmark execution time vs. cloud function size. In the case of Azure memory is allocated dynamically.

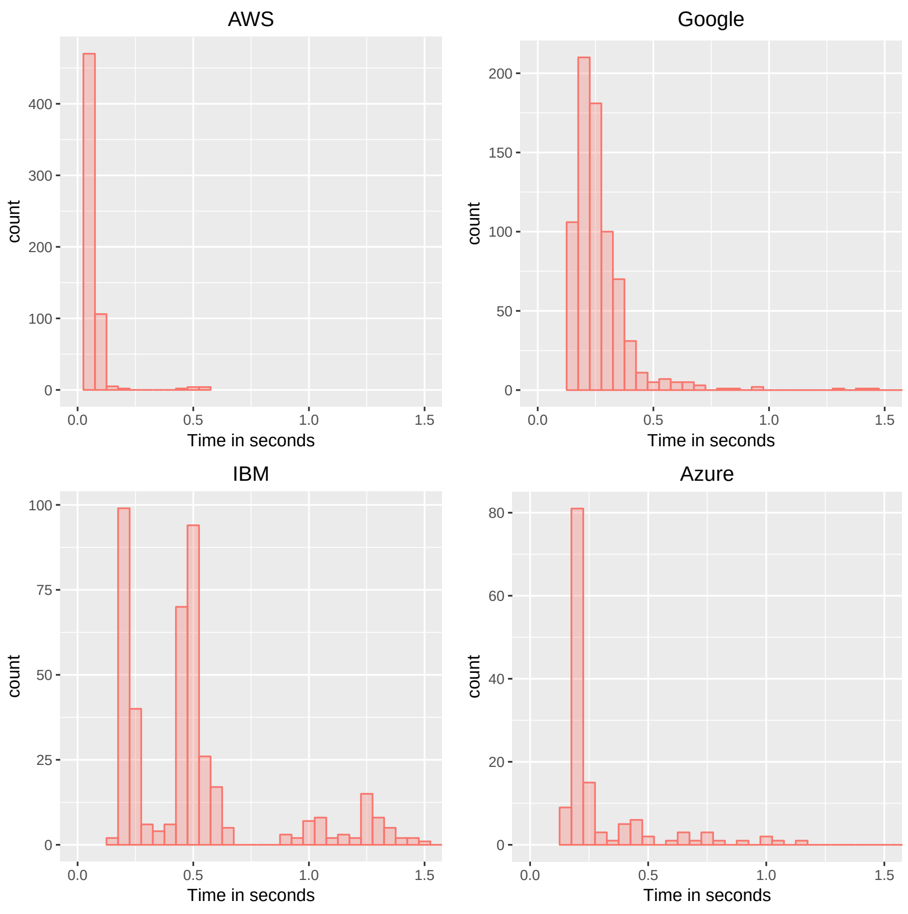

Fig. 3. Distribution of t_o overheads for cloud function providers.

estimate on total overhead $t_o = t_r - t_b$. The overhead includes: network latency, platform routing and scheduling overheads. Endpoints exposed by cloud providers are secured with HTTPS protocol. We warmed up the connection before performing each measurement, so that we were able to exclude the TLS handshake from t_r. Unfortunately, we could not measure the network latency to the clouds as AWS and Google provide access to functions via CDN infrastructure. The average round trip latency (ping) to OpenWhisk was 117 ms and 155 ms to Azure.

Histograms of t_o are presented in Fig. 2. One may observe that overhead is stable with a few outliers. However, for Bluemix one may see that there are two peaks in the distribution.

Furthermore, we measured t_r for requests targeting an invalid endpoint. This gives a hint on network latency under the assumption that invalid requests are terminated in an efficient way. The average results were consistent with typical network latency: for AWS Lambda – 43 ms, for Google Cloud Functions – 150 ms, for IBM Bluemix – 130 ms. However, for Azure the latency measured that way was 439 ms which is significantly larger than the network ping time.

5.3 Floating-point Performance Evaluation

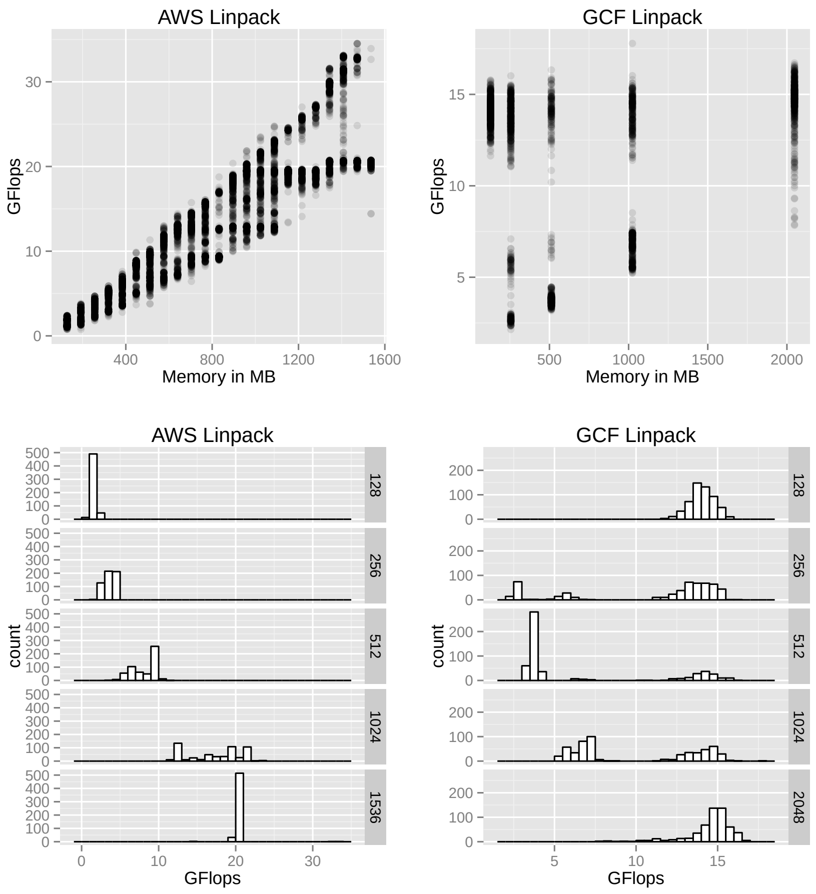

Fig. 4. Linpack performance versus cloud function size.

Results of the Linpack runs are shown in Fig. 4, as a scatter-plots where density of circles represents the number of data points. AWS data consists of over 12,000 points, and GCF of over 2,600 points. We show also histograms of subsets of these data.

In the case of AWS, we observe that the maximum performance grows linearly with the function size. There is, however, a significant portion of tasks that achieved lower performance. With the growing memory, we can see that the execution times form two clusters, one growing linearly over 30 GFlops, and one saturating around 20 GFlops.

In the case of GCF, we observe that the performance of tasks is clustered differently. The performance of one group of tasks grows linearly with memory. On the other hand, there is a large group of tasks, which achieve the top performance of 15 GFlops regardless of the function size. Interestingly, we observed that the smallest functions of 128MB always achieved the best performance of about 14 GFlops.

To illustrate the multimodal nature of performance distribution curves of GCF, we show the results as histograms in Fig. 4 for selected memory siezes. As in the case of integer-based performance tests, the AWS Lambda show much more consistent results, while for GCF the performance points are clustered.

5.4 Discusion of Results

The most interesting observation is regarding the scheduling policies of cloud providers, as observed in both MT and Linpack experiments. Both GCF and AWS claim that the CPU share for cloud functions is proportional to the memory allocated. In the case of AWS we observe a fairly linear performance growth with the memory size, both for the lower bound and the upper bound of the plot in Fig. 4. In the case of GCF, we observe that the lower bound grows linearly, while the upper bound is almost constant. This means that Google infrastructure often allocates more resources than the required minimum. This means that their policy allows smaller functions (in terms of RAM) to run on faster resources. This behavior is likely caused by optimization of resource usage via reuse of already spawned faster instances, which is more economical that spinning up new smaller instances. Interestingly, for Azure and IBM we have not observed any correlation between the function size and performance.

Another observation is the relative performance of cloud function providers. AWS achieves higher scores in Linpack (over 30 GFlops) whereas GCF tops at 17 GFlops. Interestingly, from the Linpack execution logs we observed that the CPU frequency at AWS is 3.2 GHz, which suggests Xeon E5-2670 (Ivy Bridge) family of processors, while at GCF it is 2.1 GHz which means Intel Xeon E5 v4 (Broadwell). Such difference in hardware definitely influences the performance. These Linpack results are confirmed by the MT benchmark. Since we have not run Linpack on Azure and IBM yet, we cannot report on their floating point performance, but the MT results also suggest the differences in hardware.

Although we did not perform such detailed statistical tests as in [3], our observations confirm that there is not significant dependency of the time of day or day of week on the cloud providers performance. The existing fluctuations tend to have random characteristics, but it will be subject to further studies once we collect more data.

6 Summary and Future Work

In this paper, we presented our approach to performance evaluation of cloud functions. We described our performance evaluation framework, consisting of two suites, one using the Serverless Framework, and the one based on HyperFlow. We gave the technical details on how we address the heterogeneity of the environment, and we described our automated data taking pipeline. We made our experimental primary data available publicly to the community and we set up the data taking as a continuous process.

The presented results of evaluation using Mersenne Twister and Linpack benchmarks show the heterogeneity of cloud function providers, and the relation between the cloud function size and performance. We also revealed the interesting observations on how Amazon and Google differently interpret the resource allocation policies. These observations can be summarized that AWS Lambda functions execution performance is proportional to the memory allocated, but sometimes sightly slower, while for Google Cloud Functions the performance is proportional to the memory allocated, but often much faster.

Since this paper presents the early results of this endeavor, there is much room for future work. It includes the integration of HyperFlow with our serverless benchmarking suite, measurement of influence of parallelism, delays and warm-up times on the performance, possible analysis of trends as we continue to gather more data, as well as cost-efficiency analysis and implications for resource management.

Acknowledgements. This work was supported by the National Science Centre, Poland, grant 2016/21/B/ST6/01497.

References

1. Balis, B.: Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows. Future Generation Computer Systems 55, 147 – 162 (2016)
2. Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. IEEE Transactions on Parallel and Distributed Systems 22(6), 931–945 (jun 2011)
3. Leitner, P., Cito, J.: Patterns in the chaos - A study of performance variation and predictability in public iaas clouds. ACM Trans. Internet Techn. 16(3), 15:1–15:23 (2016), `http://doi.acm.org/10.1145/2885497`
4. Leitner, P., Scheuner, J.: Bursting with possibilities - an empirical study of credit-based bursting cloud instance types. In: 8th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2015, Limassol, Cyprus, December 7-10, 2015. pp. 227–236 (2015), `http://doi.ieeecomputersociety.org/10.1109/UCC.2015.39`
5. Malawski, M.: Towards serverless execution of scientific workflows – HyperFlow case study. In: WORKS 2016 Workshop, Workflows in Support of Large-Scale Science, in conjunction with SC16 Conference. CEUR-WS.org, Salt Lake City, Utah, USA (November 2016)

6. Malawski, M., Kuzniar, M., Wojcik, P., Bubak, M.: How to Use Google App Engine for Free Computing. IEEE Internet Computing 17(1), 50–59 (Jan 2013)
7. McGrath, M.G., Short, J., Ennis, S., Judson, B., Brenner, P.R.: Cloud event programming paradigms: Applications and analysis. In: 9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016. pp. 400–406. IEEE Computer Society (2016)
8. Prodan, R., Sperk, M., Ostermann, S.: Evaluating High-Performance Computing on Google App Engine. IEEE Software 29(2), 52–58 (Mar 2012)
9. Spillner, J.: Snafu: Function-as-a-service (faas) runtime design and implementation. CoRR abs/1703.07562 (2017), http://arxiv.org/abs/1703.07562
10. Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., Lang, M.: Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 179–182 (May 2016)
11. Wagner, B., Sood, A.: Economics of Resilient Cloud Services. In: 1st IEEE International Workshop on Cyber Resilience Economics (Aug 2016), http://arxiv.org/abs/1607.08508