Resource Management for Cloud Functions with Memory Tracing, Profiling and Autotuning

Josef Spillner
josef.spillner@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

Abstract

Application software provisioning evolved from monolithic designs towards differently designed abstractions including serverless applications. The promise of that abstraction is that developers are free from infrastructural concerns such as instance activation and autoscaling. Today's serverless architectures based on FaaS are however still exposing developers to explicit low-level decisions about the amount of memory to allocate for the respective cloud functions. In many cases, guesswork and ad-hoc decisions determine the values a developer will put into the configuration. We contribute tools to measure the memory consumption of a function in various Docker, OpenFaaS and GCF/GCR configurations over time and to create trace profiles that advanced FaaS engines can use to autotune memory dynamically. Moreover, we explain how pricing forecasts can be performed by connecting these traces with a FaaS characteristics knowledge base.

CCS Concepts: • **Computing methodologies** → *Concurrent computing methodologies*; • **Software and its engineering** → **Cloud computing**; System modeling languages.

Keywords: serverless computing, vertical scaling, models

ACM Reference Format:
Josef Spillner. 2020. Resource Management for Cloud Functions with Memory Tracing, Profiling and Autotuning. In *Workshop on Serverless Computing (WoSC'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429880.3430094

1 Problem Statement

The provisioning characteristics of a FaaS-based application are affected by memory use in two ways: First, apart from a static per-invocation cost component, most providers include a duration-utilisation product as complementary cost component (e.g. $GB \times s$ or $CU \times s$). For compute-intensive services that run more than just a few seconds and consume more than the minimum amount of memory, this cost component becomes dominant and significant. For instance, in AWS Lambda, the per-invocation fee is 0.20 USD per million instances, whereas the duration-utilisation fee is 0.2083 USD for the same in the minimum execution ($\frac{1}{8}GB \times \frac{1}{10}s \times 1M$, us-east region), and a multiple of that price tag otherwise (not accounting for free tiers, which slightly favour execution cost). Pricing in Google Cloud Functions is slightly more complex by including deployment cost (depending on runtime), but otherwise the per-invocation fee is 0.40 USD per million instances, and minimum 0.23 USD or a multiple thereof for their execution (again without free tiers). Hence, the price effectiveness of FaaS is depending on not wasting memory allocation (the GB factor) [7]. Accordingly, it is less useful to target the already fine-grained billing cycles (the s factor, measured typically in 100 ms intervals or even just 1 ms in Azure Functions) or the practically recessive per-invocation cost. Second, some providers speed up function execution when more memory is assigned, leading to an optimisation problem due to the non-proportionality and runtime dependence of the speedup [5, 6], as well as due to the inability to decouple desired speed from desired memory allocation, leading to allocation waste when needing single-instance performance, and thus to reduced malleability.

FaaS engines and especially commercial FaaS are furthermore subject to three constraints concerning memory which limit the ability to reduce allocation waste. First, most providers support only coarse-grained profiles from pre-defined lists such as <128, 256, 512> MB. Second, most implementations isolate the function execution with Docker containers [2], a technology that only recently allowed dynamic allocation adjustments during the execution, without any known FaaS API pass-through. This contrasts other virtualisation technologies that include ballooning, which can be exploited programmatically [1, 8]. Third, even after deployment, most FaaS do not systematically trace actual versus declared memory consumption to propose or enforce reduced waste reduction, leaving the task to retrieve usage statistics and create performance models [3] to the function engineer. Our work attempts to empower the engineer to solve that task.

WoSC'20, December 7–11, 2020, Delft, Netherlands
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8204-5/20/12…$15.00
https://doi.org/10.1145/3429880.3430094

Hence, we can devise the problem statement: Cloud functions should for technical reasons allocate all the memory necessary to perform their work, following the expected memory utilisation curve over time as close as possible (true *malleability*). They should for economic reasons not allocate more memory, as it is a costly resource (true $pay-per-use$). The problem is then the conjunction of two subproblems: knowing the required amount of memory at any point in time, possibly through speculative means, and enforcing its timely allocation and deallocation, leading to malleable units of execution than are microbilled for actual memory consumption.

An overview table about the different handling of memory in selected FaaS services is given in Table 1. The table omits variations such as Lambda@Edge, GCR and GCF Tier Two that only differ in pricing, not in resource characteristics. No two memory allocation models and pricing models are alike, and further differences exist in the proportional coupling between memory allocation and execution performance (detailed reports are available [4]). These differences underline the non-trivial nature of the problem.

The paper constructs the solution in stages. First, the overall solution approach is presented. Several tracing techniques and tools with varying precision are then introduced along with an elaboration on profiling. Afterwards, the autotuning of containerised function execution and the cost forecasting are discussed, before concluding the work.

2 Solution Approach

From an applied research perspective, we contribute three practical tools. The first, `functracer`, measures the memory consumption of a containerised function execution over time to create trace profiles. The second, `autotuner`, applies the trace profiles and can be activated in subsequent runs of `functracer` to trace the behaviour with memory limits. The third, `costcalculator`, simulates the execution in public FaaS versus advanced adaptive FaaS engines that can exploit the traces, and calculates the potential economic gains.

3 Memory tracing

3.1 Tracing Method

With functions being stateless microservices, the memory needs depend primarily on the input data which must be provided at measurement time. We postulate that given sufficient numbers of traces and the ability to classify input data into a finite set of profiles, the convex hull of all determined memory profiles per data input profile can serve as model for the function's respective maximum memory consumption.

The determination of the profiles shall be conducted as manual or automated profiling based on the correlation of traces and context characteristics. The context involves the input data and the function configuration. We consider an extensive discussion of profiling methods out of scope for the paper, but provide a brief overview about potential strategies as well as a simple convenience method below.

The tracing method is conceptually shown in Fig. 1. Its entry point could be a composite application or an individual function. Given that the majority of publicly known serverless applications (e.g. from AWS SAR) consist of a single function, and that more complex applications often involve mixed technologies (i.e. non-function execution units), our method targets the single function level. The tracing is non-intrusive, so that it works with testbeds injecting artificial input data, as well as, with some restrictions, in production scenarios where the execution characteristics concerning memory use are merely recorded without interference. The restrictions relate to the knowledge of input data, which may need to be recorded through alternative means. Eventually, trace files are produced and post-processed, with or without knowledge about the input data, into execution profiles.

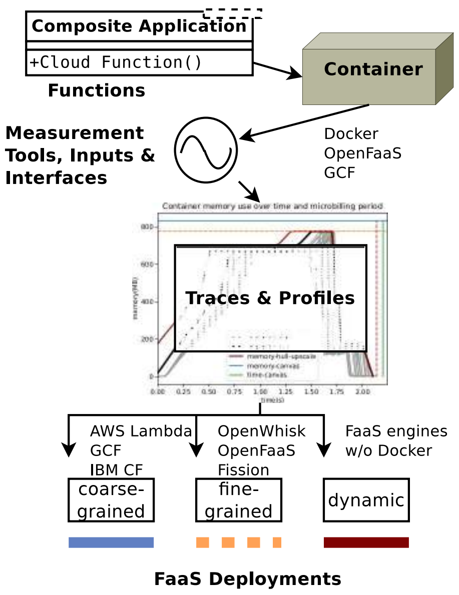

Figure 1. Method to determine containerised function traces and input data-dependent profiles

The ability to trace differs depending on the interfaces. In our work, we consider pure Docker container tracing, OpenFaaS running on Docker (faasd) and Kubernetes (faasnetes), as well as Google Cloud Functions as representative of a commercially operated FaaS. To accommodate the evolving development of Knative, we also cover Google Cloud Run that serves parallel requests to pre-built containers, a model that is known from other FaaS such as IBM Cloud Functions.

Fig. 2 shows an exemplary plot of traces from an existing containerised image scaling function published on Docker Hub (`futils/resize`). Each trace (grey line) slightly differs due to system interference and non-deterministic system behaviour, as well as measurement discretisation (Δ). Eventually, the maximum memory consumption as a hull function

Table 1. Memory considerations in FaaS, default regions and plans, observed in August 2020

FaaS	Memory allocation	Increment	Execution cost $GB \times s$	Performance
AWS Lambda	128 ... 3008 MB	$+ n \times 64$	0.0000166667 USD	implicitly coupled
Google Cloud Functions	128 ... 2048 MB	$+ 2^n, n \geq 7$	0.0000025 USD	explicitly coupled $GHz \times s$
Azure Functions	128 ... 1536 MB	$+ n \times 128$	0.000016 USD	decoupled/fixed
IBM Cloud Functions	128 ... 2048 MB	$+ n \times 1$	0.000017 USD	decoupled/fixed
Alicloud Function Compute	128 ... 3072 MB	$+ n \times 64$	0.000016384 USD	decoupled/fixed
Oracle Functions	128 ... 1024 MB	$+ 2^n, n \geq 7$	0.00001417 USD	decoupled/fixed

over time $m_{max}^{profile}(t)$ with explicit discretisation period δ is determined (black line). This amount of memory would need to be configured with dynamic allocation. FaaS characteristics concerning microbilling (temporal, red/green lines) and memory (spatial, orange/blue lines) are represented as well, loaded from a knowledge base that encodes the content of Table 1. The dashed lines refer to the actually needed maximum memory amount and execution time, and concerning memory refer to the amount that has to be configured if only static but fine-grained allocation was possible. The solid lines refer to the coarse-grained configuration options of the FaaS provider.

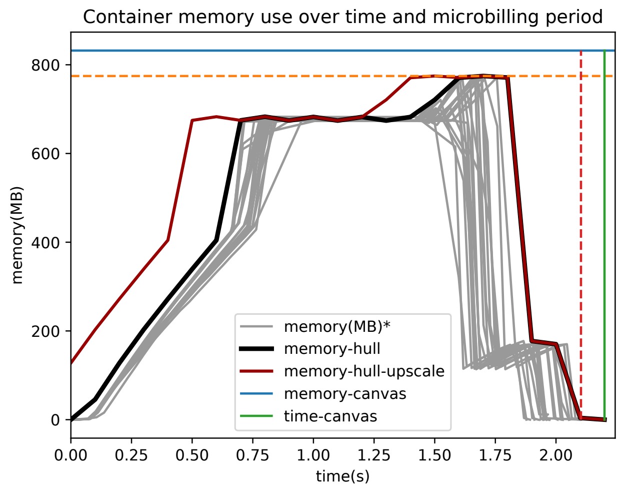

Figure 2. Convex hull ($\delta = \frac{1}{10}s$) for single input profile after 20 traced invocations of containerised image scaling function

Hence, depending on the ability of the FaaS engine, the maximum is statically configured or the profile over time is used for dynamic configuration, taking the vertical scaling time into account. Most functions are executed with an isolation layer that demands a timely pre-allocation of sufficient memory, thus the configuration needs a slightly modified (leftshifted) hull to allow for upscaling in advance, while downscaling happens without shifting (red line). In both cases, a safety buffer is used which may extend to the next permissible configuration limit.

The potential gains – from static coarse-grained allocation to static fine-grained allocation, and further to dynamic allocation - differs depending on the function. In the following, the technical detail of the tracing per interface (Docker, OpenFaaS, GCF/GCR) are explained.

3.2 Tracing for Docker

Containerisation of code execution offers a sweetspot in the trade-off between high enough secure isolation and almost penalty-less performance. In particular the fast startup of containers has made them the technology of choice for isolating function instances in multi-tenant environments. The dominantly used Docker container engine is based on the `containerd` runtime, which in turn interfaces with the `runc` tool. The runtime coexists as daemon alongside `dockerd`, contrasting daemonless approaches such as Podman that directly interface `runc` from the command-line. In turn, the `runc` tool implements the Open Container Initiative specification and controls the operating system commands around containerisation, such as cgroups. For memory tracing, all of these layers offer useful information. Recent versions of the Docker command-line tools offer a convenient high-level interface to the essential memory configuration, which are also used in our work. More accurately, we measure a container's lifecycle status, memory consumption and limits in almost arbitrarily short intervals through a combination of `docker inspect` and the `sysfs` cgroups interfaces, specifically its `usage_in_bytes` and `limit_in_bytes` files. The invocation of `docker inspect` does consume more time than the virtual file system read, and can thus be configured to be performed only every *n*th round. This technique works well even for short-lived functions, yielding measurements with at least 100 ms precision ($\Delta < \frac{1}{10}s$), the equivalent of the microbilling cycles of most FaaS providers. According to our observations, only for very short-lived functions ($t \ll 100$ ms) measurements may fail sporadically but are auto-repeated until they succeed.

Fig. 3 compares the characteristic hull curves of four typical functions implemented in public container images. It is evident that the compression tool would benefit most from fine-grained allocation, the video transcoding would benefit little from dynamic allocation, and the benchmark and sleep would benefit a lot from autotuning, although the benchmark to be divided into two execution profiles despite not having any input data dependency.

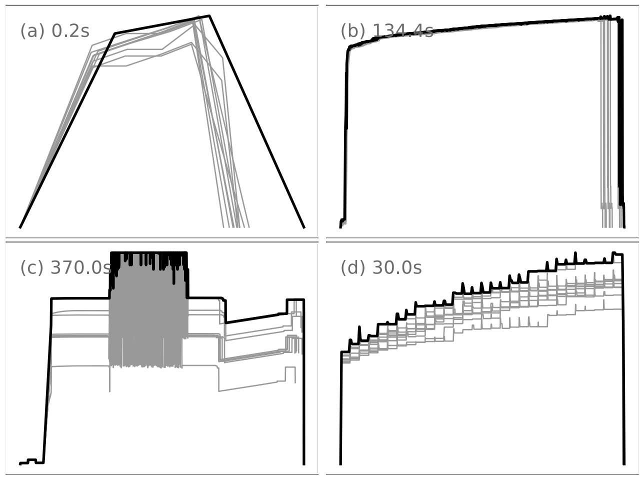

Figure 3. Examples of characteristic trace shapes ($\delta = \frac{1}{10}s$, $\Delta = \frac{1}{27}s$): (a) recursive directory compression, (b) video transcoding, (c) multi-stage Linux benchmark, (d) sleep

The downside is that the results do not necessarily translate into an identical consumption when deployed in a shared FaaS environment, but they offer a first approximation. We contribute the `functracer` tool which performs not only the tracing (based on `docker inspect` and cgroups), but also discretisation, hull determination and hull barrelling for vertical scalers. Further, it performs the plotting of figures as shown above for further analysis by the serverless application provider, and the production of machine-readable memory profiles in CSV and JSON formats.

3.3 Tracing for OpenFaaS and Kubernetes

Along with OpenWhisk, Fn and Fission, OpenFaaS is widely considered one of the more popular open source FaaS engines. Its particular appeal is the ability to run atop Kubernetes for scale, dubbed `faas-netes`, as well as standalone for simple setups or restricted devices, through `faasd`. While the latter option runs containers directly through the system's Docker daemon, allowing to re-use our `functracer` tool, the former option abstracts containers through pods, running them via a `containerd` shim directly in another `runc` namespace. To avoid too many redirections, it is possible to directly retrieve usage statistics through the Kubernetes `metrics-server` which is deployed in the `kube-system` namespace, in parallel to the function pods that run in the `openfaas-fn` namespace. Information is delivered in structured JSON format on memory use and limits, caching and shared memory, as well as on other resources like CPU usage and throttling. On the downside, by default the information delivered via the `metrics-server` log is updated only every 5 s, rendering it unsuitable for precisely determining the memory hull of short-lived cloud functions with dynamic allocation behaviour. Although there is also an emerging `metrics-server` API, it is barely documented, requires complex authentication and has been considered unsuitable at present as interface to memory usage statistics.

3.4 Tracing for Google Cloud Functions/Cloud Run

While OpenFaaS tracing already approximates the memory consumption of a container in a cloud function context, many deployments rely on public FaaS. We consider Google Cloud Platform (GCP) as representative environment. On GCP, there are two serverless compute offerings: Google Cloud Functions (GCF), which accepts function code in various programming languages, produces containers and executes them, and Google Cloud Run (GCR), which accepts and executes pre-built containers.

Tracing and monitoring for both is implemented through Stackdriver, and offered publicly as Cloud Monitoring with a number of predefined resource metrics. The metric of interest for GCF is `function/user_memory_bytes`. The monitoring API requires authentication, making this tracing technique less convenient to set up. Furthermore, the memory consumption data is represented as aggregated value distribution around a mean value covering multiple instances. For GCF, it is only sampled every 60 s and only retrievable with a delay of 240 s, almost defying the possibility to create meaningful allocation profiles.

In GCR, there are two metrics – `container/memory/-utilizations` and `container/memory/allocation_time`. The utilisations measurement is in alpha stage and delays by 60 s, whereas the more mature allocation time measurement delays by 180 s. The sampling frequency for both is 60 s. This is not much better than in GCF regarding the applicability to short-lived functions.

To be able to exploit Cloud Monitoring metrics for function profiling at least heuristically, hundreds of invocations have to be measured and the maximum memory consumption measured at any sampling point determines the lower bound for the function memory configuration. One advantage GCR has over GCF is that although the range is the same, 1 MB stepping is allowed, leading to more fine-grained matching of configured versus actual maximum memory use.

3.5 Profiling Techniques

A typical use case for FaaS is image processing in web applications, often performed asynchronously. We investigated the processing with tracing on Pixelfed, a free social network to share photos similar to Unsplash, Instagram or Flickr. To create meaningful traces, a clustering (binning) of the input data needs to happen. This uses either upfront knowledge, such as the fixed size of avatar (profile) pictures and the usually much larger size of user-uploaded content, or derives the binning via machine learning from a set of training traces.

To demonstrate learned binning, we downloaded 20 randomly selected avatar pictures and 20 content pictures for image processing. In total, 11 are in PNG format and 29 in

JPEG format, with an average size ratio of 1:10.8 between avatar and content pictures, largely confirming the hypothesis. Their sizes overlap however, with the five largest avatar pictures exceeding the sizes of the smallest content picture, making a learned binning technique or explicit avatar/content profile division subject to false positives. Fig. 4 shows the corresponding memory traces, which underline the inability to strictly separate two bins of input data to the processing function.

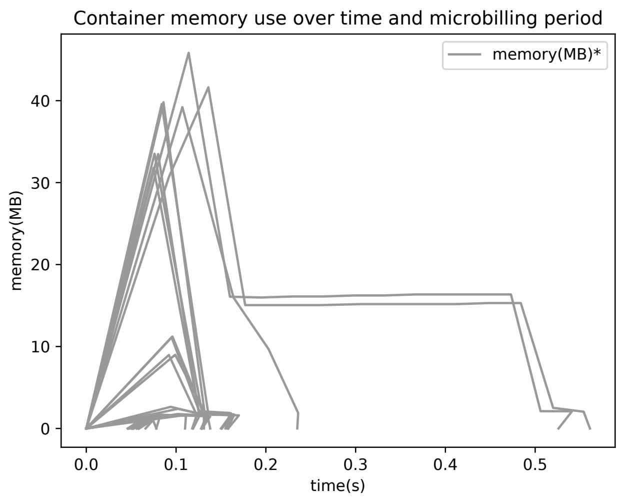

Figure 4. Overlay of 40 image processing memory traces with two desired input profiles

The file size then remains the only relevant input for the binning. When correlating file size to maximum amount of memory needed, the result looks like in Fig. 5. The takeaway for the engineer configuring the cloud functions and understanding if static profiling is applicable is that two function profiles should be deployed, one for avatar pictures with a safe upper bound of 4 MB (if the FaaS provider allows such small values), and one with a presumed upper bound of 64 MB in need of further profile refinements due to the largely uncorrelated memory requirements. As mentioned before, a precise profiling is out of scope but would include additional context information about image formats, source compression levels and content analysis, or deep profiling based on correlation between function variables and memory needs.

4 Autotuning

Autotuning here refers to the ability to dynamically adjust the memory allocation for containers as isolation layer for cloud functions. This brings advantages for FaaS-based software engineers, but also for FaaS providers due to the increased container packing density. Fig. 6 visualises schematically how a host can execute more containers in parallel if it is known in advance that one will release memory that another one will need, so that the sum of all function-level memory needs at a time is less than the available memory.

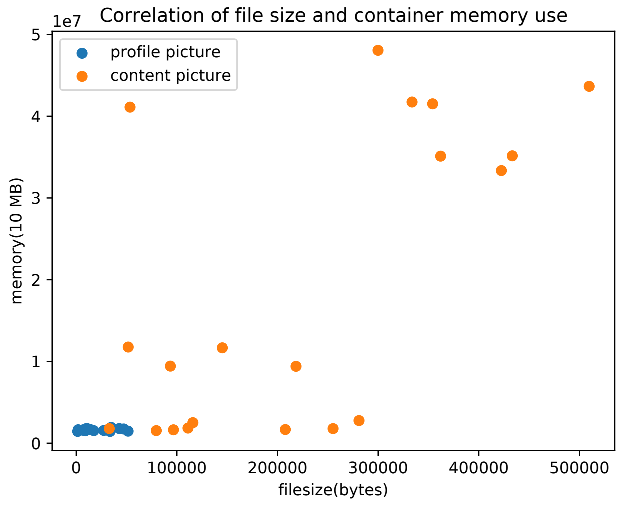

Figure 5. Correlated processing characteristics with file size-based input profiles

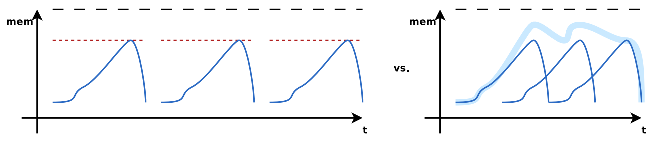

Figure 6. Dense scheduling of containers with memory autotuning – schematic

Autotuning has been implemented in the `autotuner` tool which is based on `docker update`. It runs standalone in production, but can also be hooked into `functracer` to produce plots. Fig. 7 shows how the recorded convex upscaling hull is replayed to adjust memory dynamically (grey line). The amount of wasted memory (difference to blue line) is small and could be reduced further by future work. The initial crossing of the blue line over the grey line is due to initially granting infinite memory to the container which is automatically converted to 0 bytes to maintain plot readability and avoid large memory allocation when the default is always suitable.

Future work plans for autotuning include the interfacing with commercial FaaS management APIs to facilitate the automatic function reconfiguration when behavioural changes are detected, for instance in a continuous deployment and integration scenario.

5 Cost calculation

For an application engineer, a key concern is less the amount of compute resources wasted due to them being mostly hidden, but rather the unnecessary cost associated to the non-optimised application delivery in FaaS. Our tool, named `costcalculator`, re-uses the trace files to perform a monetary analysis. It parses the continuously community-curated

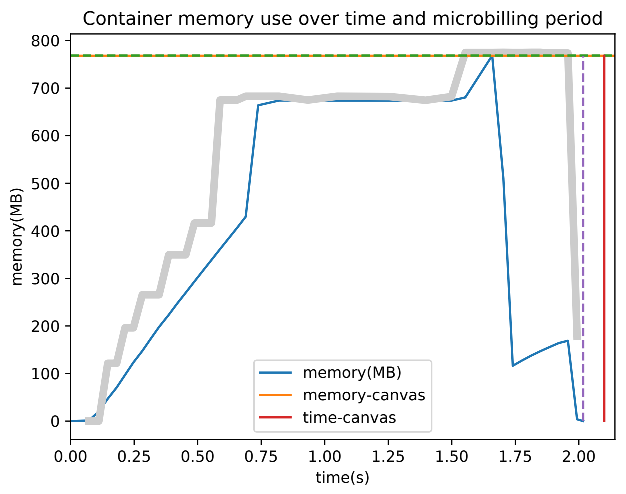

Figure 7. Autotuning with upscaling hull (perceived risk of underallocation around $t = 0.12s$ explained in text)

FaaS Characteristics and Constraints dataset to get declarative pricing information about providers. A sample output for the image processing function used as example in the previous graphs specifically for a comparison with running the same function on AWS Lambda looks as follows:

```
The function used 772 MB of memory and was allocated 832
     MB by AWS.
The total cost for AWS Lambda for 1 million requests per
     month would be 28.63 EUR.
The net cost would be 26.58 EUR, and the overhead cost
     2.05 EUR.
The price increases 7.71% due to wasted memory, and 0.00%
      due to wasted computation time.
0 milliseconds of computation time are being wasted.
60 MB of memory are being wasted.
```

The overhead for static allocation is always between 0% and 50%, whereas when taking dynamic autotuning into account, it is much higher and can reach above 90%, which is significant from a cost or revenue perspective.

6 Conclusions

We have explained how to systematically reduce costly memory overallocation in cloud function execution in two stages. First, applicable to current FaaS, by tracing memory consumption and configuring the minimum possible allocation that fits the maximum required amount of memory over time per function instance. Second, applicable to next-generation FaaS, by dynamically adjusting the memory allocation through means of vertical container resource scaling. Both stages free up memory on the host and therefore reduce cost not only for the cloud function provider, but also for the FaaS provider who can afford a higher deployment density.

Our approach is limited by having to combine coarse-grained resource metrics from FaaS with fine-grained but not necessarily representative metrics from the Docker engine. To overcome this limitation, we expect that future FaaS and cloud monitoring services offer simple interfaces to capture more precise resource utilisation metrics with short intervals.

Resources

All tools are available at https://doi.org/10.5281/zenodo.3911303 and https://github.com/serviceprototypinglab/lambda-docker-measurements. Reference datasets of traces and profiling are available at https://doi.org/10.5281/zenodo.4095480.

Acknowledgments

This material is based upon work supported by Google Cloud, with GCP Research Credits for Serverless Data Integration. The `functracer` and `costcalculator` tools improve earlier implementations co-developed with Daiana Boruta.

References

[1] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2017. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017*, Geoffrey C. Fox (Ed.). IEEE Computer Society, 472–479. https://doi.org/10.1109/CLOUD.2017.67

[2] Christian Bargmann and Marina Tropmann-Frick. 2019. A Survey On Secure Container Isolation Approaches for Multi-Tenant Container Workloads and Serverless Computing. In *Proceedings of the Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2019, Ohrid, North Macedonia, September 22-25, 2019 (CEUR Workshop Proceedings, Vol. 2508)*, Zoran Budimac and Bojana Koteska (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-2508/paper-bar.pdf

[3] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the Costs of Serverless Workflows. In *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, José Nelson Amaral, Anne Koziolek, Catia Trubiani, and Alexandru Iosup (Eds.). ACM, 265–276. https://doi.org/10.1145/3358960.3379133

[4] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance evaluation of heterogeneous cloud functions. *Concurr. Comput. Pract. Exp.* 30, 23 (2018). https://doi.org/10.1002/cpe.4792

[5] David Jackson and Gary Clynch. 2018. An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018, Zurich, Switzerland, December 17-20, 2018*, Alan Sill and Josef Spillner (Eds.). IEEE, 154–160. https://doi.org/10.1109/UCC-Companion.2018.00050

[6] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, Abhishek Chandra, Jie Li, Ying Cai, and Tian Guo (Eds.). IEEE Computer Society, 159–169. https://doi.org/10.1109/IC2E.2018.00039

[7] Kunal Mahajan, Daniel R. Figueiredo, Vishal Misra, and Dan Rubenstein. 2019. Optimal Pricing for Serverless Computing. In *2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019*. IEEE, 1–6. https://doi.org/10.1109/GLOBECOM38437.2019.9013156

[8] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker