EMARS: Efficient Management and Allocation of Resources in Serverless

Aakanksha Saha, Sonika Jindal, Aisha Syed, Jacobus Van der Merwe

University of Utah *

Abstract: We introduce EMARS, an efficient resource management system for Serverless Cloud Computing frameworks with the goal to enhance resource allocation among containers. We have built the prototype on top of an open-source serverless platform, OpenLambda. The prototype implementation uses a novel approach of resource allocation (memory) to the containers running the functions. It is based upon application workloads and function's memory needs. As a background motivation we analyze the latencies of serverless offerings from few serverless offerings (AWS Lambda and Microsoft Azure). We analyzed the effects of memory limits on latencies. The memory limits also lead to variations in number of containers spawned on OpenLambda. Thus we use memory limit settings to present our proposed model of efficient memory management.

1 Introduction

1.1 Background

The cloud industry has seen multiple ways in which the cloud providers can allocate the servers to the customers. There has been a shift from bare-metal servers to virtualized environments to containers. The shift towards containers gives an advantage to developers to provide them the freedom to care about only their application without worrying about the environment and dependencies. Next leap towards this goal is the introduction of "Functions", which are event-driven action based piece of code that runs on serverless platform. These functions execute inside sandboxes based upon the run-time environments such as containers or light-weight VMs. Serverless cloud computing is a model in which the cloud provider manages the execution of applications written in the form of functions. The developers don't need to worry about the management and provisioning of servers. They don't need to define how much storage or how much compute resources are required by the application. Containers provide operating system level virtualization for running multiple isolated systems on a host. This is achieved using linux kernels capability of cgroup[3] (control groups) and namespaces. Many different implementations like Docker[2], LXC[4], etc exist to create Linux containers. These implementations allow configuration of system resources for a group of processes running in containers.

The applications are decomposed into short-lived event-driven functions which can be independently scaled by the cloud provider. It is the responsibility of the cloud provider to manage the resources in efficient manner such that more and more containers (and hence the applications) can be executed at the same server. Thus, cloud providers need to improve the resource utilization by only allocating as many resources as and when required. This in-turn helps customers to only pay for their usage.

There are different ways in which resource allocation and usage can be done efficiently. We define four Rs of efficiency:

*Aiming for HotCloud'18 (https://www.usenix.org/conference/hotcloud18)

1. *Reduce*: limiting the resources given to each container as per the actual needs.

2. *Reuse*: allowing containers to be kept in *warm* [1] state and be made usable to avoid start up time.

3. *Recycle*: killing the containers only when they are not used for a long duration.

4. *Refuse*: not letting new containers get spawned if the system is overloaded.

Overall, the resource efficiency can be achieved if resource availability can be correctly mapped to the application requirements. To perform such efficient allocations, applications should be monitored on memory, latency and the invocations. By such monitoring, various insights can be drawn about the system requirements that can help in capacity planning and scalability.

With an intent to find a method of efficient resource allocation for serverless platforms, we make the following contributions in this paper:

1. Analyzed the behavior of different serverless platforms.

2. Proposed models to efficiently allocate resources based upon the memory usage and workload.

3. Evaluation of the proposed model and opportunity for future work.

1.2 Motivation

We attempted to find, how the resource optimization is done currently on different platforms. This was done by performing tests on different serverless platforms, hands-on experience with some open source platforms and reading through different work in this area.

We performed latency tests and plotted the response times for few initial requests on AWS Lambda [6] and Azure Functions [7]. As seen from the graphs in Figure 1 and Figure 2, both AWS and Azure take longer during the first invocation and the following ones take much lesser time.

The key insight from this experiment was that cloud providers maintain containers in warm state after the execution has completed for the first request. Then the same container is reused so that the startup time for booting and setting up dependencies can be avoided. (Some work have been done to compare the delays in scaling due to different types of startup procedures for containers [15]).

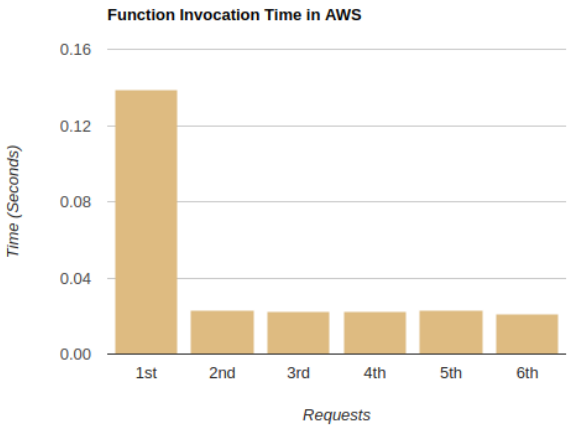

Figure 1: Response Time AWS

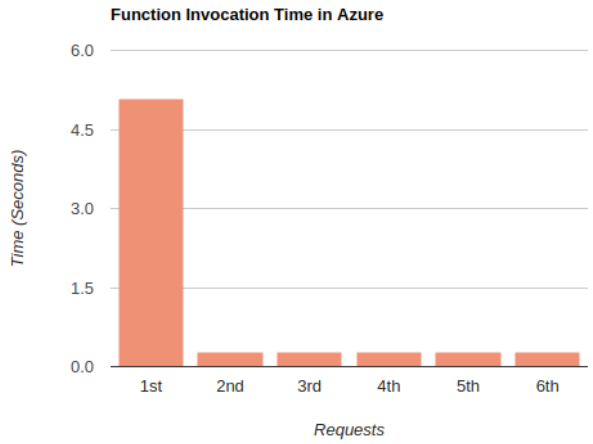

Figure 2: Response Time for Azure

[1] Keeping containers 'warm' means to have the necessary pre-processing step already executed which initializes the container and pull the related dependencies and images.

We also analyzed the code of OpenWhisk[5] and Fission.io[13] to find out the methods used by them. Both of them create pools of warm containers; either generic ones or function specific ones to reduce the start-up time.

Moving on to the analysis of memory requirements, we wrote two functions, one more memory intensive than the other and plotted the response time with different memory limits imposed on them. AWS allows to change the memory limits for the functions: Minimum is 128MB and maximum 3008MB.

The graph in Figure 3 is a plot of memory limit *vs.* latency for two different functions running on AWS. Here the Y axis denotes the response times in milliseconds and X axis represents the memory limit in MBs. At 512 MB memory limit, the mem_alloc function took about 949ms while at 1024 MB, it took only 400ms. Other lambda functions displayed the similar trend of decreasing latencies with increasing memory. But, after a certain point the latency remains constant even if we increase the memory limit to maximum.

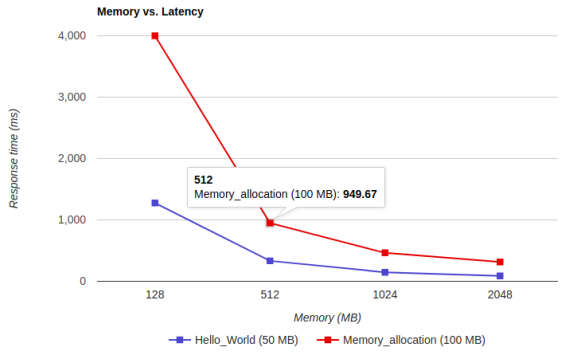

Figure 3: Memory vs Latency

The key takeaway from this experiment was that as we decrease the memory to a function, the latency in responses increase. But when we increase the memory further after an optimal point, there is no effect on latency. Thus, memory limits should be assigned to functions based upon function needs overriding the defaults .

Another experiment was performed with OpenLambda[12] to plot the number of containers spawned by the server with respect to different memory limits. From Figure 4 and Figure 5, we see that memory limits in control groups can affect the number of containers created by the server. If these containers can be reused optimally, a better faster solution can be built.

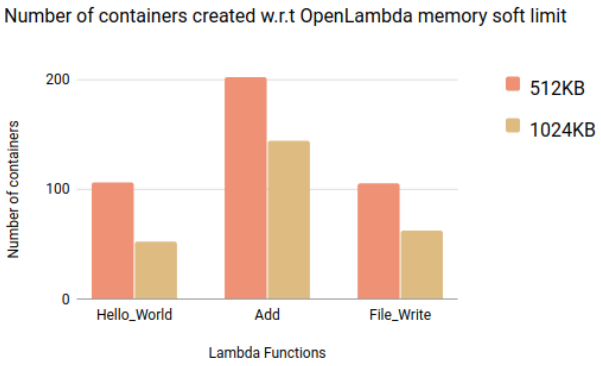

Figure 4: Containers spawned with different cache sizes

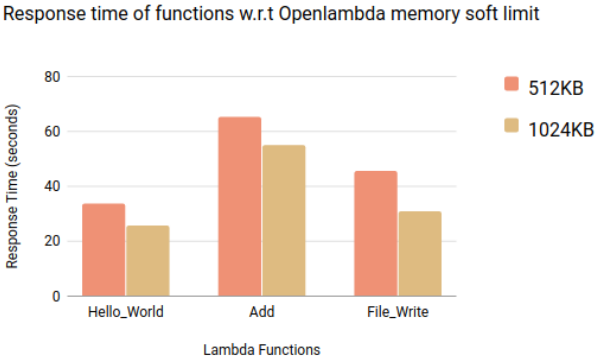

Figure 5: Time taken with different cache sizes

With all these experiments we realized that the memory allocated by default for a particular function might not be optimal and there is a scope to efficiently assign memory at the cost of latency to execute more containers and as a result more functions. It might also be possible to limit the resources

to functions if the system is under memory starvation.

In the next section we describe our proposed solution in detail.

2 Implementation

2.1 Solution

To realize resource allocation when the system requirements are not known beforehand, applications need to be monitored for their usage of resources. By monitoring the resource usage, a fair idea can be derived about the application needs. This can help in capacity planning and scalability. EMARS is our solution for such monitoring on the basis of memory needs and invocations of the functions. EMARS is a pluggable architecture which runs parallel to the server's logic of executing applications. It collects information about different parameters from the request to functions and generate optimal configurations to be used while handling further requests to those functions. This can be done *Predictively* as well as *Reactively*.

1. Predictive: Log the requests and predict the configurations for future invocations based upon the current usage.

2. Reactive: Update the resource limits dynamically based upon the system load and bursts in traffic conditions.

2.2 Approach

Our solution is based on top of the open source serverless platform OpenLambda [12]. It is a platform for building next-generation lambda services and applications in the burgeoning model of serverless computation. It is written in Go language. The goal of this project is to enable exploration of new approaches in serverless architecture.

OpenLambda creates a separate cgroup for its docker containers and allocates default memory limits to the function handlers(containers). For efficiently allocating memory to the containers, we propose two *Predictive* models; **workload-based and memory-based** for spawning containers with optimal resource restrictions. These models will be used to recommend new cgroup configurations at run time for the docker containers running the lambda functions.

Control groups, or cgroups [3], is a kernel feature introduced to provide a way of limiting access to system resources for a group of processes. Serverless platforms run functions on the same server with shared resources. So, servers need to provide resource isolation [memory, CPU, disk and network] to the function handlers running on the same server. Control groups allow monitoring of the group's configurations, deny access to certain resources, and even reconfigure them dynamically on a running system. These groups are attached to one or more subsystems, which corresponds to single resource like memory, cpu, etc. The memory subsystem is of particular interest to us for our solution. It allows us to set limits on memory usage of the containers dynamically as well as while loading. Few of the cgroup parameters from the memory subsystem which we use to efficiently manage the container spawning are:

1. memory.max_usage_in_bytes: It reports the maximum memory used by processes in the cgroup (in bytes).

2. memory.limit_in_bytes: It sets the maximum limit of memory that can be used by a process.

3. memory.oom_control: If enabled, tasks that attempt to consume more memory than they are allowed are immediately killed by the Out of Memory (OOM) killer.

The next section describes our proposed EMARS model build on top of OpenLambda for predictably setting fine grained memory limits based on application requirements.

2.3 Overview

OpenLambda mirrors the Lambda model of AWS where, Lambda handlers or function handlers from different customers share common pools of servers managed by the cloud provider, so developers need

not worry about server management. Figure 6 shows the flow diagram of OpenLambda: the left blocks correspond to the current workflow of and the right blocks correspond to our new EMARS predictive solution based upon which the container instantiation block of OpenLambda is modified. As we can see in Figure 6 that whenever the user issues an http request, the server (handling the function) pulls the appropriate lambda handler from the function repository. After that it goes to the next if block and verifies if a container is already running for this handler; if yes it unpauses it and serves the request and if not it creates a container and serves it. In the basic flow the creation and execution of new the container is based on the default cgroup configurations for the memory subsystem.

2.4 Proposed EMARS solution

As a step towards efficient allocation of memory to the containers we propose two models: the workload-based model and the memory-based model. These models log different parameters based upon the http request invocations and feed this report in a configuration file at the end of a certain period. Currently this period is set to 24 hours. So, from Figure 6 we can see that the thread running workload based model and the thread running memory based model feed there data to the config generator thread which generates the optimal memory configurations per lambda function and recommends these configurations to the creation and execution of containers. Going into the details of each of these threads:

1. **Memory based model:** In this modeling we capture the memory requirements of each function throughout the day. We use the command *docker stats* to retrieve the memory usage of the function and log the maximum usage. At the completion of 24 hours this data is fed to the config generator thread that recommends memory optimization for the new container to be spawned. The following is a pseudo code for memory based modeling.

```
while True{
   if(time.Hour() == 24){
     Send signal to config generator thread;
}
   else{
 Run docker stats;
 Map the container IDs to the function names;
 Log the max memory usage by each function;
 sleep 10;
   }
}
```

2. **Workload based model:** Knowledge about the workload is an important aspect for scheduling of resources. So, in this method we log the number of requests for each function throughout the day. At the end of 24 hours we send the details to the config generator thread which stores this value for every function. The logging can be done at different granularity levels like on hourly basis. This will be retrieved for any new function call on the next day. The following shows the pseudo code for workload based modeling.

```
while True{
   if(time.Hour() == 24){
  Send signal to config generator thread;
}
   else{
 Log the number of requests for each function;
   }
}
```

3. **Config Generator:** This thread keeps on receiving the data from the the memory based and workload based model and generates a configuration file for each lambda function. An example can be: {*Number_of_request, Maximum_memory*}. With this setting in place every time a lambda function is invoked it will read the corresponding configuration file and based on that it will restrict/allocate resources (using the memory subsystem of cgroup) to the container serving the function.

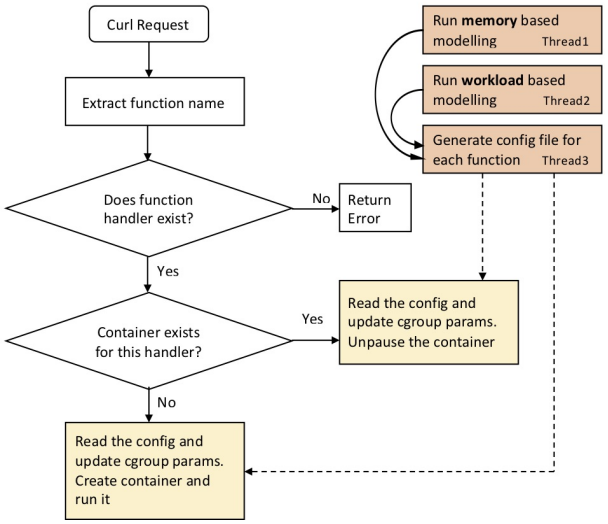

Figure 6: OpenLambda: Proposed workflow

Thus, with this predictive model of EMARS we aim to generate the optimal configurations for the containers based upon application usage and requirement. Currently the workload based modeling along with the configuration file generator is completed and integrated with the OpenLambda code. A separate script for memory based modeling has been written which can dynamically update the memory configurations of containers. We aim to provide Reactively updating resource limits based on the machine load as our future work.

3 Evaluation

The results achieved from our implementation of dynamic memory allocation were in sync with the behavior of AWS lambda. It was seen that reducing the memory increases the latency of lambda functions and vice versa.

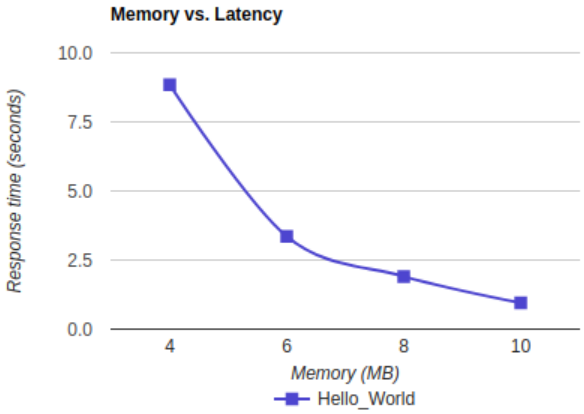

Figure 7: OpenLambda: Hello_World function

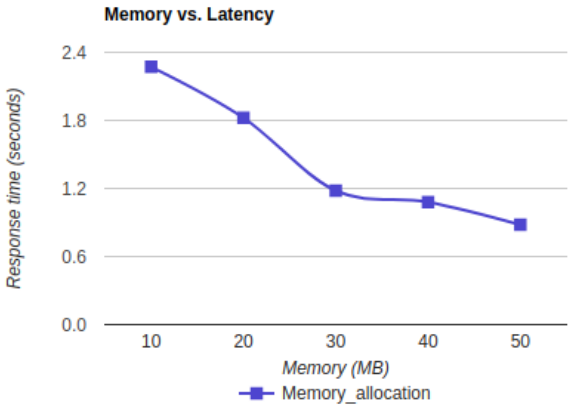

Figure 8: OpenLambda: Memory_alloc function

Test Environment: The tests were performed on CloudLab Server using a virtual machine (VM) having Ubuntu 16.04. The OpenLambda code version v.1.0.0 was downloaded and configured on the VM. Two lambda functions *Hello_world* and *Mem_alloc* were written in Python. **Results:** The http request to the *Hello_world* function simply responds with a "Hello" message while *Mem_alloc* is a memory intensive function which creates an array of 1 million integers and does array operations. The latencies with varying memory limits were captured by sending http requests from a remote system to instantiate these

lambda functions.

Figure 7 shows the Memory vs. Latency graph for *Hello_world* function and at 6MB the latency of the function is 3.3 seconds while at 10 it's close to 1.2 seconds. And the least memory which can be assigned to this function is 4 MB but it comes at a cost of latency, which is around 8 seconds. Similarly Figure 8 shows the plot for *Mem_alloc* function. Since it is a memory intensive function the memory requirements of this function is more. A stark difference in response time can be noticed when the function handler was given 50 MB *vs.* 10 MB.

The key insight from the evaluation results are: restricting memory to a function downgrades the performance by increasing latency but assigning memory limits based upon the function needs can help in efficient management and allocation of resources.

4 Related work

There have been work done in analysing the latencies due to container allocation and start-up [11], [15]. To enable the serverless technology, there are still some gaps to be filled in terms of monitoring and analysis which is done by various tools. There are tools like [16] to monitor the function performance specific to AWS Lambda. OpenLambda [12] compares the response times of lambda functions *vs.* elastic BS[1] virtual machines. Wes Felter et al. compare the containers and VM performance in [10].

To our knowledge, most of the platforms perform resource scaling using eager pooling and resource shrinking like in OpenWhisk[5] and Fission.io[13]. An initial pool of containers is allocated and based upon the demand the pool is expanded or shrunk.

There have been work in optimizing the cloud for the business needs of the cloud [14], enhancing security in containers [8] etc. Also some work has been done in optimizing resources by using cgroups like in [9]. Our solution is done specifically for serverless platforms with the goal to keep memory utilization optimal.

5 Discussion and possibilities

With this work, we created a model where memory and workload can be monitored to predict the future requirements. In our current implementation, memory limits are set by our code dynamically which runs independent of OpenLambda. As an immediate goal, this dynamic update to memory needs to be integrated with OpenLambda to make it a complete solution. Optimal method to generate the configuration files for each function, based upon the logged information needs to be worked on. For workload based modeling, a threshold value needs to be figured out based upon further tests. For example if there are 500 calls to a function assign 1024MB as soft limit, so that more containers can be spawned.

For long term goals, we can build a reactive methodology as well which can work in conjunction to the predictive method to change the limits as per the machine load or as per the real usage if it is different than the prediction. This will provide elastic scaling. Like AWS, an infrastructure can be built to let users specify their requirements. User parameters can be given higher priority. Apart from this applying machine learning techniques to predict optimal memory requirements can be another direction of future work.

6 Conclusion

In this work we presented an analysis of resource allocation done in various serverless platforms. We also presented effects of resource allocation techniques used in some open source serverless platforms. Further, we proposed models based upon OpenLambda for resource allocation as per the learnings from variety of loads. By performing different tests, we also concluded that with changing memory limits, latencies are also changed. Thus, the trade off between memory and latency is the important insight we draw from our work and finding an optimal value of memory usage which guarantees the right latency is the "sweet spot" which we proposed to exploit.

References

[1] AWS Elastic BeanStalk. https://aws.amazon.com/elasticbeanstalk/.

[2] Docker Containers. https://www.docker.com/.

[3] Introduction to control groups(cgroups). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[4] Linux Containers. https://linuxcontainers.org/.

[5] Apache/IBM OpenWhisk. IBM. http://openwhisk.incubator.apache.org/.

[6] AWS. Amazon aws. https://aws.amazon.com/.

[7] Azure. Microsoft azure. https://azure.microsoft.com/en-us/.

[8] Bacis, E., Mutti, S., Capelli, S., and Paraboschi, S. Dockerpolicymodules: Mandatory access control for docker containers. In *CNS* (2015), IEEE, pp. 749–750.

[9] Bellasi, P., Massari, G., and Fornaciari, W. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Trans. Embed. Comput. Syst. 14*, 2 (Mar. 2015), 39:1–39:17.

[10] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (March 2015), pp. 171–172.

[11] G, M., and PR, B. Serverless computing: Design, implementation, and performance. http://www.serverlesscomputing.org/wosc17/#p4, 2017.

[12] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. *HotCloud'16 Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (2016), 33–39.

[13] Kubernetes Fission. Serverless Functions as a Service for Kubernetes developed by Platform9. http://fission.io.

[14] Litoiu, M., Woodside, M., Wong, J., Ng, J., and Iszlai, G. A business driven cloud optimization architecture. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, ACM, pp. 380–385.

[15] Nadgowda, S., Suneja, S., and Kanso, A. Comparing scaling methods for linux containers. In *2017 IEEE International Conference on Cloud Engineering (IC2E)* (April 2017), pp. 266–272.

[16] Pipe, I. See inside your lambda functions. https://www.iopipe.com, 2017.