MAFF: Self-Adaptive Memory Optimization for Serverless Functions

Tetiana Zubko, Anshul Jindal[0000−0002−7773−5342], Mohak Chadha[0000−0002−1995−7166], and Michael Gerndt[0000−0002−3210−5048]

Chair of Computer Architecture and Parallel Systems,
Technical University of Munich, Garching, Germany
{tetiana.zubko, anshul.jindal, mohak.chadha}@tum.de, gerndt@in.tum.de

Abstract. Function-as-a-Service (FaaS), a key enabler of serverless computing, has been proliferating, as it offers a cheap alternative for application development and deployment. However, while offering many advantages, FaaS also poses new challenges. In particular, most commercial FaaS providers still require users to manually configure the memory allocated to the FaaS functions based on their experience and knowledge. This often leads to suboptimal function performance and higher execution costs. In this paper, we present a framework called MAFF that automatically finds the optimal memory configurations for the FaaS functions based on two optimization objectives: cost-only and balanced (balance between cost and execution duration). Furthermore, MAFF self-adapts the memory configurations for the FaaS functions based on the changing function inputs or other requirements, such as an increase in the number of requests. Moreover, we propose and implement different optimization algorithms for different objectives. We demonstrate the functionality of MAFF on AWS Lambda by testing on four different categories of FaaS functions. Our results show that the suggested memory configurations with the Linear algorithm achieve 90% accuracy with a speedup of $2x$ compared to the other algorithms. Finally, we compare MAFF with two popular memory optimization tools provided by AWS, i.e., AWS Compute Optimizer and AWS Lambda Power Tuning, and demonstrate how our framework overcomes their limitations.

Keywords: serverless · cost optimization · memory optimization · duration optimization · Function-as-a-Service · memory allocation

1 Introduction

In recent years, the popularity of serverless computing technology has been proliferating in different domains [13,26,18]. Cloud users profit from the automatic scalability, faster deployments, and the possibility to outsource control and maintainability over the underlying hardware infrastructure to the cloud service providers [28,17]. Function-as-a-Service (FaaS) is a key enabler of serverless computing [29]. In FaaS, a serverless application is decomposed into simple, standalone functions uploaded to a FaaS platform such as AWS Lambda, Google Cloud Function (GCF), and Azure Functions (AF) for execution [28]. The pricing is charged based on the number of requests to the functions and the execution duration [10].

However, while offering many advantages, FaaS faces some challenges that hinder its widespread adoption [11,16,27]. While most infrastructure management is abstracted away from the user, major commercial FaaS providers still require users to manually configure the amount of function memory allocated to the FaaS function [29]. For most developers, this is often done by using their experience and knowledge, leading to suboptimal function performance and higher function execution costs. Furthermore, the cost of the FaaS function depends on the execution duration of the code and assigning the smallest or random memory can be considered as an anti-pattern [10,22,15]. Thus, the user has to do a trade-off analysis between them to define the suitable configuration for their required SLOs [29], and it's not trivial to find the optimal configuration where the overall cost and execution duration are both optimal.

The importance of optimizing memory configuration for the FaaS functions has already been described in various scientific works and implemented in practice [3,30]. However, the existing tools are either only implemented to be actively invoking the analyzed functions [12] or require functions to have specific settings and execution frequency to be able to provide the result [9]. To this end, we develop **MAFF** (**M**emory **A**llocation **F**ramework for **F**aaS functions), a python-based framework for automatically finding the optimal memory configurations for the FaaS functions. It is implemented in two execution modes – *active* and *passive*, depending on the way of how the function execution information is received. Our key contributions are as follows:

- We develop and present a framework called **MAFF** that automatically finds the optimal memory configurations for the FaaS functions (§3). Furthermore, it automatically self-adapts the memory configurations for the FaaS functions based on a change in the function input or other user requirements.
- We propose and implement three optimization algorithms – *Linear*, *Binary*, and *Gradient Descent*, for the minimum cost optimization objective, and two optimization algorithms – *Optimization value*, and *Duration Change*, for the balanced (balance between cost and execution duration) objective (§2).
- Although our approach is generic and *MAFF* can be easily extended to support other commercial and open-source FaaS platforms, we demonstrate the functionality of *MAFF* with AWS Lambda (§5) on four FaaS functions.
- We compare *MAFF* with other existing memory optimization tools: AWS Lambda Power Tuning [12] and AWS Compute Optimizer [9].

2 Methodology

According to business requirements, there are different optimization objectives when using FaaS functions. For example, it is essential to ensure a quick function execution in some cases. In other, the balance between the function's execution and the cost plays a more significant role. Therefore, we have considered two optimization goals:

- **Cost-only**: In this case, the users' primary goal is to minimize the cost of the function execution even if the duration is not the lowest.
- **Balanced**: It finds the balance between the cost and execution duration of the function. Here the goal is to find the best possible performance for a fair cost.

Algorithm 1: Linear Algorithm

```
   Input: start_mem, step_size, threshold_count, function
   Output: min_cost_mem
 1 step_count = 0, dur1 = getDuration(function, start_mem);      // get the duration
 2 min_cost_mem = start_mem, min_cost_dur = dur1;
 3 for step_count ≤ threshold_count do
 4 |   old_cost = (dur1 × start_mem);
 5 |   new_mem = start_mem + step_size;
 6 |   dur2 = getDuration(function, new_mem);
 7 |   new_cost = (dur2 × new_mem);
 8 |   if new_cost > old_cost then
 9 |   |   min_cost = min_cost_mem × min_cost_dur;
10 |   |   if old_cost ≤ min_cost then
11 |   |   |   min_cost_mem = start_mem;
12 |   |   |   min_cost_dur = dur1;
13 |   |   else
14 |   |   |   step_count += 1;
15 |   |   end
16 |   end
17 |   dur1 = dur2, start_mem = new_mem;
18 end
19 return min_cost_mem ;                          // return the min cost memory
```

In the scope of this work, we developed multiple algorithms for each of the optimization goals. In the following subsections, we describe each of the algorithms.

2.1 Cost Optimization

Linear Algorithm: The main idea behind this algorithm is to continuously increase the memory allocated to the function linearly and calculate the cost for each memory configuration until a memory sweet spot is found where the optimization goal, i.e., the cost, is minimum. The pseudocode for this algorithm is shown in the Algorithm 1.

By default, it starts at the minimum memory configuration possible in AWS - `128MB` (*min_mem*) and increases the allocated memory with a pre-defined step size of `128MB` (*step_size*). Firstly, the memory of the function is set to *min_mem* and then the execution duration of the function at that memory is determined (Line 1). We further determine the cost by multiplying the allocated memory and execution duration at *min_mem*, since the cost is directly proportional to them [10] (Line 4). We then continuously increase the function's memory by *step_size* (Line 5) and determine the new execution duration of the function at that memory, and then the cost (Lines 6-7). If the cost with the new memory configuration is smaller than the previous one, the algorithm moves to the next memory iteration (Line 16). If not (Lines 8-15), the previous memory point is a minimal cost point, and the algorithm stops. However, in such a case, the algorithm could stop in the local minima. Thus, the additional logic of overcoming the local minima was added. The algorithm does not stop execution when the first local minima is found, but

continues for a few more iterations until a threshold (*threshold_count*) is reached. The higher the value of the *threshold_count*, the more precise result can be delivered, but at the same time, more iterations will be performed.

Furthermore, typically the cost of the Lambda function stays the same with minor fluctuations until some memory level, after which it starts increasing almost linearly [20]. Following the Pareto optimization principle, when two memory configurations have the exact cost, a memory with the bigger value is selected, as it positively affects the function's execution duration.

Binary Algorithm: This algorithm is based on the classical binary search algorithm, which operates on a sorted list of numbers by iteratively comparing the searched item to the middle element of the list and eliminating parts in which the searched element can not be found. The same principle is borrowed to create this algorithm.

For finding the optimal memory, this algorithm first calculates the execution cost at the start and the middle memory configurations from the provided memory list. The user can define the memory list; by default, it is the whole range of memory values available on AWS (from `128MB` to `10240MB`) [6]. Suppose the cost at the start memory configuration is lower than the middle memory configuration. In that case, the algorithm continues execution on the left part of the memory array (from start to middle), otherwise on the right part. The stopping criteria for the algorithm is when the memory at the start of the analyzed memory interval is equal to the memory in its middle, meaning that the interval consists of only one value.

Gradient Descent Algorithm: This algorithm is based on the popular Gradient Decent optimization algorithm in Machine Learning. The idea is to continue finding the minimum of a metric by choosing the direction (left or right direction) towards the minimum cost at each iteration until the minimum is reached.

In this algorithm, a random memory value from the provided memory list is selected at which the cost metric is calculated along with the cost of its left neighbor. If the cost of the neighbor is higher than the cost of the current point, the algorithm continues execution on the right side of the current point (in the direction of decreasing cost, otherwise on the left side. The minimum cost is also updated if the current cost is less than the minimum cost.

The known issue with the *Gradient Descent* algorithm is that it can get stuck in the local minimum [24]. To overcome this problem, an additional counter *step_count* was added. The counter *step_count* is updated when a local minimum is reached. It is used to control that the algorithm does not stop in the first minimum that it encounters but continues execution until a threshold *threshold_count* is reached. The neighbors of the current memory configurations are found by the addition or subtraction of the memory *step_size* from the current memory value.

2.2 Balanced Optimization:

In the following paragraphs, we describe two algorithms for balanced optimization goal.

Optimization-Values-Based Algorithm: The first approach to finding such an optimal point is to transform cost and execution duration into percentage format using the maximum value of cost and duration, respectively. To avoid the exhaustive search of finding maximum values [12], it is assumed that the function has maximum execution

duration at the beginning of the memory list (*mem_config_list*), i.e., `128MB` and the maximum cost at the end of the list, i.e., `10240MB`. The assumption is based on the fact that increasing the allocated resources does not negatively influence a function's performance, but make its execution faster [7,20] by having more underneath resources.

The algorithm starts analyzing memories starting from `128MB` and increases memory allocated to the function with the defined memory *step_size*. For each memory configuration, the algorithm calculates a value called *optimizationValue* shown in the Equation 1. Memory configuration having the lowest *optimizationValue* is selected as the optimal memory spot with the balanced optimization goal. As part of Equation 1, we introduce an additional parameter, α, by which the influence of duration and cost on the final result can be adapted. The values of the parameter can range between 0 and 1. When α is equal to 0, the algorithm goal corresponds to the cost optimization, and when the α is set to 1, it will be optimizing the duration. By default, the parameter value equals 0.5, which means that both cost and duration are equally important, and a balance between them needs to be found.

$$\textbf{optimizationValue} = \frac{\alpha \times duration}{maxDuration} + \frac{(1-\alpha) \times cost}{maxCost} \tag{1}$$

where α is the coefficient for adjusting the influence of optimization variable (cost or duration) on the final result, and the other variables are self-explanatory from their names. The algorithm operates similarly to the Linear Algorithm, but uses *optimizationValue* as the optimization parameter. It also contains the logic of overcoming local minimums, as explained in other algorithms.

Duration Change Algorithm: This algorithm is based on the fact that, the optimal memory spot for balanced optimization goal is a point after which any additional memory increase does not provide any significant performance improvement [14]. So, the idea of this algorithm for *balanced* optimization is to incrementally increase a function's memory configurations until there is no significant improvement in its execution duration. In other words, we need to find a vertex of a hyperbola, a point at which the logarithmic curve of the function's execution duration bends. This algorithm tries to find a memory configuration after which the duration curve has flattened, and subsequent increases in memory will not significantly improve the function's performance.

The algorithm operates similarly to the Linear algorithm. By default, the algorithm starts with memory `128MB` and compares execution duration at this point to the execution duration of the next point on the right side. The memory value of the right neighbor is equal to the current memory plus the defined memory *step_size*. If the duration of the right neighbor is decreased by more than the defined change threshold percentage (γ), the algorithm continues execution for the next iteration; otherwise, execution stops. The default value of the γ is 10%, the higher the value, the closer the memory will be selected to the hyperbola vertex.

3 MAFF Framework

In this section, we present **MAFF** (**M**emory **A**llocation **F**ramework for **F**aaS functions), a python-based framework for automatically finding the optimal memory configurations for the FaaS functions according to the defined optimization goal.

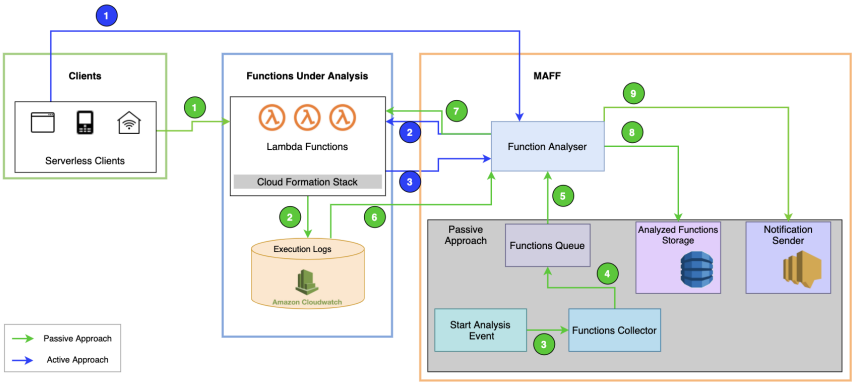


Fig. 1: High-level system architecture and workflow of *MAFF*

Figure 1 shows the high-level system architecture of *MAFF*, its components, and the workflow between them. All the components are developed in Python and deployed on AWS infrastructure. On the high level, there are two main approaches for executing *MAFF* – *active* and *passive*, differentiated by the method of how the function's execution information is gathered.

Active Approach: In the *active* approach, *MAFF* invokes function by itself. A short execution log is returned synchronously after each execution. Blue lines in Figure 1 represent interactions between parties when using *MAFF* in the *active* approach. As can be seen, as soon as a trigger event is received by the *Functions Analyzer* (§3.1) including the optimization goal (step 1), it performs requests to the FaaS function to find out its execution duration and cost at different memory configurations (step 2). Then it collects execution logs and uses different algorithms described in §2 for finding the optimal configuration (step 3). Once the configuration is found, it is then saved for the function, and *Functions Analyzer* stops its execution.

Passive Approach: In the *passive* approach, *MAFF* does not send requests to the analyzed function but relies on the real user's traffic to receive information about the function's execution. In this case, *MAFF* observes CloudWatch logs, which are generated by the Lambda function, when users invoke it. The *passive* approach of *MAFF* is developed for such scenarios, where it is not possible or not cost-efficient to actively invoke the Lambda function (e.g., if the function creates new products to the online store or adds the users to the database). When executing *MAFF* in the *passive* approach, additional components such as *Start Analysis Event*, *Functions Collector*, *Functions Queue*, are used. The flow of the passive *MAFF* approach is marked with the green lines in the Figure 1 and starts automatically when the scheduled CloudWatch event containing optimization goal is triggered (step 2). This event is configured to invoke the *Function Collector* Lambda function (step 3), which gathers Amazon Resource Names (ARNs) of stack functions and adds them into an Amazon Simple Queue Service (SQS) queue (step 4). Every new item in the queue is processed by *Functions Analyzer* Lambda for finding the optimal memory configuration at the defined optimization goal (steps 5 -

(7). If *Functions Analyzer* can identify the optimal memory for the function, it adds a record into the DynamoDB database to avoid unnecessary analysis in future iterations (step (8)). *Notification Sender* sends an email notification if the memory configurations proposed by *MAFF* are significantly different from the initial configuration (step (9)).

Both *active* and *passive* approaches can adapt memory of the analyzed function in real-time on AWS Lambda as soon as the optimal memory configuration is found. Such self-adaptive configuration is performed with the help of AWS SDK for Python (Boto3). *MAFF* is a language-agnostic tool, and it can analyze any Lambda function, regardless of the programming language used for source code.

3.1 MAFF Components

Internally, *MAFF* consists of several components, each of them is based on a specific AWS service. In the following subsections, we describe its components in more detail.

Function Analyzer: This component contains the main logic of the *MAFF* and is used in both *active* and *passive* approaches. In the *active* approach, Function Analyzer sends requests to the function to generate execution logs at different memory configurations. In contrast, in the *passive* approach, it just reads all the function's logs created when users invoke the function. Further, it is responsible for analyzing those logs of the function and selecting its optimal memory configuration based on the given optimization goal and the algorithms described in §2. *Function Analyzer* itself is deployed as a Lambda function with `512MB` memory and 10 minutes timeout. As input, it expects the Amazon Resource Name (ARN) of the Lambda function to be analyzed.

Start Analysis Event: It is used to invoke *MAFF* in the *passive* approach. It is implemented as a scheduled AWS CloudWatch event rule, which triggers an analysis process based on the time interval specified by the user (e.g., every four hours).

Functions Collector: It is responsible for gathering ARNs of the functions which belong to a CloudFormation stack and need to be analyzed. This component is also implemented as a Lambda function with `512MB` memory and 10 minutes timeout. As input, this function receives the name of the CloudFormation stack.

Functions Queue: It stores the list of functions' ARNs generated by *Function Collector* before they are passed to *Function Analyzer*. It is implemented with Amazon Simple Queue Service (SQS) and uses *Function Analyzer* as a Lambda trigger.

Analyzed Functions Storage: This component stores the past optimal memory configurations of the functions found in the previous *MAFF* executions. It is implemented using the AWS DynamoDB database with function name as the primary key. This component acts as the cache, and if the function optimal memory configuration exists in the database, then the unnecessary iterations of the algorithm are avoided.

Notification Sender: It sends an email notification if the memory configurations proposed by the *MAFF* are significantly different from the initial configuration. By default, the notification will be sent if the memory selected by *MAFF* is four times lower or higher than the initial one.

4 Evaluation Settings

We test the proposed *MAFF* framework for FaaS functions deployed on AWS Lambda, a popular serverless cloud platform. *MAFF* framework itself was deployed on AWS lambda as described in §3.1. Each of the algorithms introduced in §2 are executed 5 times on each of four different benchmark functions (§4.1). We also describe the evaluation scenarios conducted to evaluate *MAFF* (§4.2).

4.1 Benchmark Functions

In the evaluation, we have considered four types of functions. All of them are implemented in Python 3.8, which is one of the most popular languages used in AWS [28]. Moreover, each function was configured with a three-minute timeout, which allows them to finish execution with any memory configuration.

CPU-Intensive Function: CPU intensive functions have a logarithmic dependency between allocated memory and execution duration of the function [14]. For the test purposes of this work, a specimen CPU-bound function was created. It calculates tangent and arctangent for the numbers between 0 and 8^7.

I/O-Intensive Function: I/O function used in this work is based on a popular Linux utility for the file operations - dd [23]. Using dd, an input file `/dev/zero` is copied to an output file `/tmp/out` using 50 blocks, each of 512 bytes size. The file `/dev/zero` represents an unlimited flow of null characters.

Memory-Intensive Function: Memory bound function used in this work consists of a for-loop iterating from 0 to, 100000. Every iteration adds a number to the initially empty array, thus slowly filling up the memory.

Network-Intensive Function: Here a large JSON file over the Internet is read.

4.2 Evaluation Scenarios

We design our experiments to answer the questions:

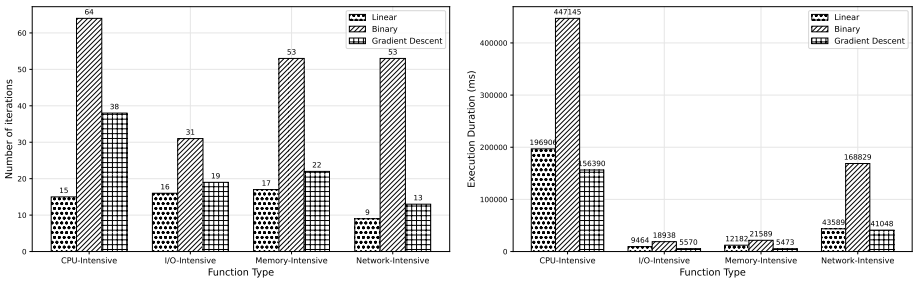
Q1. Optimal configuration finding efficiency : how efficient are the *MAFF* algorithms in finding the optimal memory configurations for various types of functions at different optimization goals?

Q2. Optimal configuration finding accuracy: how accurate are the *MAFF* algorithms in finding the optimal memory configurations given the optimization goal?

Q3. Active vs passive approach: how do the two approaches in *MAFF* compare against each other in terms of accuracy?

5 Results

In this section, we present the results of the evaluation scenarios described in §4.2.



(a) Number of iterations (b) Execution duration

Fig. 2: The required number of iterations and the execution duration of the various algorithms for the cost optimization objective.

5.1 Q1. Optimal configuration finding efficiency

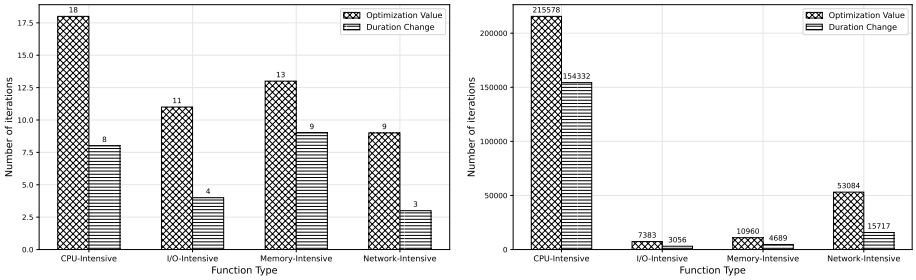
Algorithms are compared based on the number of iterations and time taken by them.

Cost Optimization: Figure 2a shows the number of iterations that each of the algorithms performed to identify optimal memory configuration with the *cost optimization* as the minimization objective. For every function type, the *Linear* algorithm managed to find a minimal *cost* point with the least number of iterations. The *Binary* algorithm, in all cases, took the most steps to find a memory sweet spot. It can be explained by the fact that memory points with minimal cost for all function types lay in the region `128MB - 1280MB`. But *Binary* algorithm was executed on the whole memory range (`128MB - 10240MB`), which took more steps to narrow the search to the correct memory region. For every function type, *Gradient Descent* required more steps than *Linear* algorithm and less than *Binary* algorithm to find the optimal memory spot.

Additionally, from the Figure 2b showcasing the average execution duration of each algorithm, one can observe that the *Binary* algorithm has the highest execution duration for all function types. This also corresponds to the fact that this algorithm requires the highest number of iterations to find an optimal memory configuration. *Linear* algorithm performed better than the *Gradient Descent* in terms of the required iterations. However, *Gradient Descent* algorithm outperformed the *Linear* one in terms of the execution duration. For all the functions, *Gradient Descent* has the shortest execution duration.

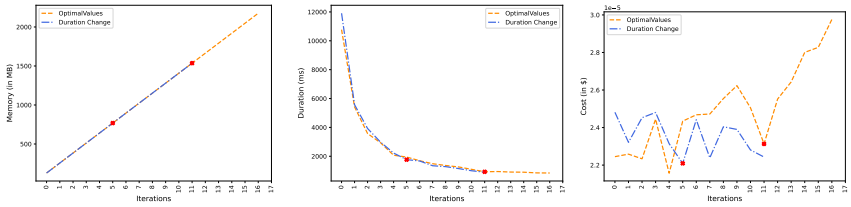
Binary algorithm shows the worst results; however, it can be explained by the fact that optimal memory configuration was located closer to the beginning of the memory interval. Thus, as the interval for the algorithm execution was wide (`128MB-10240MB`), it took many iterations for the algorithm to find the optimal memory configuration.

Balanced Optimization: As it can be seen from Figure 3a, in general *Duration Change* algorithm requires fewer iterations to find the optimal memory configuration compared to the *Optimization Value* algorithm. The *Duration Change* algorithm uses the definition proposed by AWS, which says that the balance between cost and duration is achieved at the memory, at which the duration curve of the function bends [14]. In the *Optimization Value* algorithm, it is assumed that the balance point of the function is such at which minimal duration can be achieved for the smallest cost, following Equa-



(a) Number of iterations (b) Execution duration

Fig. 3: The number of iterations and execution duration for various algorithms for the balanced optimization objective.



(a) Allocated memory (b) Execution duration (c) Execution cost

Fig. 4: Changes in different metrics with the iterations of the three algorithms for balanced optimization objective concerning the CPU benchmark.

tion 1. This algorithm usually selects the higher memory values (on the right side of the duration curve's knee). From Figure 3b, showing the average execution duration of both algorithms, one can observe that, for all function types, the *Optimization Value* algorithm required more time than the *Duration Change* algorithm, which is proportional to the number of iterations required by them.

Furthermore, Figure 4 shows how different parameters (cost, execution duration, and cost) behave for the CPU benchmark function when executed for the two algorithms with the balanced optimization objective. The resulting optimal configuration for each case is highlighted in all the three sub-figures.

5.2 Q2. Optimal configuration finding accuracy

As the duration and cost profiles of every test function are known, optimal memory intervals for each of them are calculated manually (displayed under optimal in Table 1). Thus, if the algorithm managed to find the memory in the optimal interval, its result is assumed to be correct. However, in the case of balanced optimization objective, estimating the correctness of the algorithms is more challenging as there is no clear definition of the term "optimal memory spot". Therefore, we only show accuracy measures for cost and duration optimization objectives in the below paragraphs.

Table 1: Memory configurations selected for the cost optimization objective.

Function Type	Optimal (in MB)	Linear (in MB)	Binary (in MB)	GD (in MB)
CPU-Intensive	<1280	845	1047	900
I/O-Intensive	< 1280	794	2071	767
Memory-Intensive	< 1152	947	1723	973
Network-Intensive	< 256	154	130	370

Cost Optimization: To evaluate the correctness of algorithms, the average result of their five executions for every benchmark function was calculated and compared to the correct memory intervals. Due to the variability of a cloud environment and lack of user control over it, it is hard to predict the exact memory configurations with which function will be executed with the lowest cost. Thus, based on the data received from the function's profiles (§4.1), optimal memory interval is defined as an interval in which the optimal memory spot can be located. The second column in Table 1 specifies the optimal memory interval for every function type. The next columns show average memory levels selected by each algorithm. If the selected memory level by the algorithm is inside the correct interval, we consider its result to be correct, and the corresponding table cell is marked green. Otherwise, the result is wrong and marked red.

For all function types, the *Linear* algorithm output results in the correct memory interval. *Binary* and *Gradient Descent* algorithms managed to find optimal memory configurations for two and three functions, respectively. To better evaluate the accuracy of the algorithms for the cost optimization objective, we conducted an experiment where each algorithm was executed five times for each of the four example functions, so there are twenty executions in total. The experiment concluded that, *Linear* algorithm has the highest accuracy - 95%, *Gradient Descent* - 85%, and *Binary* - 55%.

5.3 Q3. Active vs passive approach

As part of this evaluation, we only discuss the results of the balanced optimization goal deployed with the *Duration Change* algorithm. Figure 5 shows the scheme of execution *MAFF* in the *passive* approach used as part of this work for evaluation. Four test functions (CPU-, I/O-, memory- and network-intensive) were deployed in one CloudFormation stack and invoked every 5 minutes by a scheduled CloudWatch Event (Event A). This event was used to simulate user invocations. After each execution, corresponding log data was generated and stored in the CloudWatch service. In parallel to that, the analysis process for finding optimal memory configuration was also executed. The process was triggered by another CloudWatch scheduled event (Event B) with 30 minutes intervals. Thus, there were six function executions between every analysis round. Event B was configured to invoke the Function Collector Lambda function, which gathered ARNs of stack functions and added them into the SQS queue. Every new item in the queue was processed by Analyzer Lambda, which evaluated execution logs of the corresponding function. If Analyzer could identify the optimal memory for the function, it added a record into the DynamoDB database to avoid unnecessary analysis in future iterations. The whole experiment lasted for 6 hours, during which every of the

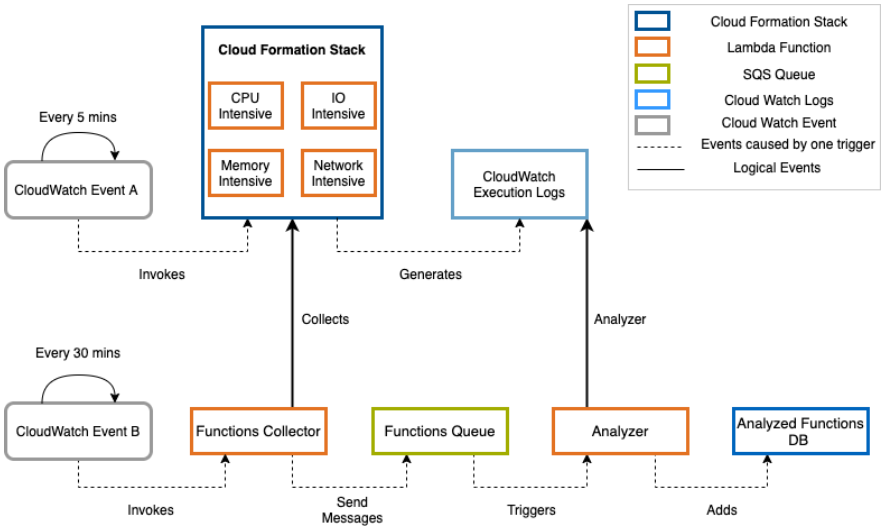


Fig. 5: Scheme of experiment on MAFF in passive approach.

test functions was executed 72 times and the analyzer function 12 times. It was enough to find optimal memories for all functions in the stack. As expected (Table 2), memory values selected by both approaches are quite similar, with some minor differences due to fluctuations in the value of the function's execution duration.

6 Comparison to Analogs

In this section, *MAFF* was compared to two popular resource optimization tools: AWS Compute Optimizer (ACO) [9] and AWS Lambda Power Tuning (ALPT) [12]. All experiments were performed on CPU-intensive function and the optimization objective was set to *cost* for *MAFF* and ALPT. *MAFF* was configured to use *Linear* algorithm for *active* and *passive* approaches. Optimization goal cannot be selected for ACO.

Table 2 shows a comparison between *MAFF* in *active* and *passive* approaches to its two analogs. Optimal memory suggested by the tools is quite different, but for both MAFF approaches and ALPT, the resulting value lies in the correct memory interval defined in §4.1. Memory suggested by ACO is below the defined interval (initial memory was set to `128MB`). ACO has the most strict requirements for its execution than all other tools. There must be at least 50 function invocations in the last 14 days, and memory allocated to the function must not be higher than `1792MB` [9]. For *MAFF* in the *passive* approach, the algorithm should have enough log values to perform analysis, and the number can vary depending on the function.

Execution duration for *MAFF* (*active* approach) and ALPT are similar - around 3 minutes per analysis. ACO can take up to 12 hours to find an optimal memory value. *MAFF* (*passive* approach) requires only 12 seconds for execution, on the condition that enough log values are provided. ALPT uses exhaustive search to identify optimal mem-

Table 2: Comparison of *MAFF* to its analogs

	M-Active	**M-Passive**	**ACO**	**ALPT**
Suggested Memory	845	896	160	1024
Requirements	None	approx. 20 function's invocations	minimum 50 invocations, less than `1792MB` allocated	None
Duration of Analysis	3 min 16 sec	11 sec	up to 12 hours	2 min 30 sec
Cost	0.0025	0.0012	0	0.0131
Automatic Value Setup	Yes	Yes	No	Yes

ory level for a cost, or execution duration. By default, this tool will need to perform at least 225 requests to the function to identify the optimal memory point. AWS Compute Optimizer is provided free of charge, while other optimization tools incur additional costs. The cost per analysis provided in this table can vary depending on the analyzed function and amount of steps the algorithm needs to perform, but in general, *MAFF* in both approaches is cost-efficient than the others.

While performing experiments on AWS Compute Optimizer, an interesting behavior of the tool was observed. To demonstrate it, a CPU-intensive function was deployed on four separate Lambda instances in eu-central-1 AWS region. Each of the functions was allocated different memories: `128MB`, `256MB`, `512MB`, and `1024MB` invoked every 5 minutes by the scheduled CloudWatch Event [8]. It was expected that the tool would suggest one optimal memory for the CPU-intensive function regardless of the initial memory level with which the function was created, as the application logic and workload for all functions is the same. However, after 12 hours of the experiment, AWS Compute Optimizer suggested different memories for each of the functions. Figure 6 shows the memory values proposed by Compute Optimizer for each function. For all of them, the tool recommended increasing memory value. Thus, the tool does not suggest the optimal memory configurations but relies on the initial memory allocated and increases them always.

7 Related Work

With the advent of serverless computing, there is a significant amount of research aimed at optimizing cloud computing resource utilization [4,3,12,21]. There has been some work on the performance profiling of various FaaS platforms. Wang et al. [28] performed an in-depth study of resource management and performance isolation with three popular serverless computing providers: AWS Lambda, Azure Functions, and Google Cloud Functions. Their analysis demonstrates a reasonable difference in performance between the FaaS platforms. Furthermore, Shahrad et al. [25] studied the architectural implications of serverless computing and pointed out that the short function runtimes hamper exploitation of system architectural features like temporal locality and reuse in FaaS. Chadha et al. [14] examine the underlying processor architectures for Google

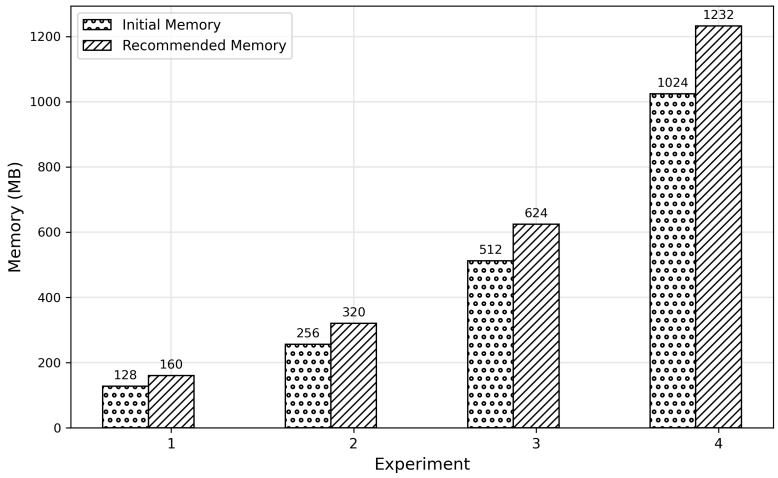


Fig. 6: Experiment on AWS Compute Optimizer showcasing wrong optimal memory suggested for the same function allocated with different initial memory configurations.

Cloud Functions (GCF) and determine the optimization of FaaS functions using Numba can improve performance by and save costs on average.

Furthermore, a significant number of research works aim to optimize the memory and cost for the FaaS functions. COSE [3] framework finds the optimal configurations for a FaaS function using the Bayesian Optimization algorithm while minimizing the total cost of execution. It models the behavior of a function and the environment (cloud, edge) in which those functions are deployed. However, they optimized based on cost only, does not guarantee the accuracy of the process, and can only be used in active mode. Bayesian Optimization was also used in CherryPick [5] tool for creating performance models for different cloud applications. The system provides 45-90% accuracy in finding optimal configurations and decreases cost up to 25%. But, they focused on traditional cloud applications. Another framework, Astra [19], is designed to optimize FaaS function configurations for specifically map-reduce usecase.

Google has developed a recommendation system to help the users choose the optimal virtual machine (VM) type [1]. It currently does not support Google Cloud Functions. As discussed in §6, AWS Compute Optimizer [2] can only be executed for the functions whose allocated memory level is less or equal to 1792MB and invoked at least 50 times in the last two weeks. AWS Lambda Power Tuning [12] tool uses exhaustive search to identify optimal memory level for a cost or execution duration. AWS Lambda Power Tuning is quite similar to MAFF in terms of lack of requirements, quick analysis time, and the possibility to set up recommended memory automatically. But users can use the AWS Lambda Power Tuning tool only in active mode, which can be impossible or not recommended for some business scenarios. Thus, the MAFF tool developed in this work outperforms AWS Compute Optimizer in the time required for the analysis, and provides a possibility to execute the tool in the *passive* approach.

None of the aforementioned research efforts address the issue of automatically configuring the optimal memory of FaaS functions from different objectives. The proposed tool *MAFF* fills that gap.

8 Conclusion

Serverless computing has abstracted most cloud server management decisions away from the users but configuring the memory of FaaS functions: a low-level configuration, which directly influences the performance and cost of the FaaS functions, is still left up to the users. To solve this problem, we introduced **MAFF**[1] to find optimal memory configuration for the FaaS function based on two optimization objectives: *cost*, and *balanced* (§2). For cost objective, it was possible to achieve 90% of accuracy using the *Linear* algorithm with at least two times smaller number of steps as compared to others. For achieving the *balanced* optimization goal, *Optimization Value* and *Duration Change* algorithms were used. We further introduced two different approaches for performing memory optimization - *active* and *passive*, differs based on the method of collecting the functions execution logs (§3). We also showcase *MAFF* advantages over the others in terms of cost and finding the optimal memory configurations (§6).

In the future, we plan to extend *MAFF* with other public serverless compute providers. Adding the functionality of tracking updates in the program code of the analyzed function is another prospect.

ACKNOWLEDGEMENTS

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. The authors also thank the anonymous reviewers whose comments helped in improving this paper.

References

1. Google cloud recommendations (2018), `https://cloud.google.com/compute/docs/instances/apply-machine-type-recommendations-for-instances`, (Accessed on 06/17/2021)
2. Aws compute optimizer (2021), `https://aws.amazon.com/compute-optimizer/`, (Accessed on 06/17/2021)
3. Akhtar, N., Raza, A., Ishakian, V., Matta, I.: Cose: Configuring serverless functions using statistical learning. In: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications. pp. 129–138 (2020). https://doi.org/10.1109/INFOCOM41043.2020.9155363
4. Akin, M.: How does proportional CPU allocation work with AWS Lambda? — Opsgenie Engineering, `https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac`
5. Alipourfard, O., Liu, H.H., Chen, J., Venkataraman, S., Yu, M., Zhang, M.: Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In: Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation. p. 469–482. NSDI'17, USENIX Association, USA (2017)

[1] https://github.com/tetzubko/self-adaptive-memory-faas

6. Amazon Web Services: AWS Lambda – Serverless Compute - Amazon Web Services, https://aws.amazon.com/lambda/
7. AWS: Choosing the Optimal Memory Size - Serverless Architectures with AWS Lambda, https://docs.aws.amazon.com/whitepapers/latest/serverless-architectures-lambda/choosing-the-optimal-memory-size.html
8. AWS: Creating a CloudWatch Events Rule That Triggers on a Schedule - Amazon CloudWatch Events, https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/Create-CloudWatch-Events-Scheduled-Rule.html
9. AWS: Supported resources and requirements - AWS Compute Optimizer, https://docs.aws.amazon.com/compute-optimizer/latest/ug/requirements.html#requirements-lambda-functions
10. AWS: Aws lambda pricing (2020), https://aws.amazon.com/lambda/pricing/
11. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P.: Serverless computing: Current trends and open problems. In: Research Advances in Cloud Computing, pp. 1–20. Springer Singapore (2017)
12. Casalboni, A.: AWS Lambda Power Tuning, https://github.com/alexcasalboni/aws-lambda-power-tuning
13. Chadha, M., Jindal, A., Gerndt, M.: Towards federated learning using faas fabric. In: Proceedings of the 2020 Sixth International Workshop on Serverless Computing. p. 49–54. WoSC'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3429880.3430100
14. Chadha, M., Jindal, A., Gerndt, M.: Architecture-specific performance optimization of compute-intensive faas functions. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). pp. 478–483 (2021). https://doi.org/10.1109/CLOUD53861.2021.00062
15. Eismann, S., Bui, L., Grohmann, J., Abad, C., Herbst, N., Kounev, S.: Sizeless: Predicting the Optimal Size of Serverless Functions, p. 248–259. Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3464298.3493398
16. Eivy, A.: Be wary of the economics of "serverless" cloud computing. IEEE Cloud Comput. **4**(2), 6–12 (2017). https://doi.org/10.1109/MCC.2017.32
17. Fan., C., Jindal., A., Gerndt., M.: Microservices vs serverless: A performance comparison on a cloud-native web application. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,. pp. 204–215. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009792702040215
18. Grafberger, A., Chadha, M., Jindal, A., Gu, J., Gerndt, M.: Fedless: Secure and scalable federated learning using serverless computing. In: 2021 IEEE International Conference on Big Data (Big Data). pp. 164–173 (Dec 2021). https://doi.org/10.1109/BigData52589.2021.9672067
19. Jarachanthan, J., Chen, L., Xu, F., Li, B.: Astra: Autonomous serverless analytics with cost-efficiency and qos-awareness. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 756–765 (2021). https://doi.org/10.1109/IPDPS49936.2021.00085
20. Jindal, A., Chadha, M., Benedict, S., Gerndt, M.: Estimating the capacities of function-as-a-service functions. In: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion. UCC '21 Companion, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3492323.3495628
21. Jindal, A., Frielinghaus, J., Chadha, M., Gerndt, M.: Courier: Delivering serverless functions within heterogeneous faas deployments. In: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing. UCC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468737.3494097, https://doi.org/10.1145/3468737.3494097

22. Jindal, A., Gerndt, M.: From devops to noops: Is it worth it? In: Ferguson, D., Pahl, C., Helfert, M. (eds.) Cloud Computing and Services Science. pp. 178–202. Springer International Publishing, Cham (2021)
23. Linux: dd(1) - Linux manual page, https://man7.org/linux/man-pages/man1/dd.1.html
24. Ruder, S.: An overview of gradient descent optimization algorithms. Tech. rep. (2017), http://caffe.berkeleyvision.org/tutorial/solver.html
25. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 1063–1075 (2019)
26. Shankar, V., Krauth, K., Vodrahalli, K., Pu, Q., Recht, B., Stoica, I., Ragan-Kelley, J., Jonas, E., Venkataraman, S.: Serverless linear algebra. In: Proceedings of the 11th ACM Symposium on Cloud Computing. p. 281–295. SoCC '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3419111.3421287
27. Steinbach, M., Jindal, A., Chadha, M., Gerndt, M., Benedict, S.: Tppfaas: Modeling serverless functions invocations via temporal point processes. IEEE Access **10**, 9059–9084 (2022). https://doi.org/10.1109/ACCESS.2022.3144078
28. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: 2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18). pp. 133–146. USENIX Association (2018)
29. WG, C.S.: Cncf wg-serverless whitepaper v1. 0. https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf (March 2018), [Online; Accessed: 15-July-2020]
30. Zhang, M., Zhu, Y., Zhang, C., Liu, J.: Video Processing with Serverless Computing: A Measurement Study (2019). https://doi.org/10.1145/3304112.3325608