SLAM: SLO-Aware Memory Optimization for Serverless Applications

*Note: This is the preprint version of the accepted paper at IEEE CLOUD'22

Gor Safaryan, Anshul Jindal, Mohak Chadha, Michael Gerndt
Chair of Computer Architecture and Parallel Systems, Technische Universität München, Germany
Garching (near Munich), Germany
Email: {gor.safaryan, anshul.jindal, mohak.chadha}@tum.de, gerndt@in.tum.de

***Abstract*—Serverless computing paradigm has become more ingrained into the industry, as it offers a cheap alternative for application development and deployment. This new paradigm has also created new kinds of problems for the developer, who needs to tune memory configurations for balancing cost and performance. Many researchers have addressed the issue of minimizing cost and meeting Service Level Objective (SLO) requirements for a single FaaS function, but there has been a gap for solving the same problem for an application consisting of many FaaS functions, creating complex application workflows.**

In this work, we designed a tool called SLAM to address the issue. SLAM uses distributed tracing to detect the relationship among the FaaS functions within a serverless application. By modeling each of them, it estimates the execution time for the application at different memory configurations. Using these estimations, SLAM determines the optimal memory configuration for the given serverless application based on the specified SLO requirements and user-specified objectives (minimum cost or minimum execution time). We demonstrate the functionality of SLAM on AWS Lambda by testing on four applications. Our results show that the suggested memory configurations guarantee that more than 95% of requests are completed within the predefined SLOs.

***Index Terms*—serverless, cost optimization, memory optimization, SLO**

I. Introduction

Significant progress has been made in different domains [1]–[5] based on the idea of *serverless computing* since its launch by Amazon as AWS Lambda in November 2014 [6]. Serverless computing is a cloud computing model that abstracts server management and infrastructure decisions away from the users [7]. In this model, the allocation of resources is managed by the cloud service provider rather than by *DevOps*, thereby benefiting them from various aspects such as no infrastructure management, automatic scalability, and faster deployments [8], [9]. Function-as-a-Service (FaaS) is a key enabler of serverless computing [7]. In FaaS, a serverless application is decomposed into simple, standalone functions that are uploaded to a FaaS platform such as AWS Lambda [10], Google Cloud Function (GCF) [11], and Azure Functions (AF) [12] for execution. Most of the public cloud providers in their FaaS offerings allow users to configure memory allocation for the functions [11], [13], [14].

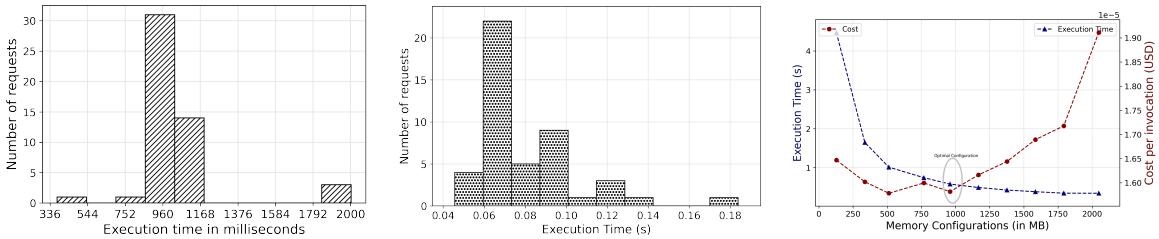
Despite having many advantages, serverless computing suffers from some pain points that obstruct its wide adoption [15]–[17]. The most commonly known is optimally configuring the memory of the FaaS functions within the application based on the required the Service Level Objective (SLO). The difficulties lie in the following aspects:

Cold start: It is mainly connected with loading the FaaS function into the main memory of the executing server and preparing the execution environment for the target code (starting up the VM/container, loading libraries, loading function code, etc.) [18], [19]. The cold start phenomenon combined with the heterogeneity of the cloud environment makes the function execution time quite unpredictable. Figure 1a shows an execution time distribution for a sample compute-intensive function having a high variance when deployed with 128MB memory configuration on AWS Lambda.

FaaS functions integration with BaaS services: The FaaS functions are usually closely integrated with other services, e.g., cloud databases, authentication and authorization services, and messaging services. These services are called Backend-as-a-Service (BaaS) [20]. These services also do influence the execution time of the FaaS functions, thus adding the variance in the time. Figure 1b shows an execution time distribution for a sample function querying DynamoDB having a high variance when deployed with 128MB memory configuration on AWS Lambda.

Trade-off analysis between performance and cost: Users need to define memory configuration for their FaaS functions: a low-level information which directly influences the performance and cost of the serverless application [10], [21], [22]. Thus, the user has to do a trade-off analysis between them to define the right configuration for their required SLOs [23]. Figure 1c shows an execution time vs the cost graph for a sample compute-intensive function when deployed with different memory configurations on AWS Lambda. We can observe that it's not trivial to find the optimal configuration where the overall cost and execution time are both optimal.

Complex application workflows: Usually, the serverless applications comprise dozens if not hundreds of small FaaS functions and these connect together to form complex event-driven workflows. Furthermore, the SLOs are usually defined at the application level instead of the function level and thus based on the required application SLOs configuring the memory of the FaaS functions within the application even



(a) Execution time variance due to cold start problem.

(b) Execution time variance due to BaaS service (DynamoDB).

(c) Performance vs cost trade for finding the optimal configuration.

Fig. 1: Various factors making it difficult to optimally configure the memory of the FaaS functions within a serverless application.

becomes more challenging since a change in one can influence the others.

The aspects above highlight some factors that make it difficult for the users to optimally configure memory for serverless applications based on the required SLOs. However, there are many other factors such as I/O and network bandwidth, and co-location with other functions affecting the performance and cost which the users are not aware of [22]. Many researchers have addressed the issue of optimizing the memory and cost for meeting SLO requirements for a single cloud function [24]–[26], but there has been a gap for solving the same problem for a serverless application consisting of many FaaS functions, which create a complicated workflow of function calls. To this end, we develop **SLAM**, a python-based tool that can automatically find the optimal memory configurations for the FaaS functions within the given serverless application based on the specified SLO. Our key contributions are as follows:

- We develop and present a novel tool called **SLAM** that automatically determines the optimal memory configuration for the FaaS functions within the given serverless application based on the specified SLO requirements (§II). To the best of our knowledge, this is the first work that find and configure FaaS functions with optimal memory configurations within a serverless application based on the specified SLO.
- We propose and implement an optimization algorithm along with its variants for various optimization objectives (minimum cost and minimum overall time) in addition to the SLO requirements in finding the optimal memory configuration for the given serverless application (§II-E2).
- Although our approach is generic and *SLAM* can be easily extended to support other commercial and open-source FaaS platforms, we demonstrate the functionality of *SLAM* with AWS Lambda (§IV) on four serverless applications comprising of various number of functions.
- We evaluate the performance of the *SLAM* on 3 different aspects: 1) Estimation time accuracy (§IV-A), 2) Configuration finding accuracy (§IV-B), and 3) Configuration finding efficiency and scalability of *SLAM* (§IV-C). From the experimental evaluation, the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs.

TABLE I: Symbols and definitions used in this paper

Symbol	Interpretation
N	total number of functions in an application
M	total number of memory configurations
S	total number of sequence groups formed from application call graph
U_i	total number of sub-sequence groups within some group i
K	total number of user-requests for load generation
X	possible number of memory configurations adhering to the defined SLOs.
m_i^j	memory allocated to i^{th} function in the j^{th} configuration set
mem_config_list	a list of memory values [128, 256, 512, 1024, 2048, 4096, 8192, 10240]
$F = \{f_1, \ldots, f_N\}$	functions within an application
$G = \{g_1, \ldots, g_S\}$	sequence groups from application's call graph
$\bar{G} = \{\bar{g}_1^i, \ldots, \bar{g}_U^i\}$	sub-sequence groups within some group i
$C = \{C_1, \ldots, C_X\}$	memory configs adhering to the defined SLOs.
α	n^{th} percentile (called choice percentile) of the distribution as a representative for the execution time for the given function at a particular memory configuration.

II. System Description

In this section, we present *SLAM*, a python-based tool for automatically configuring the FaaS functions within a serverless application with optimal memory such that the overall execution time of invocations to the application conform to the defined Service-Level Objective (SLO) requirements. In this work, we consider the 95^{th} percentile execution time of an application invocation as the SLO. *SLAM* also supports additional user-specified objectives on top of the SLO requirements: 1) Minimum Overall Cost (MOC), and 2) Minimum Overall Execution Time (MOET), by which the *SLAM* suggested configuration for the serverless application not only conforms to the defined SLO requirements, but also meets user-specified objectives. *SLAM* can dynamically adapt to changes in the given serverless application and automatically adjust memory configurations of functions. *SLAM* can be incorporated into a Cloud Service Provider (CSP) Function-as-a-Service (FaaS) platform and then leveraged by application developers for optimizing the memory configuration of their serverless applications.

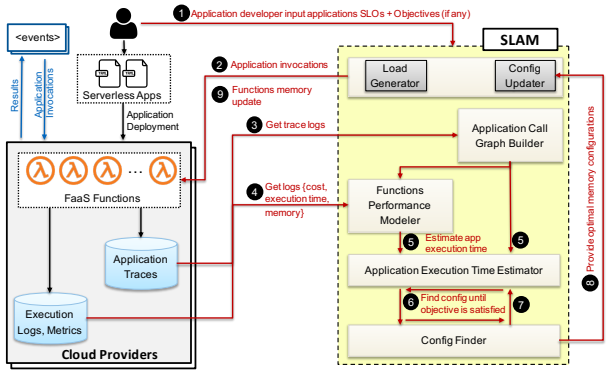


Fig. 2: High-level architecture of the *SLAM* and the interaction between its components in a general use case.

Figure 2 provides an overview of our developed *SLAM* tool and the interaction between its components in a typical usecase. *SLAM* assumes that the serverless application which is to be configured is already deployed by the application developer on a FaaS platform (AWS Lambda [10] in our case) and is instrumented with a middleware tracing library (such as AWS X-Ray [27]) to trace the incoming and outgoing requests to other functions, or other cloud components/services.

SLAM takes the SLOs requirement for the application as the input along with other user-specified objectives (if any) (step ❶). Then the *Load Generator* component of it generates a minimal amount of user workload ($K = 50$ application invocations, see Table I) to the application's public endpoint (step ❷) and collects application trace logs (step ❸) and various monitoring metrics[1] data (step ❹). The collected logs are used by *Application Call Graph Builder* component to construct the application call graph (step ❸) and this call graph along with the monitoring metrics are further used by *Functions Performance Modeler* component for building the application's functions performance models. *Application Execution Time Estimator* component use models along with the application call graph for estimating the overall application response time on the different memory configurations provided by *Config Finder* component. *Config Finder* component generates the configuration based on the developed algorithms (§II-E2) and examine the estimated time, memory configurations, and cost for the SLOs requirements and user-specified objective (if any) satisfaction (step ❻). If the SLOs requirements and user-specified objective are not satisfied, *Config Finder* tries different memory configurations (step ❼) and continues the process until it is satisfied (steps ❻ - ❼). Once a configuration is found, the functions' memory configurations are updated by *Config Updater* component (steps ❽ - ❾). Next, we describe the six major components of *SLAM* tool in more detail.

A. Load Generator

This component is responsible for generating user workload to the deployed application. It takes a total number of requests to the application as input and based on it generates the given amount of user workload requests synchronously to the deployed application. This user workload generation allows creating application traces and collect various metrics data used by the other components of the *SLAM*.

B. Application Call Graph Builder

This component is responsible for building the application call graph involving the application functions and Backend-as-a-Service (BaaS) services such as database, storage, and queues. *SLAM* relies on external middleware tracing libraries (such as AWS X-Ray) instrumented by the application developer allowing to trace the incoming and outgoing requests to other functions, or BaaS services.

This component with the help of *Load Generator* component generates a small amount of user workload requests (§III) to the deployed application. The application traces are then parsed to generate the application call graph involving all the functions and BaaS services within the application. Afterward, the component filters out BaaS services, as it is out of the scope of this work to tune them. As a result, after this step, we get the simplified call graph for the deployed serverless application along with the composing functions. In case the user already has the application call graph and wants to skip this step, *SLAM* allows the user to input manually the call graph of the application. This also increases the testability of the *SLAM* for further development.

C. Functions Performance Modeler

After building the call graph of the application and knowing its composing functions, the next step is to estimate the execution time of each function within the serverless application at different memory configurations. This is done in two steps, explained next.

1) Create traces and metrics data for building models: This component with the help of *Load Generator* component first generates a small amount of user workload requests ($K = 50$ application invocations, see Table I) to the deployed application when all of its composing functions are deployed with a default same memory configuration (128MB). Based on the composing functions found by the *Application Call Graph Builder* component, it then requests *Config Updater* component for updating the memory configurations of those functions based on the default list of memory configuration values (mem_config_list in Table I) and *Load Generator* to again generate the same amount of user workload requests to the updated application. The process is repeated for all the memory configurations (mem_config_list in Table I) and in the end application traces and various metrics data are created for estimating execution time for each function within the application.

2) Estimation of execution time for each function : Traces are parsed and metrics are analyzed to create a distribution of execution time for each function and each memory configuration. An example of such a distribution for a test function,

[1]https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html

when deployed with 128MB memory configuration on AWS Lambda, is shown in Figure 1a.

One can observe that there is a high variance in the execution time of the function running under the same configuration due to the uncertainties from the underneath virtualized cloud infrastructure such as co-location of functions, cold-start, hardware failures, resource-overuse, etc. Therefore, to overcome this inherent variance, we choose a hyperparameter called *choice percentile* (α in Table I) representing the n^{th} percentile of the distribution as a representative for the execution time for the given function at a particular memory configuration. α is configured automatically by SLAM. Calculating prediction accuracy of execution time at multiple values of α (default test values: $50, 75, 90, 99$), SLAM selects the one which results in a minimum mean squared error.

Thus, in the end, a list of representative values for execution time for each function and memory combination is created, and how they are combined to form the overall execution time of the application is presented next.

D. Application Execution Time Estimator

Given the execution time of each FaaS function comprising the serverless application estimated by the *Functions Performance Modeler* at certain memory configurations, it is the responsibility of this component to combine them to estimate the overall application execution time.

Function invocations in the application can either be in parallel or one after the other in a sequence, or in a combination of both. Therefore, from the application call graph first, it determines which functions are executed in parallel to others by using the functions' start and end timestamps available from the traces. The tool then divides all functions into groups of sequence groups (S in Table I), where all the functions in each group are executed in parallel to other functions in the same group, and each group is executed in sequence to other groups. Since all the functions in a group are invoked in parallel, therefore to estimate the execution time of a group we take the maximum of the execution times of all functions in the group. In the end, we sum the execution times of each group to get an estimate of overall application execution time.

Mathematically, if we have an application consisting of N functions configured with certain memory configurations and defined as $F = \{f_1, f_2, f_3, ..., f_N\}$, with them being divided into S sequence groups defined as $G = \{g_1, g_2, g_3, ..., g_S\}$, then the execution time of the whole application is given by:

$$T(G) = \sum_{x=1}^{N} F(g_x) \tag{1}$$

where for some group i:

$$F(g_i) = \begin{cases} max(T(\bar{g}_1^i), \ldots, T(\bar{g}_U^i)), & \text{if } g_i \neq \text{function.} \\ \text{function execution time,} & \text{if } g_i = \text{function.} \end{cases} \tag{2}$$

where $\bar{g}_j^i$ ($1 \leq i \leq S$ and $1 \leq j \leq U$) being the sub-sequence group within g_i and U is the total number of sub-sequence groups within g_i.

E. Config Finder

Given the estimated execution time for each FaaS function comprising the serverless application provided by the *Functions Performance Modeler*, it is the responsibility of this component of *SLAM* tool, *Config Finder*, to find the right memory configurations for all functions such that the overall application execution time adheres to the defined SLOs and the specified optimization objectives (if any). We first present the two optimization objectives (§II-E1) that can be used as part of *SLAM* tool in addition to the SLO requirements, and then we introduce the algorithm for finding the optimal memory configurations (§II-E2).

1) Optimization Objectives: Suppose there are total of X possible memory configurations set for the serverless application defined as $C = \{C_1, C_2, ..., C_X\}$ such that $C_j = \{m_1^j, m_2^j, ..., m_N^j\}$ ($1 \leq j \leq K$) is a memory configuration set for F adhering to the defined SLOs and $m_i^j \in M$ ($1 \leq i \leq N$) is the memory allocated to i^{th} function in the j^{th} configuration set. Following are the two optimization objectives that can be used as part of *SLAM* tool along with the defined SLOs:

Minimum Overall Cost (MOC): Here, the idea is to find a configuration that would result in a minimum cost for each invocation of the application under the given SLO requirements. This is given by:

$$\min_{j \in C} Cost(j) \tag{3}$$

where $Cost(j)$ ($j \in C$) is the overall application estimated cost when the application is configured with C_j configuration. Our calculation only counts for the costs associated with the function execution and does not take into account the data transfer, storage, and other costs associated with the invocation of functions. To calculate the aforementioned execution cost, we used the data provided by AWS [28]. Though they provide pricing only for a limited number of memory configurations, we interpolated the cost as there was a linear relationship between allocated memory and cost.

Minimum Overall Execution Time (MOET): The objective is to find a configuration that would result in minimum overall execution time of the application under the given SLO requirements. This is then given by:

$$\min_{j \in C} ExecTime(j) \tag{4}$$

where $ExecTime(j)$ ($j \in C$) is the overall application estimated time by *Application Execution Time Estimator* when the application is configured with C_j configuration.

2) Optimal memory configuration finding algorithm: Now we describe the algorithm (called *SLAM-SLO*) for finding the optimal memory configuration for serverless applications such that the overall application execution time adheres to the defined SLOs. The modified version of the algorithm for optimizing on various objectives along with the SLOs is called *SLAM-SLO-Min-Cost* for MOC and *SLAM-SLO-Min-Time* for MOET. We additionally compared our developed algorithm with brute force (referred as *Brute-Force*) approach where

all possible combinations for configurations for the functions within the application are generated to find the configuration that conforms to defined SLOs and the given objective. The overall complexity of this brute force approach is $O(M^N)$.

SLAM-SLO: In this approach, we leverage the max-heap data structure for finding the optimal configuration which satisfies the SLO requirements. The pseudocode for the algorithm is shown in Algorithm 1. Each function's execution time at the minimum memory configuration i.e., 128MB is calculated and is used for constructing the max-heap. We store the execution time of the function at a particular configuration as the node value and the function name and its memory configuration are further saved as the node's metadata (Line 5-8). The function at a particular memory configuration having the highest execution time will be automatically stored at the head of the max-heap tree (Line 9). We first check if this base configuration satisfies the SLO requirements. In case it does, we stop the iteration and return the configuration (Line 11-13). Otherwise, in the next step, we pop the head from the max-heap (Line 14), increase its memory to decrease its execution time (Line 16) and then push the function again back to the heap with the updated memory and execution time (Line 17-20). After this update, we check if the configuration satisfies the SLO requirements. In case it does, we stop the iteration and return the configuration (Line 11-13). Otherwise, we continue the process by popping the function at the head until a configuration is found. If no configuration is found, an empty dictionary is returned. The overall complexity of this approach is given by:

$$O(NM\log N) \tag{5}$$

This method is highly scalable and also does locally optimal steps to lower the overall execution time of function call.

SLAM-SLO-Min-Cost: We further modified the *SLAM-SLO* algorithm to take cost into account for finding the optimal configuration with the MOC as the objective along with the SLO requirements. Here, the algorithm uses the *SLAM-SLO* found optimal configuration as the default configuration and tries to optimize on top of it for finding minimum cost configuration. In this, every time we pop the function from the head of max-heap, we check for the following inequality at the new updated memory for that function:

$$\left|\frac{\text{new\_cost} - \text{old\_cost}}{\text{old\_cost}}\right| \leq \left|\frac{\text{old\_exec\_t} - \text{new\_exec\_t}}{\text{old\_exec\_t}}\right| \tag{6}$$

where new_cost and new_exec_t are the cost and execution time of an application invocation after updating the memory of the function, and old_cost and old_exec_t correspond to the cost and execution time before the update. In case the inequality holds, we put the function back into the max-heap with the updated execution time. In case it doesn't, we fix the memory for that function in the final configuration. This also allows us to reduce the search space for finding the configuration satisfying the minimum cost objective.

SLAM-SLO-Min-Time: This modified version of the *SLAM-SLO* algorithm also uses the *SLAM-SLO* found optimal configuration as the default configuration and tries to optimize on top of it for finding minimum execution time configuration. It then leverages the binary search algorithm to find the configuration with minimum time. It uses the *SLAM-SLO* found optimal configuration execution time (β in seconds) as the maximum time and $0s$ as the minimum time. It then updates the SLO requirement to the middle of maximum and minimum time and calls the *SLAM-SLO* algorithm to find an optimal configuration. If a configuration is found, then the maximum is set to the execution time for that configuration, otherwise the minimum is updated to the previously found middle. This way it continues until a configuration is found with minimum application execution time. In order not to run the binary search indefinitely, we use a hyperparameter called precision (γ). When the lower and upper execution time bounds get closer than the precision hyperparameter, we stop the search and return the attained configuration. As a default value of the parameter, we chose $\gamma = 0.01s$, which can be easily changed. The complexity of the algorithm is given by:

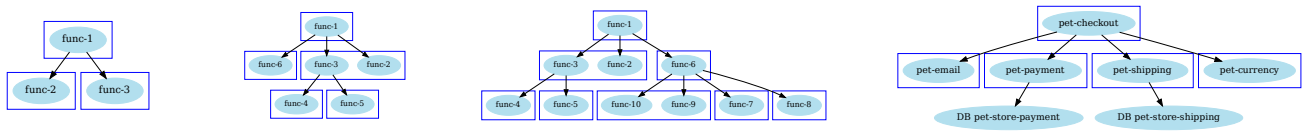
$$O\left(NM\log N \times \log\left(\frac{\beta}{\gamma}\right)\right) \tag{7}$$

Algorithm 1: SLAM-SLO Algorithm

```
   Input: func_list, mem_config_list: List[ ], SLO)
   Output: result_config = Dict[str, int]
1  min_mem_config = min(mem_config_list)
2  for func_name in func_list do
       // init minimum memory assignment for all functions
3      res_config[func_name] = min_mem_config;
4  end
   // prepare heap with function's exec time at min memory
5  for fname in func_list do
6      func_exec_time = exec_time(fname, min_mem_config);
7      func_heap.append(func_exec_time, fname);
8  end
9  heapify_max(func_heap); // reorder heap
10 do
       // check for objective(s) satisfaction.
11     if estimate_exec_time(res_config) ≤ SLO then
12         return res_config;
13     end
14     top_func = heappop_max(func_heap);
15     if not all_memory_config_evaluated(top_func) then
           // update memory and time, then append to heap
16         func_new_mem = update_memory(top_func);
17         func_new_exec_time = get_exec_time(top_func,
             func_new_mem);
18         func_heap.append(func_new_exec_time, top_func);
19         res_config[top_func] = func_new_mem; // update
20         heapify_max(func_heap); // reorder heap
21     end
22 while func_heap is not empty;
23 return ; // return the empty config
```

III. Evaluation Settings

We test the proposed *SLAM* tool for serverless applications deployed on AWS Lambda, a popular serverless cloud platform. *SLAM* tool itself was run on a machine with 8



(a) 3-functions test app (b) 6-functions test app (c) 10-functions test app (d) Real-world based app

Fig. 3: Call graphs for the applications used for evaluating *SLAM*.

physical cores (Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz CPU) with hyperthreading enabled and 16 GB of RAM. These conditions are similar to a typical cloud VM.

SLAM has a default list of memory configuration values (mem_config_list in Table I), that it chooses from when generating memory configurations for the application. As all the functions within our test applications use only one thread, we limit the maximum memory configuration to 2GB, since as at that point AWS stops increasing the portion of the allocated vCPU and increases the number of available vCPU [22], which will then not used by the application. For our experiments, total number of requests for load generation is set to as `50`.

A. Test Applications

1) Synthetic Applications: To test the *SLAM* tool, we have developed an interface that can create automatically synthetic applications having a different number of functions. The input to the interface defines the application call tree containing functions that are either invoked in parallel or sequence. This way we can generate complex applications with as many functions. Such a structure gives us the opportunity to test the limits of the *SLAM* tool and understand how much error is accumulated if the application contains many cloud functions with a combination of sequence and parallel invocations. Each function within the application is a compute-intensive function which calculates the sum of remainders for N when divided by all numbers between 2 and N, where N is the parameter fixed for the function. The simplicity of the algorithm allows us to simulate test applications with heterogeneous functions requiring different compute/memory resources by scaling N. Each function within the application has a different value for N and is assigned randomly. An example application with three functions where one function (func-1) is invoking the other two (func-2, func-3) in the sequence is created, and its call graph is shown in Figure 3a.

We additionally created two more synthetic complex applications containing 6 and 10 functions incorporating sequence and parallel invocations to test the *SLAM* tool. Their call graphs are shown in Figure 3b and Figure 3c respectively. Functions in the same box are called in parallel to each other, while the ones on the same level are called in sequence. The directed edges show the function which has generated the invocation for the other functions on the lower level.

Such complex application call graphs allow us to estimate how well *SLAM* adapts to changing execution time when a change in a leaf function's configuration affects the execution time of the other higher level functions.

2) Real-world based Application: Since the synthetic application workloads do not fully represent the real-world use cases for serverless applications, therefore we created a pet store application based on an open-source spring-based application[2] consisting of five FaaS functions and two NoSQL databases. Its call graph is shown in Figure 3d. We used DynamoDB for the two NoSQL databases. This application is special, since the functions querying databases will not have any influence on execution time with the increase in memory.

In this application, when the client selects a pet in the front-end for buying, it first automatically invokes the *pet-checkout* function, which in turn is responsible for getting all the details needed for the purchase by invoking other functions. First, it calls the *pet-currency* function to convert the pet price to USD. Then it calls the *pet-payment* service for the client to pay for the pet. If the payment is successful then the *pet-checkout* function will invoke *pet-shipping* which will log the pet shipping details in the database. After successful completion of all the previous steps, the final *pet-email* function is called, which generates a summary email and sends it to the client. The application is just a skeletal representation of what a real one would look like; it does not ship anything.

IV. Evaluation

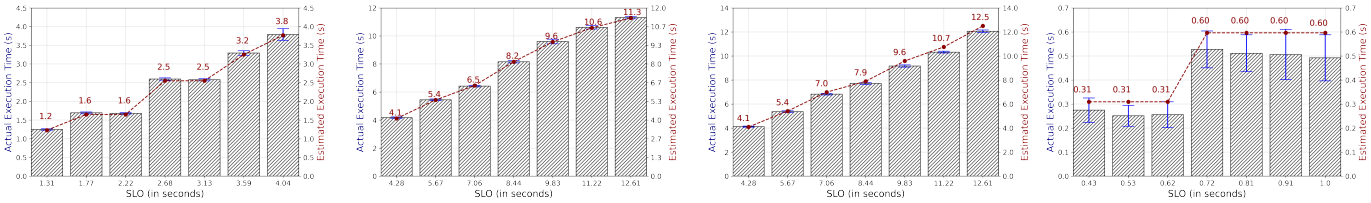
We design our experiments to answer the questions:

- **Q1. *SLAM* estimation time accuracy**: how accurate is *SLAM* in estimating the execution time of an application for the given or found configuration at different SLOs?
- **Q2. *SLAM* configuration finding accuracy**: how accurate is *SLAM* in finding the configuration satisfying the given SLOs and objectives for an application?
- **Q3. *SLAM* configuration finding efficiency and scalability**: how efficient is *SLAM* in finding the configuration satisfying the given SLOs and objectives for an application? Additionally, how does the *SLAM* tool scale with the increase in the number of functions of the application?

A. Q1. SLAM estimation time accuracy

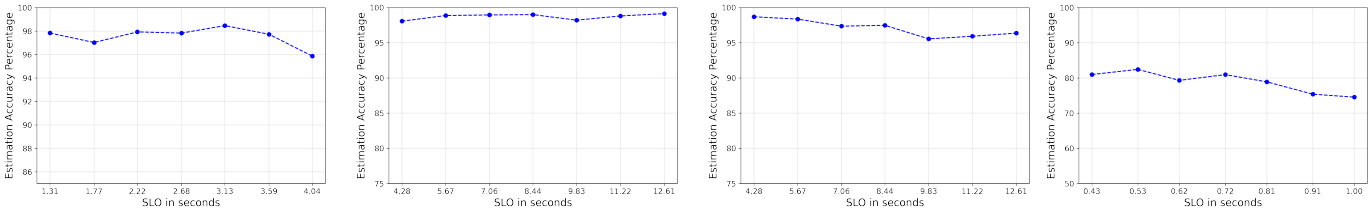
To demonstrate the effectiveness of the *SLAM* tool in estimating the execution time of the application, we test it on three synthetic and one real-world-based application. For this test, *SLAM* tool's *SLAM-SLO* algorithm is used to find the

[2]https://github.com/spring-projects/spring-petclinic



(a) 3-functions test application (b) 6-functions test application (c) 10-functions test application (d) Real-world based application

Fig. 4: Actual execution time box plot overlaid with the estimated execution time by *SLAM* run with different SLOs.



(a) 3-functions test application (b) 6-functions test application (c) 10-functions test application (d) Real-world based application

Fig. 5: Execution time estimation accuracy percentage for the four test applications at different SLOs.

memory configurations for the given different SLOs without any additional objectives. Based on the found configuration, we then configured all the functions with the memory values suggested by *SLAM-SLO* and invoke the serverless application 100 times to get the actual application's execution time distribution. Figure 4 shows the actual experiment execution time box plot overlaid with the estimated execution time by *SLAM-SLO* algorithm for all four test applications at different SLOs when configured with the found memory configurations.

Additionally, we measured the execution time estimation accuracy percentage for the four test applications at different SLOs and is shown in Figure 5. For computing the accuracy at different SLOs, we calculate the mean squared percentage error between the estimated and actual execution time for the found configuration and then subtract it from 100.

Next, we discuss the results of the two classes of the test applications in more detail.

1) Synthetic Applications: From the Figure 4, one can observe that in the three synthetic applications the estimated execution time is either lower or equal to that of the specified SLOs. Additionally, from the overlaid graph of estimated execution time in Figure 4, we can observe that the estimated execution time to a great extent is closer to the actual execution time at different SLOs. To verify it further, in Figure 5, the measured execution time estimation accuracy percentage for the three test applications at different SLOs is above 90%.

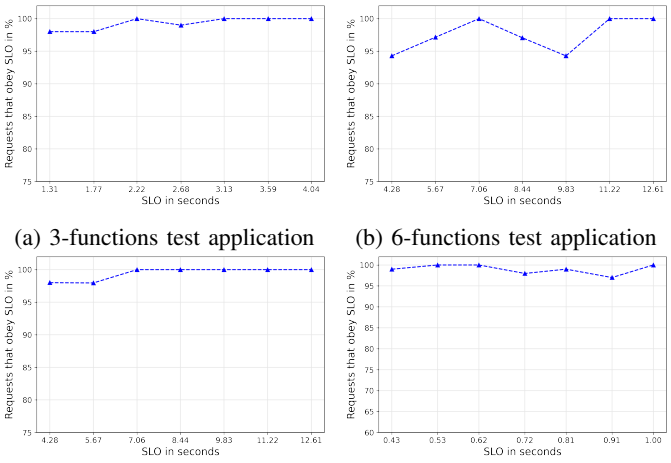
2) Real-world based Application: From the overlaid graph of estimated execution time in Figure 4d, one can observe that the estimated execution time is a bit higher than the actual execution time at different SLOs which is also evident from the Figure 5d where the measured execution time estimation accuracy percentage at different SLOs is lower as compared to synthetic applications (ranging between 70% and 85%), but similar to the three synthetic applications, the estimated execution time for this application is also either lower or equal to that of the specified SLOs. Thus, the configuration selected by the *SLAM* tool is good enough to fulfill the desired SLOs.

One reason for the higher estimated execution time at different SLOs could be due to the high variance in the actual execution time of the functions within the application (as seen in Figure 1b) because of the involvement of components such as DynamoDB which can lead to the variable execution time of the application. Moreover, the overall execution time of this application is smaller as compared to synthetic applications and thus even the small inherent variance within the application can cause high relative error rates and hence the drop in the estimation of the accuracy. Nonetheless as mentioned earlier, the configuration selected by the *SLAM* tool is good enough to fulfill the desired SLOs.

B. Q2. SLAM configuration finding accuracy

In this experiment, for determining the accuracy of *SLAM* in finding the configuration at the given SLOs, we have considered two aspects presented next.

1) Precision of requests conforming SLO requirements: Here, we calculate the percentage of requests conforming to the defined SLOs when the functions are configured with the memory configurations suggested by *SLAM-SLO* algorithm. Experiment results on the four test applications is shown in Figure 6 for different SLOs when a total number of 100 requests were issued to the application at each SLO. We can observe that for all the synthetic applications, the percentage of requests conforming to the given SLOs is either equal or above 95% which means that out of issued 100 requests at least 95 requests were served within the specified SLO execution time. Additionally, for the *Real-world based* application as well, despite having lower estimation time accuracy as compared to synthetic applications, *SLAMv* is still able to generate



(a) 3-functions test application (b) 6-functions test application

(c) 10-functions test application (d) Real-world based application

Fig. 6: Percentage of the requests conforming to the given SLOs based on the configurations suggested by *SLAM*.

configurations that result in above 95% precision of requests conforming to the given SLOs.

2) Various objectives configuration finding effectiveness: In this aspect, we determine the effectiveness of *SLAM* tool when requested to optimize for various optimization objectives (§II-E1) in addition to the SLOs. In this regard, we calculate the overall execution time and the cost needed by one invocation of the application when configured with memory configurations selected by *SLAM* for those optimization objectives and compared them against static minimum-memory=128MB (*min-mem*) and maximum-memory=2GB (*max-mem*) configurations to get the worst/best execution times for the applications, and also the corresponding costs. The memory configurations of *min-mem* and *max-mem* signify configurations where each function in the application is configured to that memory. It is to be noted that, it is not necessary that we get the worst/best execution times at the extreme end of execution time [29], therefore we compare them with the global minimum cost (*BF-min-cost*) and execution time (*BF-min-time*) for each application obtained by checking every configuration and function combinations using *Brute force*.

Experiment results on the four test applications are shown in Figure 7 and the results are averaged over 100 application invocations. From Figure 7, we can see that for all the applications, *SLAM* optimization objective algorithms find the optimal/near-optimal cost and time configurations such that they are very close to the global minimum cost (*BF-min-cost*) and time (*BF-min-time*). Since the behavior of the *SLAM* on different applications is very similar, we only explain the results for the *3-functions* application on two objectives:

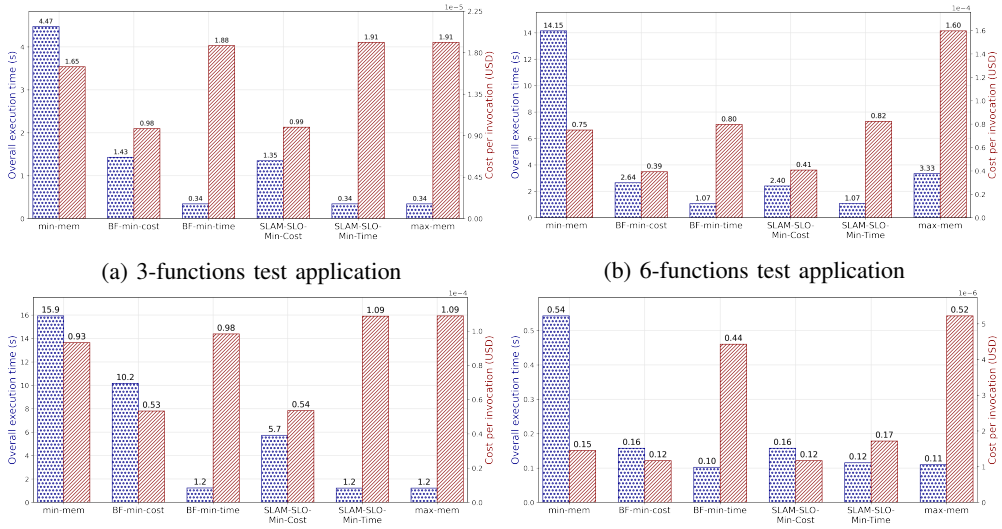
Minimum Overall Cost: For the *3-functions* application, *SLAM-SLO-Min-Cost* ($\$0.99 \times 10^{-5}$ as seen in Figure 7a) is only $\$0.01 \times 10^{-5}$ higher than *BF-min-cost* ($\$0.98 \times 10^{-5}$). When comparing *SLAM-SLO-Min-Cost* with the *min-mem* and *max-mem* configuration, *SLAM-SLO-Min-Cost* takes on average 1.6x less cost than *min-mem* and 1.9x less cost than *max-mem*. Additionally, *SLAM-SLO-Min-Cost* configuration ($1.3s$) is able to process application request faster than the *min-mem* ($4.5s$) and *BF-min-cost* configurations ($1.4s$) but takes longer time than the *max-mem* configuration ($0.3s$).

Minimum Overall Execution Time: For the *3-functions* application, the execution time for *SLAM-SLO-Min-Time* configuration (1.07s as seen in Figure 7a) is equivalent to that of *BF-min-time* configuration and the overall cost for *SLAM-SLO-Min-Time* ($\$0.82 \times 10^{-5}$) is only a bit higher than the *BF-min-time* configuration ($\$0.80 \times 10^{-5}$). This shows that *SLAM* is able to find the optimal/near-optimal execution time configuration such that it is very close to the global minimum execution time configuration (i.e., *BF-min-time*, which requires a long time for determination). From Figure 7a again we can see that the execution time taken by *max-mem* configuration (3.3s) is higher than that of *BF-min-time* configuration (1.07s), therefore it may not always be true that the largest memory results in minimum overall execution time [29]. When comparing *SLAM-SLO-Min-Time* configuration (1.07s) with the *min-mem* (14.15s) and *max-mem* (3.3s) configurations, *SLAM-SLO-Min-Time* configuration takes on average 13.5x less execution time than *min-mem* configuration and 3x less execution time than *max-mem* configuration. Additionally, *SLAM-SLO-Min-Cost* configuration ($1.3s$) is able to process application request faster than the *min-mem* ($4.5s$) and *BF-min-cost* ($1.4s$) configurations but takes longer time than the *max-mem* ($0.3s$) configuration.

C. Q3. SLAM configuration finding efficiency and scalability

In Figure 8, we show how efficient and scalable *SLAM* is in finding the optimal configurations at various objectives. In Figure 8a we can see the time required for different optimization algorithms to find the optimal configuration when run on *6-functions* application. The *Brute-force* algorithm performed worst as compared to the developed optimization algorithm (almost took 871x time more than the developed algorithm). Although, it is possible to parallelize the *Brute-force* search, but it is beyond the scope of this work. When comparing *SLAM-SLO* (0.0182s) with *SLAM-SLO-Min-Cost* (0.0289s) and *SLAM-SLO-Min-Time* (0.0237s), *SLAM-SLO-Min-Cost* requires the most amount of time for this application with 6 functions. This can also be validated from the Figure 8b where the scalability of the three algorithms is tested on applications containing a larger number of functions (from 1 to 100) and *SLAM-SLO-Min-Cost* requires the most amount of time. All algorithms scale linearly with the number of functions in the application, but with different slops and *SLAM-SLO* having the least slope.

SLAM-SLO-Min-Cost, which has to estimate the cost at every step of the search, has to go through a higher number of configurations as compared to *SLAM-SLO* and *SLAM-SLO-Min-Time*. Nevertheless, for an application containing 100 functions *SLAM-SLO-Min-Cost* took 5.5s, which is not a lot considering the benefits of the algorithm in terms of cost-saving.



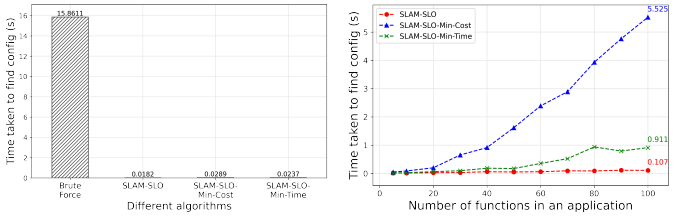
(a) 3-functions test application

(b) 6-functions test application

(c) 10-functions test application

(d) Real-world based application

Fig. 7: Execution time and the cost when configured with configurations selected by *SLAM* for various objectives.



(a) Configuration finding time for different algorithms

(b) Performance of SLAM when the number of functions are scaled

Fig. 8: SLAM efficiency and scalability performance

V. Related Work

With the advent of serverless computing, there is a significant amount of research aimed at optimizing cloud computing resource utilization [30]–[32]. There has been some work on the performance profiling of various FaaS platforms. Wang et al. [22] 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. [33] studied the architectural implications of serverless computing and pointed out that exploitation of system architectural features like temporal locality and reuse are hampered by the short function runtimes in FaaS. Chadha et al. [34] examine the underlying processor architectures for Google Cloud Functions (GCF) and determine the optimization of FaaS functions using Numba can improve performance by and save costs on average.

Furthermore, there is a significant number of research works aimed at optimizing the memory and cost for the FaaS functions. COSE [24] framework finds the optimal configurations for a FaaS function using the Bayesian Optimization algorithm while minimizing the total cost of execution. It not only models the behavior of a function, but also the environment (cloud, edge) in which those functions are deployed. However, they consider FaaS functions separately and optimized based on cost. Bayesian Optimization was also used in CherryPick [35] 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 [36], is designed to optimize FaaS function configurations for specifically map-reduce usecase.

Similar optimization tools have also been developed by Google and Amazon. Google has developed a recommendation system to help the users choose the optimal virtual machine (VM) type [37]. It currently does not support Google Cloud Functions. AWS Compute Optimizer [38] recommends optimal AWS resources for applications to reduce costs and improve performance by using machine learning to analyze historical utilization metrics. It can also be used to find optimal memory configuration for the lambda-based function. However, it can only be executed for the functions whose allocated memory level is less or equal to 1792MB and which are invoked at least 50 times in the last two weeks. AWS Lambda Power Tuning [29] tool uses exhaustive search to identify optimal memory level for a cost, or execution time. By default, this algorithm will need to perform at least 225 requests to the function to identify the optimal memory point.

None of the aforementioned research efforts address the issue of automatically configuring optimal memory of FaaS functions within a serverless application based on the user-defined SLOs. Most of the research either addresses a single FaaS function or an application consisting of step functions that do not have complex call graph workflows. The proposed tool *SLAM* fills that gap by creating a recommendation tool

that in a short time can find optimal memory configurations of FaaS functions within a serverless application given the SLOs.

VI. Conclusion and Future work

Serverless computing has abstracted most cloud server management and infrastructure scaling decisions away from the users, but configuring the memory of FaaS functions is still left up to the users. To solve this problem, we introduced **SLAM** to find optimal memory configurations given predefined SLO requirements. **SLAM** uses a max-heap-based optimization algorithm along with its variants for various optimization objectives (minimum cost and minimum overall time) in finding the optimal memory configuration for the given serverless application based on the specified SLO. It supports complex serverless application call-graph workflows and has the ability to adapt to changes in a serverless application. We demonstrate the functionality of *SLAM* with AWS Lambda (§IV) on four serverless applications consisting of a various number of functions and found that the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs.

In the future, we plan to extend *SLAM* with other public serverless compute providers and to open source FaaS platforms. Transitioning from a discrete search space for the memory configurations to a continuous one could be the next improvement for the *SLAM*.

References

[1] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, "Fedless: Secure and scalable federated learning using serverless computing," in *2021 IEEE International Conference on Big Data (Big Data)*, Dec 2021, pp. 164–173.

[2] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[3] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*. IEEE, 2017, pp. 445–451.

[5] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service(faas) in industry and research."

[6] (2020) Aws lambda releases. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html

[7] C. S. WG, "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].

[8] CloudFlare, "Why use serverless computing?" https://www.cloudflare.com/learning/serverless/why-use-serverless/, accessed: 2020/12/16.

[9] M. Roberts, "Serverless architectures," https://martinfowler.com/articles/serverless.html, 2018, accessed: 2020-04-17.

[10] Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[11] "Cloud functions overview," https://cloud.google.com/functions/docs/concepts/overview, (Accessed on 08/22/2020).

[12] An introduction to azure functions. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview

[13] Amazon Lambda, https://aws.amazon.com/lambda/, accessed on 09/24/2020.

[14] "Azure functions hosting options, https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale," accessed on 02/18/2021.

[15] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20.

[16] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Software: Practice and Experience*, vol. 51, no. 9, pp. 1936–1963, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966

[17] A. Eivy, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, 2017. [Online]. Available: https://doi.org/10.1109/MCC.2017.32

[18] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, 2019.

[19] R. Byrro, "Can we solve serverless cold starts?" https://dashbird.io/blog/can-we-solve-serverless-cold-starts/, 2019, accessed: 2020-04-17.

[20] K. Lane, "Overview of the backend as a service (baas) space," *API Evangelist*, 2015.

[21] J. Spillner, "Resource management for cloud functions with memory tracing, profiling and autotuning," in *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*. ACM, 2020, pp. 13–18. [Online]. Available: https://doi.org/10.1145/3429880.3430094

[22] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*. USENIX Association, 2018, pp. 133–146.

[23] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The spec cloud group's research vision on faas and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, ser. WoSC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 14. [Online]. Available: https://doi.org/10.1145/3154847.3154848

[24] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 129–138.

[25] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," *CoRR*, vol. abs/1811.09721, 2018. [Online]. Available: http://arxiv.org/abs/1811.09721

[26] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," 2021.

[27] AWS, "What is aws x-ray?" https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html, 2020, [Online; Accessed: 4-Feburary-2020].

[28] (2020) Aws lambda pricing. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[29] A. Casalboni, "AWS Lambda Power Tuning." [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning

[30] M. Akin, "How does proportional CPU allocation work with AWS Lambda? — Opsgenie Engineering." [Online]. Available: https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac

[31] S. Kulkarni, "Optimize AWS lambda memory — Towards Data Science." [Online]. Available: https://towardsdatascience.com/optimize-aws-lambda-memory-more-memory-doesnt-mean-more-costs-51ba566fecc7

[32] J. Grogan, C. Mulready, J. McDermott, M. Urbanavicius, M. Yilmaz, Y. Abgaz, A. McCarren, S. MacMahon, V. Garousi, P. Jamshidi *et al.*, "An analysis of function-as-a-service (faas): vendors, challenges and implications for software developers."

[33] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1063–1075.

[34] M. Chadha, A. Jindal, and M. Gerndt, "Architecture-specific performance optimization of compute-intensive faas functions," 2021.

[35] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 469482.

[36] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Astra: Autonomous serverless analytics with cost-efficiency and qos-awareness," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 756–765.

[37] “Google cloud recommendations,” 2018, (Accessed on 06/17/2021). [Online]. Available: https://cloud.google.com/compute/docs/instances/apply-machine-type-recommendations-for-instances
[38] “Aws compute optimizer,” 2021, (Accessed on 06/17/2021). [Online]. Available: https://aws.amazon.com/compute-optimizer/