Function Memory Optimization for Heterogeneous Serverless Platforms with CPU Time Accounting

Robert Cordingly
School of Engineering and Technology
University of Washington
Tacoma, United States
rcording@uw.edu

Sonia Xu
School of Engineering and Technology
University of Washington
Tacoma, United States
sxu253@uw.edu

Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma, United States
wlloyd@uw.edu

***Abstract*—Serverless Function-as-a-Service (FaaS) platforms often abstract the underlying infrastructure configuration into the single option of specifying a function's memory reservation size. This resource abstraction of coupling configurations options (e.g. vCPUs, memory, disk), combined with the lack of profiling, leaves developers to make ad hoc decisions on how to configure functions. Solutions are needed to mitigate exhaustive brute force searches of large parameter input spaces to find optimal configurations which can incur high costs.**

To address these challenges, we propose CPU Time Accounting Memory Selection (CPU-TAMS). CPU-TAMS is a workload agnostic memory selection method that utilizes CPU time accounting principles and regression modeling to recommend memory settings that reduce function runtime and subsequently, cost. Comparing CPU-TAMS to eight existing selection methods, we find that CPU-TAMS finds maximum value memory settings with only 8% runtime and 5% cost error compared to brute force testing while only requiring a single profiling run to evaluate function resource requirements. We adapt CPU-TAMS for use on four commercial FaaS platforms demonstrating efficacy to optimize function memory configurations where platforms feature heterogeneous infrastructure management policies.

***Index Terms*—Serverless Computing, Function-as-a-Service, Performance Evaluation, Performance Modeling**

I. Introduction

The serverless computing design paradigm has grown to become a popular execution model offered by many cloud computing providers. Amazon Web Services (AWS), Google Cloud, IBM Cloud, DigitalOcean, and Microsoft Azure all provide serverless computing services that offer automatic scaling, billing models that only charge for resource use, and fault tolerance with minimal infrastructure configuration [1]–[5]. Function-as-a-Service (FaaS) is a common serverless cloud computing delivery model where developers deploy microservices that run in isolated environments known as function instances [6]. FaaS platforms automatically create, destroy, and load balance requests among many function instances. Cloud consumers are billed only for individual requests and runtime of function instances. If no requests are made, consumers are not billed, while deployed functions remain highly available and ready to rapidly scale to meet future changes in demand.

Functions on FaaS platforms require configuring different parameters to optimize performance and hosting cost, the most common being memory reservation size which usually determines the number of vCPUs allocated to the function. If configured sub-optimally, FaaS functions can perform slowly and cost an order of magnitude more than the same function hosted using traditional Infrastructure-as-a-Service platforms.

In this paper, we describe and evaluate a novel method known as CPU Time Accounting Memory Selection (CPU-TAMS) which leverages regression modeling to identify desirable function memory reservation sizes to automate FaaS function configuration. While memory selection appears as simple as choosing a setting higher than what a function requires to prevent out-of-memory errors, FaaS platforms often scale infrastructure performance and cost to a function's memory reservation size [1], [4]. Without knowing how their functions will perform or scale with respect to specific memory configurations, developers are left to make ad hoc decisions on which setting to choose. Many developers use common practices to configure function memory, such as selecting the lowest memory setting possible, selecting the maximum memory setting, or retaining the default setting [7]. These options, however, may lead to slower performance and higher costs [6], [8], [9]. CPU-TAMS targets finding the MAX-VALUE memory setting, the setting that offers the highest performance at the lowest cost.

The CPU-TAMS methodology is workload agnostic and can be applied to support function memory configuration on many FaaS platforms. In this paper, we describe our implementation of CPU-TAMS on AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and DigitalOcean Functions. We compared the accuracy of our approach to eight baseline memory selection approaches including: minimum required memory, maximum platform memory, a mid-range setting between min and max, brute force search, linear search, binary search, gradient descent, and the AWS Compute Optimizer. The AWS Compute Optimizer is a service offered by AWS that recommends optimal AWS Lambda configurations to reduce costs and improve performance by using machine learning to analyze historical utilization metrics.

A. Research Questions

This paper investigates the following research questions:

RQ-1: (FaaS Performance Scaling) How do workload characteristics, such as CPU, disk, or network utilization, impact FaaS function performance, and how is a function's share of these resources scaled with memory reservation size?

RQ-2: (FaaS Memory Configuration) How accurately can we predict FaaS function memory reservation size to achieve CHEAPEST, FASTEST, or MAX-VALUE deployments using CPU-TAMS compared to baseline memory selection methods and rules of thumb?

B. Paper Contributions

This paper makes the following research contributions:

1) We present our CPU Time Accounting Memory Selection (CPU-TAMS) approach that leverages CPU metrics to predict maximum value FaaS memory configurations. We identify common goals for function memory configurations (e.g. CHEAPEST, FASTEST, MAX-VALUE), and compare our memory configuration approach to eight baseline approaches.
2) Using a suite of twelve functions we investigate memory selection methodologies to quantify their effectiveness at selecting settings to meet specific goals (e.g. CHEAPEST, FASTEST, MAX-VALUE).
3) We apply CPU-TAMS to the AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and DigitalOcean Functions FaaS platforms to investigate implications of heterogeneous platform scaling policies and differences in observability of CPU metrics when profiling functions on specific platforms.

II. Background

The challenge of FaaS function configuration has been addressed by existing literature in two ways: research that observes the performance and scaling characteristics of serverless platforms, and methods to predict optimal serverless function configurations.

A. Serverless Performance Modeling and Scaling

Several efforts have investigated performance implications for hosting a variety of workloads using serverless computing including: scientific computing [10]–[13], machine learning inferencing [14], [15], NLP inferencing [16], and even neural network training [17]. Other literature have evaluated how performance scales on FaaS platforms when changing function configurations. Wang et al. identified AWS Lambda performance at 128MB as only ∼1/10th of 1-core VM performance in [6] which more recently is estimated to be ∼1/12th of 1-core performance [18]. Multiple efforts have investigated the impact of memory settings on the performance and cost of serverless functions [8], [19]. These studies found that functions often exhibit behaviour where the lowest memory setting does not offer the lowest cost due to increased runtime. A 2020 survey found that 86% of functions on AWS Lambda use a memory setting less than 512MB, with over half using the default minimum memory of 128MB [7]. Copik et. al and Kim developed a suite of functions for different FaaS platforms and evaluated the impact of memory settings on performance, showing that the lowest memory setting (128MB) often provided slower performance and higher costs [20], [21]. Multiple authors have bench marked various workload characteristics of FaaS platforms such as memory scaling, cold start performance, latency, and network I/O [8], [22]. Many research efforts that benchmark serverless performance did not address optimizing function memory. Recent enhancements to FaaS platforms, such as the ability to increase the maximum function memory up to 10GB on AWS Lambda, have introduced new challenges not yet addressed. For example, after 3GB of RAM (which was the previous maximum on AWS Lambda) function instances are allocated more than 2 vCPUs, up to a maximum of six at 10GB. The second generation of Google Cloud Functions also introduced expanded memory options up to 16GB.

B. FaaS Memory Configuration Methods

Different methods of configuring memory for serverless functions have been investigated previously. Tools such as the AWS Lambda Power Tuning tool support the automation of brute force testing to find different configuration goals [23]. Eismann et al. proposed a modeling method called Sizeless to predict optimal memory reservation size on AWS Lambda using multi-target regression modeling [24]. Akhtar et al. proposed COSE, another modeling approach that utilized Bayesian Optimization to find optimal serverless configurations [25]. Cordingly et al. utilized CPU metrics to create regression models to predict runtime across different memory configurations [26], [27].

All three methods [24]–[26] predicted memory configurations by estimating function runtime across a variety of memory settings starting with one base configuration. In these examples, evaluations were limited to a finite set of memory sizes (e.g. 128MB, 256MB, 512MB), and these sizes did not span the 10240MB range available on AWS Lambda or consider the corresponding scaling of vCPU cores. These modeling approaches also required large training data sets resulting in high costs, limited support for evaluating functions written in different programming languages, and the requirement for models to be trained for specific workloads.

In [28], the authors proposed methods to model performance of serverless platforms focusing on optimization of infrastructure state (cold vs. warm), and the infrastructure life cycle on serverless platforms. In [29], the authors predicted performance of serverless workflows using mixture density networks and continuous model learning to predict FaaS round trip time. Another common approach to finding optimal memory configurations is creating techniques to search through the range of available memory settings. Zubko et. al. created the Memory Allocation Framework for FaaS functions (MAFF) to search for optimal memory settings using a linear search, binary search, and gradient descent algorithms [30]. Safaryan et. al. created a search technique that focuses on optimizing configurable service level objectives using a max-heap binary tree data structure for their search algorithm [31]. Even with a small 2GB memory setting range these search techniques

require dozens of function invocations. For larger ranges of memory, or higher precision, hundreds or even thousands of function invocations may be required.

In this paper, we extend the CPU time accounting performance modeling approach initially described in [32] and [26] to predict MAX-VALUE FaaS memory configurations for serverless functions. Compared to search techniques that require hundreds or thousands of function invocations to profile performance, our CPU-TAMS approach only requires a single function invocation. We investigate the accuracy of CPU-TAMS for many different workloads across four commercial FaaS platforms: AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and DigitalOcean Functions. All previous work in the area of FaaS function memory configuration focused only on AWS Lambda.

III. Methodology

This section details tools and methods used to investigate our research questions (RQ-1, RQ-2).

A. CPU Time Accounting Memory Selection (CPU-TAMS)

Commercial FaaS platforms present developers with a variety of different function configuration options. The most common parameter developers must configure is the function's memory setting. Function memory settings must be configured on AWS Lambda, Google Cloud Functions, DigitalOcean Functions, and IBM Cloud Functions [1], [3], [4]. For each of these platforms, function memory settings can impact the performance and cost [26]. Without comprehensive knowledge on how performance is impacted by memory configurations on each platform, developers are left to make ad hoc decisions on how to select memory settings. Over-provisioning and under provisioning of function memory can lead to overspending. The abstract nature and limited documentation of FaaS platforms necessitates the need for developers to reverse engineer or heavily profile platforms to understand performance implications of memory selection for their functions.

TABLE I
Memory Selection Goals

Objective	Description
CHEAPEST	Lowest hosting cost with no regard to runtime.
FASTEST	Lowest runtime with no regard to cost.
MAX-VALUE	Maximizes the ratio between cost and performance. Offering both high performance and reduced cost.

CPU-TAMS utilizes CPU metrics collected from the function instance's operating system combined with regression modeling to enable finding ideal FaaS function memory configurations easily. The Linux /proc filesystem provides metrics that detail CPU time spent executing in different modes. The Severless Application Analytics Framework (SAAF) reports CPU metrics for time spent in: user mode (CPU User), kernel mode (CPU Kernel), idle mode (CPU Idle), waiting for I/O (CPU I/O Wait), or servicing interrupts (CPU Int Srvc and CPU Soft Int Srvc). Using these CPU metrics, we calculate the wall clock execution time of a workload by summing all the CPU metrics and dividing by the number of vCPUs of the function's virtual environment:

$$Runtime = \frac{\begin{matrix} cpuUsr + cpuKrn + cpuIdle + cpuIOWait + \\ cpuIntSrvc + cpuSoftIntSrvc \end{matrix}}{\#\ of\ vCPUs} \quad (1)$$

In addition to calculating workload runtime, we can leverage CPU metrics to estimate the average number of utilized vCPUs of a workload. By removing CPU idle time, summing the other CPU metrics, and dividing by runtime, we can solve for the number of utilized vCPUs of a workload:

$$Utilized\ vCPUs_{pred} = \frac{\begin{matrix} cpuUsr + cpuKrn + cpuIOWait + \\ cpuIntSrvc + cpuSoftIntSrvc \end{matrix}}{Runtime_{obs}} \quad (2)$$

After predicting the number of utilized vCPUs, for each FaaS platform supported by CPU-TAMS, we have created a specific vCPU-to-memory model to map function memory settings to a distinct number of vCPUs. We then leverage this model to map the required number of vCPUs to the appropriate memory setting. Using this approach, functions can be provisioned to have the highest performance at the lowest cost (MAX-VALUE).

1) CPU-TAMS on AWS Lambda: To develop the AWS Lambda vCPU-to-memory model used to recommend memory settings with CPU-TAMS, we deployed a function with the Stress(1) tool and ran tests at 40 different memory settings: from 128MB to 10GB in steps of 256MB [33]. The Stress(1) tool imposes a load on systems with a variety of options. We used the Stress(1) tool to maximize CPU stress to measure how many vCPUs are available to the execution environment enabling observation of the maximum usable vCPU time share allocated at each memory setting as shown in Figure 1. CPU time share scaled linearly with function memory. Each additional 1GB of memory added an additional 0.57 vCPUs capacity. The lowest setting, 128MB allocates just 8% of one vCPU to the function instance.

Using our vCPU-to-memory model shown in Figure 1, we can profile any FaaS function at the maximum memory setting to obtain CPU metrics. With these CPU metrics, we then calculate the number of utilized vCPUs as shown in equation (2) and then use our regression model to solve for the memory setting where the number of utilized vCPUs is equal to the allocated vCPUs:

$$Memory_{pred} = \frac{Utilized\ vCPUs_{obs} - 0.012346}{0.000556} \quad (3)$$

(Applying vCPU-to-memory model to predict memory selection on AWS Lambda.)

CPU-TAMS has been designed to be workload agnostic and prevent over-provisioning of resources. Over-provisioning occurs when a function's memory setting offers the function more vCPUs than the workload is able to use. CPU-TAMS also does not try to find the CHEAPEST option, but instead favors selecting MAX-VALUE which is the memory setting that maximizes the value metric defined in equation (5). Algorithm 1 defines the process of using the CPU-TAMS method alongside the model used to derive equation (3) in Figure 1.

Algorithm 1 CPU Time Accounting Memory Selection

Require: Configure function to use max FaaS platform mem.
1. Profile workload to collect CPU metrics.
2. Calculate utilized vCPUs using equation (2)
3. Solve for memory using equation (3) derived from the linear regression model in Figure 1.
4. Adjust function memory to recommendation or the required memory reported in logs (whichever is higher).

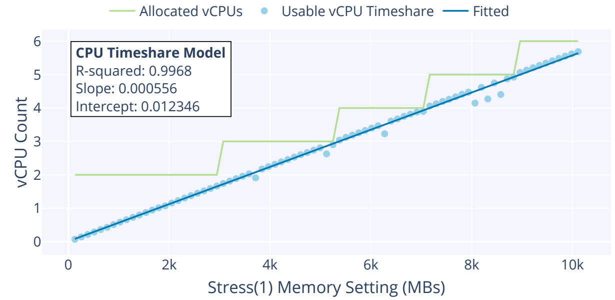

Fig. 1. Allocated vCPUs available at each memory setting and available timeshare observed using the Stress(1) function on AWS Lambda.

CPU-TAMS enables developers to quickly find MAX-VALUE memory settings offering a number of benefits compared to other memory selection techniques. Ideally, when CPU performance scales linearly with memory, a function with double the memory would require only half the runtime, resulting in identical cost as shown by equation (4). If this were the case, selecting the highest memory setting would always be the best choice since all other memory settings result in slower performance, with reduced cost per GB/s, but have identical total cost. For the vast majority of functions, performance does not scale linearly across all memory settings [8].

One flaw of selecting the CHEAPEST or FASTEST memory setting is that they may produce undesirable consequences. FASTEST can be orders of magnitude more expensive than a more conservative memory setting. Conversely, CHEAPEST may lead to prohibitively slow function runtimes making functions unusable. In this paper, in contrast to the CHEAPEST or FASTEST memory configurations, we are interested in finding memory settings that offer the MAX-VALUE, that is *high performance at a reasonable cost*. These three performance-cost objectives for FaaS function memory configurations are defined in Table I. We calculate value by multiplying the cost of a function invocation by negative runtime. The value metric in equation (5) is maximized by CPU-TAMS to achieve high performance at a low cost. This value metric exploits a trend in FaaS platforms where cost and runtime are not linearly scaled together. Low memory settings have very high runtime and lower costs. High memory settings have low runtime but significant costs. Both scenarios result in low value. A memory setting in the middle may have nearly the same performance as the high memory setting but much closer cost to the low memory setting, resulting in high value. Increasing cost without obtaining a performance improvement results in lower value. The value metric is always negative and values closest to zero offer the best price to performance ratio.

$$cost = memory_{GB} * runtime_S * 0.00016667_{\$} \tag{4}$$

(AWS Lambda Pricing Policy [1])

$$value = -runtime_S * cost_{\$} \tag{5}$$

Due to the fundamental differences in how FaaS platforms provision vCPUs with each memory setting, a one-size-fits-all approach to function memory configuration is not feasible. Instead, the vCPU-to-memory model must be adapted to account for the unique characteristics of each FaaS platform.

2) CPU-TAMS on IBM Cloud Functions: To implement CPU-TAMS on IBM Cloud Functions (IBM), the vCPU-to-memory model was trained using the Stress(1) function by measuring how the number of available vCPUs change by profiling the function using a set of increasing memory settings. Since IBM offers a small range of memory settings, we tested every 128MB interval using brute force search. Unlike on AWS Lambda where all function instances are isolated, CPU metrics are reported using host-level (i.e. VM-level) metrics on IBM. Multiple function instances can share the same host so equation (2) must have all CPU metrics divided by the observed number of tenants after testing.

By maximizing the tenancy of host VMs, we observed the available CPU timeshare of every memory setting. Using a combination of the IBM vCPU-to-memory model, and a rule of thumb *to minimize memory if concurrency is known to be low (≤4)*, we can produce accurate memory recommendations.

Unlike AWS Lambda where only a single function invocation is needed, CPU recommendations can not be immediately generated after a single run as the function tenancy must be known. On IBM, for all function memory settings, function instances share the host's 4 available vCPUs and must compete for CPU time on the VM. At the maximum memory setting (2048MB), IBM limits the maximum tenancy to 4, essentially allowing each function instance access to the timeshare of 1 vCPU. IBM limits host tenancy to a maximum of 4 function instances at 1.5GB. This creates a 'sweet spot' memory setting that offers equivalent performance to the maximum memory setting without the additional cost as higher settings do not provide any additional vCPUs as show in Figure 2.

3) CPU-TAMS on Google Cloud Functions: Implementing CPU-TAMS on Google Cloud Functions (GCF) provides a unique challenge not present on AWS Lambda or IBM. CPU metrics required to calculate the number of utilized vCPU, for CPU-TAMS are not visible to function instances. While CPU metrics may be hidden unlike other platforms we investigated, GCF allows users to observe the exact number of vCPUs allocated to their functions at any memory setting. This openness allows us to create the vCPU-to-memory model without CPU timeshare metrics. We simply run any function (even a basic Hello World function), and observe the reported vCPUs at each memory setting, where vCPUs are reported by Google using the 'CPU' read-only function attribute in the CLI. When CPU metrics are hidden, functions can be profiled locally, or on a different FaaS platform to obtain the number of utilized vCPUs and then use the GCF vCPU-to-memory model to

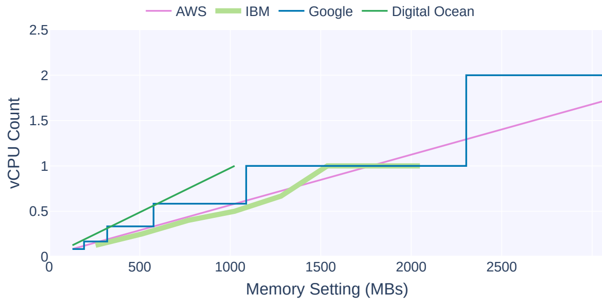

Fig. 2. Allocated vCPUs available at each memory setting on each platform.

predict the MAX-VALUE memory setting. While this process is not as streamlined compared to AWS Lambda or IBM where CPU metrics are directly observable, our workaround enables CPU-TAMS to recommend function memory configurations on GCF. GCF appears to use a tiered approach where doubling memory settings doubles the number of allocated vCPUs as shown in Figure 2. vCPU scaling for all platforms is depicted in the figure with the exception of GCF at 4 vCPUs which is omitted for readability.

4) CPU-TAMS on DigitalOcean Functions: CPU-TAMS can be applied to recommend function memory configurations for DigitalOcean Functions (DOF). Like IBM, DOF reports the use of OpenWhisk. Consequently, both platforms exhibit the same behavior where VM vCPUs are shared between multiple function instances. Unfortunately, similar to GCF, DOF hides all CPU metrics and information to support determining the number of function instances that share the host. Since function tenancy on a shared host has a large impact on function runtime, we estimated maximum tenancy by using our CPU-bound Calcs function [26].

We gradually increased the number of concurrent function invocations until identifying one invocation with equal runtime to the initial sequential function call, indicating it ran on a unique host and all of the other function calls shared the same host. Across all function memory settings, DOF share and compete for the available CPU time share from 8 vCPUs allocated by the host. In the same manner as IBM, at the maximum memory setting (i.e. 1GB) DOF limits the maximum tenancy to 8 allowing each function instance access to the timeshare of 1 vCPU. DOF has the narrowest available memory range from 128MB to 1024MB as shown in Figure 2. Using these four vCPU-to-memory models we can recommend memory settings on all four of these platforms based upon a workload's CPU Time Accounting profile.

B. Baseline Memory Selection Methods

Many developers resort to using basic rules of thumb to choose a memory setting: selecting the maximum option, choosing the minimum memory needed for their function to run, picking a memory setting in the middle of the available options, or leaving the function at the default setting [7]. While these approaches are simple, they often result in function memory configurations that are more expensive, significantly slower, or both compared to a MAX-VALUE setting.

TABLE II
MEMORY SELECTION METHODS

Name	Type	Description
MIN	RoT	Select the minimum memory required for a function
MID	RoT	Select a mid-range setting between MIN and MAX
MAX	RoT	Select the maximum available memory setting
CPU-TAMS	M	Utilize CPU Time Accounting metrics to calculate the average number of utilized vCPUs for a workload. Use vCPU model to predict value setting.
AWS-CO	M	The AWS Compute Optimizer. A tool by AWS that recommends memory settings. Requires >50 runs below 1792MB to make a recommendation.
Linear Search (LS)	S	Run a workload at many different memory settings, iterating linearly, and stopping when the settings meets a target goal.
Binary Search (BS)	S	Search through memory settings by iterating using a binary search algorithm. Test settings and progressively cut the memory setting range in half.
Gradient Descent (GD)	S	Search through memory settings by iterating using a gradient descent algorithm. Test settings and progressively move toward a value memory setting.
Brute Force	S	Run a workload at every available memory setting iterating with a desired step size.

(**RoT**: Rule of Thumb, **M**: Model, **S**: Search Method)

In this paper, we contrast our proposed CPU-TAMS approach with eight baseline memory selection approaches: three rule of thumb methods (e.g. MIN, MAX, and MID), one model based approach (e.g. AWS Compute Optimizer), and four search methods (e.g. brute force search, linear search, binary search, and gradient descent). Each of these methods targets a specific memory selection goal: CHEAPEST, FASTEST, and MAX-VALUE for MIN, MAX, and MID respectively. CPU-TAMS and the AWS Compute Optimizer targets the MAX-VALUE goal [34]. Search methods have the ability to target any goal memory setting.

To compare our CPU-TAMS selection method we evaluate the AWS Compute Optimizer (AWS-CO) service offered alongside AWS Lambda. AWS-CO requires an initial 50 function invocations and then makes memory setting recommendations to reduce cost. As this is a commercial product offered by AWS that utilizes machine learning and historical utilization data, the method of how the Compute Optimizer makes recommendations is unknown [34]. Major limitations of the AWS-CO include: it is not able to make recommendations for functions requiring over 1792MB of RAM, it can take up to 48 hours to generate a recommendation, and availability is limited to AWS Lambda.

To compare the accuracy of memory selection methods, we utilize brute force search as a baseline method. For brute force search, we profiled all of our functions on each platform using increasing memory settings from MIN to MAX in 128MB steps on IBM/DOF, and 256MB steps on AWS Lambda/GCF. Our resulting dataset includes both cold-start and warm function invocations at a 1 to 9 cold to warm ratio. With our complete brute force search dataset, we then implemented linear search, binary search, and gradient descent by iterating over the data to find the CHEAPEST, FASTEST, and MAX-VALUE memory settings. Search methods require executing functions across many different memory settings to find their selection goals. The accuracy of search methods also depends on step size (e.g. 128MB) used to search the range of

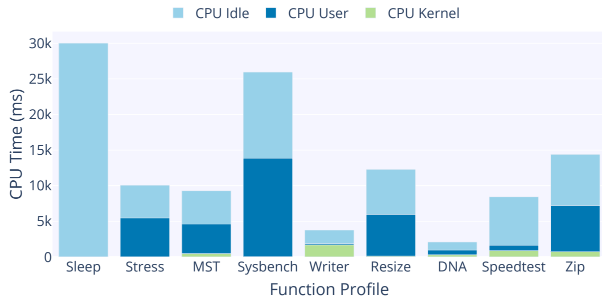

Fig. 3. Distribution of CPU timeshare at 1920MB of memory on AWS.

memory configurations. GCF supports a fine-grained 1MB step size, but profiling across all 16,256 options was impractical due to significant time and cost. To reduce profiling costs, other researches have employed similar step values to reduce the search space for FaaS function memory optimization [30]. Ultimately, applying any search method involves a large amount of profiling overhead and may result in high costs making search methods not viable for applications consisting of many functions.

C. Supporting Tools

To enable a better understanding of the performance implications for functions deployments to FaaS platforms, we developed the Serverless Application Analytics Framework (SAAF) [18], [35]. SAAF is a profiling framework supporting multiple programming languages which is included as a library in functions deployed to multiple FaaS platforms. SAAF supports analysis of functions deployed to AWS Lambda, GCF, DOF, IBM, Azure Cloud Functions, and OpenFaaS platforms [1], [2], [36], [37].

SAAF collects metrics from multiple sources inside the Linux operating system, including the **/proc** file system and environment variables created by the FaaS platform. SAAF's design allows all metrics to be collected by simply including the framework in the deployment package and adding a few lines of code to the beginning and end of the function's source code. Each FaaS platform exposes or hides different metadata about the underlying Linux environments that run functions. SAAF is specifically designed for FaaS platforms; it adds minimal overhead to functions and works around the different levels of infrastructure obfuscation of each platform. SAAF collects CPU metrics used by CPU-TAMS from within the function instance, so that profiling is unaffected by cold-start latency. To automate complex experiments on FaaS platforms, we created a supporting tool called the FaaS Runner. FaaS Runner provides a client-side application that is used in conjunction with SAAF. Both SAAF and FaaS Runner are invaluable tools for scientists and practitioners to profile functions and execute experiments on FaaS platforms.

D. Functions and Experimental Workloads

To compare CPU-TAMS to existing memory configuration methods, we leveraged a test suite consisting of 12 functions. Table III describes each function, the number of vCPUs they utilize, and which FaaS platforms they support. Functions were chosen carefully so collectively our tests would feature a broad range of resource utilization characteristics such as CPU-bound, disk I/O bound, or network I/O bound as described in Figure 3.

TABLE III
FUNCTION NAMES AND DESCRIPTIONS

Function	Clouds	vCPU	Description
Sysbench	AI	n	Linux Benchmark used to generate prime numbers.
MST	AGID	1	Generates a graph and calculates the min spanning tree.
BFS	AGID	1	Generates a graph and processes a breadth first search.
Page Rank	AGID	1.2	Generates a graph and processes page rank of each node.
Writer	AGID	1	Generates text and repeatedly writes it to disk and deletes.
Compress	AGID	1	Generates files and compresses them into a zip file.
Resize	A	1	Pulls an image from S3, resizes it and saves it back to S3.
DNA	A	0.9	Pulls DNA sequence from S3 and creates visualization data.
TLQ	A	N/A	4 transform-load-query data pipelines 4 (Java/Python/Go/Node.js).
Speed Test	A	N/A	Network speed test created by Ookla.
Random Reader	A	1	Container that includes large files which are randomly read.
Calcs	AGID	n	Executes random math operations.
Stress(1)	AI	n	Linux tool used to generate CPU stress.
Sleep	AGID	0	Sleeps for a specified duration.

Clouds: **A**WS, **G**CF, **I**BM, **D**OF
Unshaded: CPU-TAMS Evaluation Functions, Shaded: Profiling Functions

Six functions are primarily CPU bound. The minimum spanning tree (MST), breadth-first search (BFS), and Page Rank functions all generate graphs and process their respective algorithms on each graph. The DNA processing function downloads a DNA sequence file from Amazon S3 and generates visualization data for the DNA sequence. We created a wrapper function to deploy the popular Linux benchmark Sysbench to generate prime numbers, and the Linux Stress(1) tool. In contrast to functions that utilize two or fewer threads, Sysbench, Calcs, and Stress(1) utilize all available vCPUs. Three functions stress the storage volume of execution environments: The Reader and Writer function generate I/O stress by either reading large files included with the function or repeatedly writing and deleting data in **/tmp**. The File Compressor function generates files in the temporary volume and repeatedly compresses them to Zip archives. The Image Resizer function downloads images from S3, scales them to a variety of different resolutions, and then uploads the images back to S3 to stress both CPU and network I/O.

Five functions were derived from the publicly available SeBS: Serverless Benchmarking Suite [20] with minimal changes to integrate SAAF. These functions enable further evaluation of different aspects of the platform while testing memory selection methods with many different workloads. Finally, we utilized four different versions of a transform-load-query data processing pipeline each written in a different language (e.g. Java, Python, Node.js, and Go) to evaluate CPU-TAMS's recommendations on functions with varying

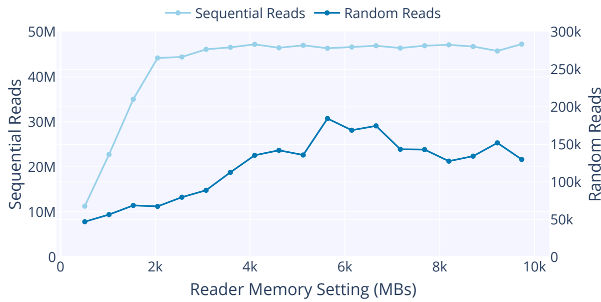

Fig. 4. Sequential and random disk read performance over 10 seconds by function memory setting on AWS Lambda.

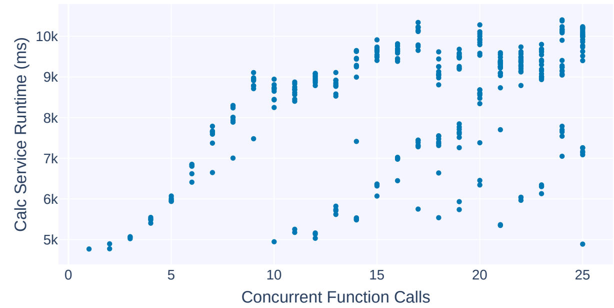

Fig. 5. Calcs function performance on DigitalOcean.

workloads and functions of different languages. Combined the suite of 12 FaaS functions and TLQ pipeline was used to investigate the utility of CPU-TAMS to provide a workload agnostic memory recommendation method.

All functions used are described in Table III, shaded rows indicate profiling functions used to investigate RQ-1 and build each platform's vCPU-to-memory model, all other functions were used to evaluate CPU-TAMS. To perform brute force search used as a baseline to compare accuracy of the memory selection methods described in Table II, we executed each of the functions 10x at each memory step (128MB on IBM/DOF, 256MB on AWS/GCF) to cover the entire range of each platform's memory settings. In addition to the 400 base runs on AWS, each function was executed an additional 60x at 1024MB to allow the AWS Compute Optimizer to make a memory setting recommendation.

IV. Experimental Results

To evaluate our research questions, we deployed the functions described in Table III. First, we executed each FaaS function over many different memory settings (brute force search) to observe the performance implications of memory configuration (RQ-1). Second, we analyzed the results to determine target memory settings to meet configuration goals described in Table I. Third, we investigated the memory selection methods described in Table II to observe how closely they were able to determine desirable memory settings (RQ-2).

A. FaaS Performance Scaling Evaluation

Of the twelve functions, Stress(1), Random Reader, and Speed Test are profiling functions that enabled the evaluation of CPU, disk, and network performance relative to function memory. As expected on AWS Lambda, CPU performance scaled linearly as memory settings increased. Parallel workloads that leverage multiple threads showed linear performance improvements as memory was increasingly scaled. Workloads that utilized only one thread did not demonstrate performance improvements above 1792MB, the point where function instances gain additional vCPUs. Figure 1 depicts how vCPU timeshare scaled with function memory using the Stress(1) function. On IBM and DOF, CPU performance was constant across all memory settings in this initial test. To inspect these platforms further, we executed an experiment that progressively increased the number of concurrent function invocations. We observed performance degradation as the number of concurrent calls increased as shown in Figure 5. IBM and DOF do not directly couple vCPUs to the memory setting and instead limit the number of tenants on the host that compete for CPU time. On GCF performance scaled similarly to AWS Lambda where sequential performance improved up until functions where allocated 1280MB and then runtime remained constant as shown in Figure 6.

Our Random Reader function demonstrated how file read performance scaled with the function's memory setting. To measure file read performance, we deployed a container-based function to AWS Lambda using a Docker container. Docker containers on AWS Lambda allow much larger function package sizes, up to 10GB. The Reader function included six large 1GB text files consisting of lines of random characters was then executed for 10 seconds to measure random and sequential read throughput. We observed a drastic performance difference between sequential reads and random reads. With a 2GB function memory configuration, sequential read throughput was ∼4.5m lines per second while random read throughput was just ∼15k lines per second as shown in Figure 4. Sequential read performance scaled with function memory, but not linearly. Performance scaled from 1m reads per second at 128MB of memory but rapidly hit a roof around 4.5m reads per second at ∼2GB. Increasing function memory had a larger impact on random read performance scaling from 5k reads per second at 128MB to 15k reads per second at 5GB. GCF and DOF do not support deploying functions as containers so this test was not performed on those platforms.

We measured the impact of scaling function memory on network I/O throughput. Using the Speed Test function, we observed how function network download throughput scaled on AWS Lambda from a low of 400Mbps at 128MB memory to a

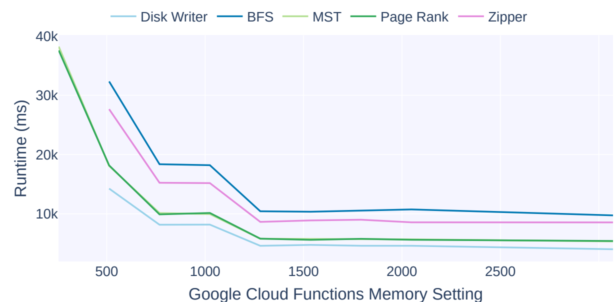

Fig. 6. Google Cloud Functions runtime.

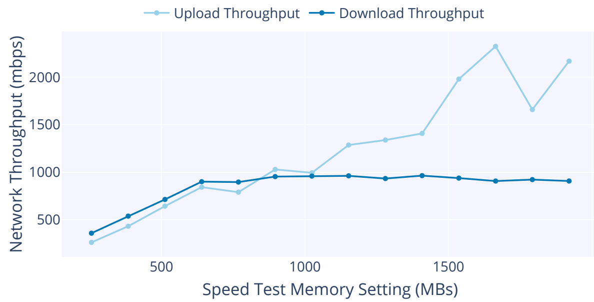

Fig. 7. Speed Test network performance by memory setting on AWS.

peak of ∼1Gbps at 768MB. Upload throughput was lower than download at low memory settings scaling from ∼200Mbps at 128MB to a peak between 2 to 2.5Gbps above 1.5GB of memory. Beyond 1.5GB of memory up to the maximum 10GB, network throughput remained at 1Gbps download and ∼2Gbps upload as shown in Figure 7. On AWS, if a function requires the highest possible network throughput, developers should select a memory setting above 1.5GB. As this test depended on containers and 3rd party libraries it could not be deployed to GCF or DOF.

In summary for RQ-1, we observed that CPU, file read, and network I/O performance scaled relative to a function's memory setting on each platform. CPU performance usually scaled linearly across all memory settings with workloads that utilize all available vCPUs. On AWS, disk and network throughput scaled up peaking around 2GB of function memory. Network performance had a maximum throughput of 1Gbps download, and ∼2Gbps upload. File read throughput also scaled with function memory, and performance benefited from successive function calls which appeared to warm file caches. As a rule of thumb, *if a workload requires maximum network or disk performance, a memory setting of over 2GB is recommended.* On IBM and DOF, performance did not scale as memory setting increased but instead limited the maximum number of co-located function instances, indirectly improving performance at higher memory settings. Performance on GCF scaled similarly to AWS Lambda but the lack of containers did not allow us to utilize all of the benchmark tools. Table IV compares the infrastructure available on each of the FaaS platforms we investigated, how performance scales with memory, and whether CPU metrics are observable.

TABLE IV
FAAS PLATFORM COMPARISON

FaaS Platform	Memory (MB)	Scales w/ Memory	vCPU Cores	CPU Metrics
AWS Lambda	128-10240	CPU Timeshare	2-6	Available
IBM CF	128-2048	Max Tenancy	4	Available
DigitalOcean	128-1024	Max Tenancy	8	N/A
Google CF	128-16384	CPU Timeshare	1-5	N/A
Azure Functions	1536	N/A	2	Available
OpenFaaS	Any	N/A	Any	Available

B. Serverless Memory Configuration Predictions

1) CPU-TAMS on AWS Lambda: While the function memory configuration guidelines discussed in the previous section

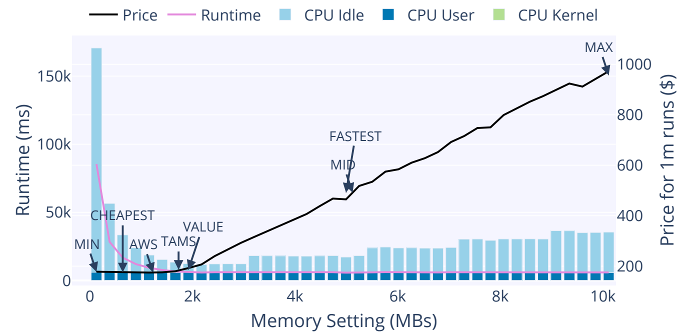

Fig. 8. Resize runtime, cost, CPU profile, and selected settings on AWS.

are helpful to ensure maximum CPU, file read, or network performance, their derivation relies on brute force testing. To derive these recommendations, we executed workloads over many memory settings to observe the results. For some functions, such as the Speed Test function, brute force memory testing can incur a high cost as data egress (e.g. from the network speed test) on AWS Lambda is billed the same as EC2. This section discusses all of the memory selection methods, their accuracy for different memory selection goals, and the data required to make CHEAPEST, FASTEST, and MAX-VALUE memory recommendations.

We initially executed all of the functions across 40 stepwise memory settings from 128MB to 10GB using the brute force method. We obtained results to identify the CHEAPEST, FASTEST, and MAX-VALUE memory configurations using a search granularity of 256MB to compare with other memory selection methods. Figure 8 shows the CPU profile, runtime, cost, and memory settings selected by each memory selection method for the Resize function. In Figure 8, annotations show the three memory selection goals (e.g. CHEAPEST, VALUE, and FASTEST), which we found using brute force search. Annotations for our three rules of thumb (e.g. MIN, MID, and MAX memory) highlight the extreme differences in cost selecting the highest memory setting can have. The recommended memory settings generated by the AWS Compute Optimizer (AWS-CO) and CPU-TAMS are also shown in the figure. On average across all functions, CPU-TAMS recommended memory settings with an average error of 236MB from the MAX-VALUE memory setting while the AWS Compute Optimizer recommended memory settings with an average error of 1916MB. Rules of thumb have significantly higher error (2570MB, 3669MB, and 6485MB for MIN, MID, and MAX respectively) as they do not account for workload characteristics. Single-threaded workloads heavily punish over-provisioning memory settings. The same workload that would cost $200 for a million invocations at around 2GB of memory would cost nearly $1,000 (400% increase) at maximum memory while providing a 4% performance increase as shown in Figure 8.

The Sysbench function which generated prime numbers enabled assessment of performance implications for a multi-threaded workload, and our results are shown in Figure 9. In contrast to the Resize function, which had a vast cost

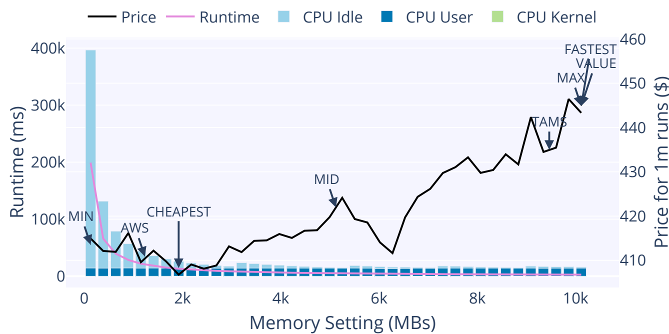

Fig. 9. Sysbench runtime, cost, CPU profile, and selected settings on AWS.

range, multi-threaded workloads with good parallelism exhibited more consistent costs across all memory settings. This reduced cost range can be attributed to the multi-threaded prime number generation's high parallelism enabling the ideal scenario where doubling memory reduces the runtime in half (i.e. perfect scaling) to provide better cost equivalence across the full range of memory settings. While the performance gain from scaling was not perfect, the cost varied less than \$40 ($\sim$10% increase) across all memory settings. For Sysbench prime number generation, the FASTEST and MAX-VALUE options were at the maximum memory (i.e. 10GB), while CPU-TAMS found a slightly lower setting (9.5GB), the AWS Compute Optimizer chose a much lower memory setting (1.3GB), and CHEAPEST was slightly higher (2GB). Since cost varies considerably in this example, selecting low memory settings results in significantly higher runtime. The CPU-TAMS and MAX memory approaches resulted in a function runtime of about 2.7 seconds per function invocation. CHEAPEST had a runtime of 13 seconds (383% slower), and the AWS-CO recommendation had a runtime of 21 seconds (711% slower). This runtime and cost disparity highlight why targeting the CHEAPEST setting may be a poor choice if high performance is needed.

The AWS Compute Optimizer (AWS-CO) attempts to minimize the cost while ignoring the implications of runtime. In contrast, our CPU-TAMS method balances cost and runtime goals. Specific optimization goals for functions can vary widely where a developer may be willing to pay significantly more for only small performance improvement if critical for the use case. For other functions, a developer may desire the absolute lowest cost. CPU-TAMS recommends memory settings after a single function invocation, while AWS-CO requires at least 50 function invocations, and brute force requires hundreds. To generate memory recommendations using CPU-TAMS, a function needs to be profiled at the MAX memory setting only once. To test the AWS Compute Optimizer, we deployed all of the functions at 1024MB, ran them 60 times, and then waited one day for AWS to make a recommendation. Even though our functions had vastly different runtime and profiles, AWS-CO recommended a 1232MB memory reservation size for all functions, except the sleep function, where it recommended 848MB.

Table V shows the impact of each memory selection method for each function, comparing the relative cost and runtime compared to MAX-VALUE. For runtime, as expected, using the MIN memory rule of thumb always resulted in the longest runtime ranging from 3 to 80x slower than MAX-VALUE while resulting in 0.5 to 1.1x the cost. Selecting MIN resulted in paying more than MAX-VALUE for only the MST function. Selecting CHEAPEST resulted in 2.6 to 5x slower performance than MAX-VALUE.

The FASTEST memory setting and MAX memory resulted in faster runtime but more cost. We observed at least double the cost for all single-threaded workloads using the MAX memory setting compared to MAX-VALUE while only offering a 3-5% performance improvement.

In summary, our model based approaches outperformed rule of thumb methods for determining function memory settings with MAX-VALUE. Like the CHEAPEST option and MIN, the AWS Compute Optimizer (AWS-CO) memory recommendations were below the MAX-VALUE setting. AWS-CO recommendations resulted in 1.4 to 8x worse performance than the MAX-VALUE option. AWS-CO recommendations supported lower costs due to its more conservative recommendations: 0.8 to 0.95x the cost compared to the MAX-VALUE memory setting. CPU-TAMS most closely selected memory settings to MAX-VALUE setting discovered using brute force, and was most effective at offering a balance between high performance and low cost. Across all functions, CPU-TAMS selected memory settings that offered 0.95 to 1.19x the performance while only costing 0.9 to 1.1x the MAX-VALUE memory setting found using brute force.

CPU-TAMS accurately predicted memory configurations that offered high performance while reducing cost. We ran functions across many different memory settings using brute force to find the MAX-VALUE memory setting, which offers the best cost to performance ratio. CPU-TAMS recommendations resulted in up to 80x faster runtime and 5x lower cost compared to configurations using the MAX and MIN memory settings, respectively. **Across our workloads, CPU-TAMS predicted a memory setting that was within 5% of the MAX-VALUE memory setting's cost and within 8% of the runtime.** CPU-TAMS only requires a single function invocation to make recommendations in contrast to brute force search methods that require extensive function profiling leading to high costs to optimize large applications or applications that frequently change.

2) CPU-TAMS on IBM Cloud Functions: As discussed earlier, IBM behaves fundamentally differently than AWS Lambda. Memory settings have no impact on performance for sequentially invoked functions. Instead, memory settings limit the number of function instances that share the 4 vCPUs of host VMs. At the maximum memory setting of 2GB, up to four function instances can share the same host VM. This means that unless more than four functions are invoked at a time, changing memory settings will have no impact on performance and only increase cost. In this case, users should always pick the MIN memory setting to have the best performance and cost. CPU-TAMS becomes a necessary tool when consider-

TABLE V
PERCENT CHANGE IN RUNTIME (LEFT) AND COST (RIGHT) FOR EACH FUNCTION USING EACH MEMORY SELECTION METHOD COMPARED TO THE MAX-VALUE MEMORY SETTING DISCOVERED USING *BRUTE FORCE TESTING ON AWS LAMBDA.

Function	Cheapest* Runtime Δ%	MIN Runtime Δ%	AWS-CO Runtime Δ%	CPU-TAMS Runtime Δ%	MID Runtime Δ%	MAX Runtime Δ%	Fastest* Runtime Δ%
Writer	23	367	50	13	−6	−4	−6
Zip	26	375	56	8	−4	−4	−6
Resize	172	1287	51	8	−7	−4	−7
DNA	50	384	50	19	7	7	0
PR	57	1355	57	−2	−11	−12	−13
MST	41	1247	41	0	−5	−7	−12
BFS	26	371	58	10	2	−2	−4
Sysbench	383	7296	711	6	92	0	0
Average	97.25	1585.25	134.25	7.75	8.5	−3.25	−6

Function	Cheapest* Price Δ%	MIN Price Δ%	AWS-CO Price Δ%	CPU-TAMS Price Δ%	MID Price Δ%	MAX Price Δ%	Fastest* Price Δ%
Writer	−10	−7	−9	−2	160	410	160
Zip	−8	−5	−7	−6	150	406	232
Resize	−9	−7	−9	−7	142	406	142
DNA	−21	−15	−21	−9	165	406	0
PR	−6	−3	−6	11	144	368	325
MST	−3	4	−3	0	185	467	341
BFS	−7	−6	−5	−5	167	419	253
Sysbench	−8	−7	−8	−2	−5	0	0
Average	−9	−5.75	−8.5	−2.5	138.5	360.25	181.625

ing workloads with higher numbers of concurrent function invocations on IBM. Compared to AWS Lambda, IBM offers a much smaller range of available memory settings. With maximum tenancy, no memory setting will provide timeshare over one vCPU. Alongside this, IBM appears to have a 'sweet spot' memory setting at 1.5GB. In all of our tests, the 1.5GB memory setting and above always had a maximum tenancy of 4, breaking the linear scaling of lower memory settings. This identifies a rule of thumb, *as long as a function does not need more than 1.5GB, the MAX memory setting ever selected on IBM should be 1.5GB for FASTEST performance*. This can be seen clearly in Figure 10 where the 1.5GB memory setting has an increase in value. This 'sweet spot' behavior and low number of available vCPUs results in CPU-TAMS always recommending 1.5GB for workloads that utilize at least 1 vCPU; which was the case for all of our functions.

In a production environment, functions on IBM should be first deployed at 1.5GB. If the function's CPU profile indicates the use of one or more vCPUs then the function should remain at 1.5GB. If not, the function should be profiled to collect data to determine the average number of concurrent invocations. If the average concurrency is never above four, change the memory setting to the lowest required memory setting. If the function uses less than 1 vCPU and has more than 4 tenants, use the model in Figure 11 to choose the lowest memory setting that provides the number of needed vCPUs for the average number of tenants.

3) CPU-TAMS on Google Cloud Functions and DigitalOcean Functions: Ideally, CPU-TAMS will have access to Linux CPU time accounting metrics from the context of the function instance. On GCF and DOF, however, these platforms abstract CPU metrics restricting the ability to profile functions on these platforms. As a workaround, we executed CPU-bound profiling functions to construct CPU models to estimate the number of allocated CPUs across a range of memory settings. DOF and IBM both implemented using OpenWhisk, exhibit similar behavior which we are able to exploit to determine how vCPUs were mapped to function memory settings. We found that host VMs on DOF had 8 vCPUs which were shared with multiple function instances for workloads with concurrent function calls. The number of vCPUs on DOF scales linearly from 0.125 vCPU at 128MB to 1 vCPU at 1024MB. While DOF did not exhibit a 'sweet spot' like IBM, having only one vCPU available to function instances with maximum tenancy resulted in 1024MB being the recommended memory setting for all of our evaluation functions.

In contrast, GCF offers the largest range of memory settings of any FaaS platform up to 16GB, and up to 4 vCPUs. Unlike other platforms, GCF does not scale CPU time share linearly across a range of memory settings. Instead, GCF has seven tiers of CPU allocations: functions at 128MB receive 0.083 vCPU, 128 to 256MB 0.166 vCPU, 256 to 512MB 0.333 vCPU, 512 to 1GB 0.5833 vCPU, 1GB to 2GB 1 vCPU, 2GB to 8GB 2 vCPUs, and above 8GB 4 vCPUs. The vCPU allocation doubles as the memory setting doubles with the exception of 4 vCPUs being allocated at 8GB. This tiered approach to vCPU allocation results in the vast majority of memory settings not offering any performance improvement as the number of allocated vCPUs only changes when elevating to a new tier. For example, a function that requires two vCPUs and little memory should adopt the memory setting that first provides two vCPUs as any higher memory setting will increase cost. The vCPU tiers greatly simplify memory selection on GCF as 128MB, 192, 320, 576, 832, 1088, 2304,

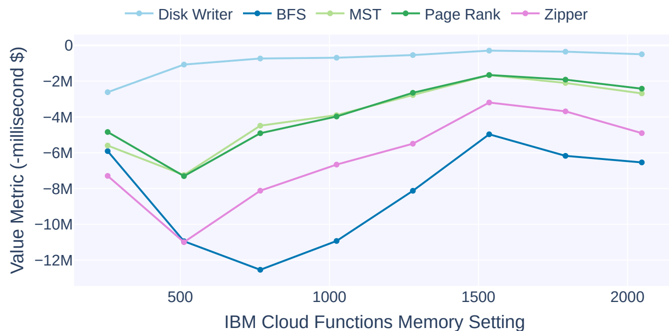

Fig. 10. IBM Cloud Functions value for different functions.

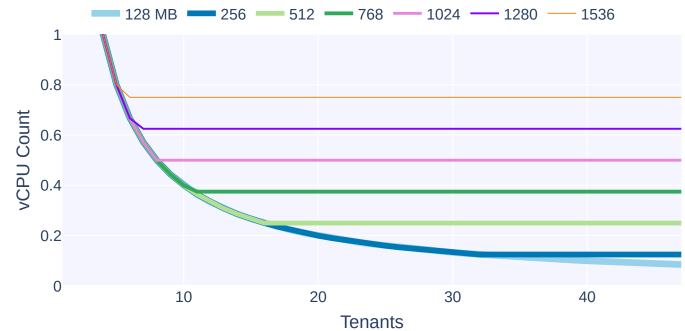

Fig. 11. IBM vCPUs at each tenancy with varying memory settings.

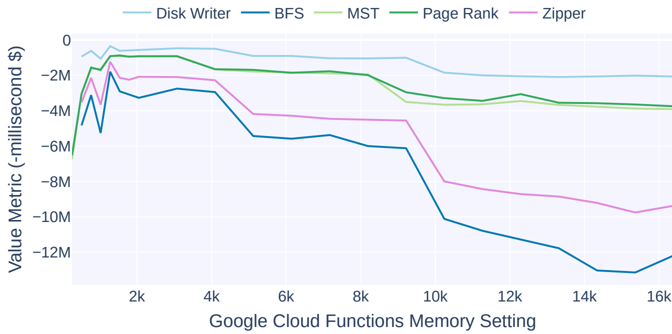

Fig. 12. Google Cloud Functions value metric for different functions.

TABLE VI
MEMORY SELECTION METHOD COMPARISON ON AWS LAMBDA

Name	Invocations	Cost	Goal	Accuracy
MIN	1	$0.002	CHEAPEST	Approximate
MID	1	$0.075	MAX-VALUE	Approximate
MAX	1	$0.15	FASTEST	Approximate
CPU-TAMS	1	$0.15	MAX-VALUE	Approximate
AWS-CO	50	$1.31	CHEAPEST	Approximate
Brute Force	n	$3.22	N/A	Optimal
Linear Search	Up to n	$0.58	MAX-VALUE	Optimal
Binary Search	$nlog(n)$	$2.28	MAX-VALUE	Optimal
Gradient Descent	Up to n	$1.50	MAX-VALUE	Optimal

Cost is derived from execution of all evaluation functions in Table III
(n: Maximum number of memory settings to test.)

and 8704 should be the only settings ever selected, as these are the lowest memory settings at each tier of vCPUs. Developers should evaluate the number of vCPUs their functions utilize and then choose the first memory setting that supplies that quantity of vCPUs for MAX-VALUE. Figure 12 depicts the MAX-VALUE of five functions across a range of memory settings. Here the functions all utilize one vCPU and we see at 1088MB, the memory setting that first offers one vCPU, offers MAX-VALUE. As new vCPU tiers are crossed, drops are observed in value caused by the cost of doubling vCPUs at each tier as GCF bills by memory and vCPU allocation.

4) CPU-TAMS on Azure Functions, OpenFaaS, and More: In addition to these platforms, we also investigated implementing CPU-TAMS on Azure Functions and self-hosted OpenFaaS clusters. The core design of both of these platforms makes memory prediction methods unneeded. On Azure Functions, all functions are allocated 1.5GB of RAM and are billed based on the amount used. Consumers do not have the option to change the amount of memory allocated to their functions. OpenFaaS also does not have configurable memory settings that scale performance in the same manner as other FaaS platforms. OpenFaaS allocates shared containers that host multiple function instances [38]. Memory limits apply to all function calls that share a container, if multiple function invocations sharing a container exceed the OpenFaaS memory or CPU limit, then function invocations begin to fail.

When FaaS platforms change their vCPU to memory mapping, or to add CPU-TAMS support for a new platform, a new vCPU-to-memory model should be trained for the platform.

C. Memory Selection Method Comparison

On AWS Lambda, CPU-TAMS was able to determine MAX-VALUE memory settings within 8% runtime mean absolute percent error (MAPE) and 5% cost MAPE of MAX-VALUE memory settings found using brute force. On GCF, IBM, and DOF, CPU-TAMS was able to always find MAX-VALUE function memory settings discovered using brute force testing. This was possible by leveraging distinct characteristics of each platform's vCPU-to-memory scaling policy, such as the 'sweet spot' on IBM, the tiered approach on GCF, and the reduced range of memory settings on DOF to simplify the challenge of function memory optimization on these platforms.

CPU-TAMS exhibits lower costs compared to other memory selection techniques as the function is profiled only once at the maximum platform memory setting. Consider the worst-case FaaS function on AWS Lambda with maximum runtime (e.g. 15 minutes @ 10GB). The CPU-TAMS profiling cost is just 15 cents. Most functions will not be worst-case and will have shorter runtimes and lower profiling costs. If a developer knows the maximum number of threads required they can reduce cost by profiling their function at a lower memory setting. The worst case scenario for the AWS Compute Optimizer is nearly 10x more costly at $1.31 per function. This is the cost of 50 function calls with 15 minutes runtime at 1792MB (the maximum supported by AWS-CO). AWS-CO is more expensive due to the larger number of function invocations required. This cost disadvantage applies to search methods which require hundreds or more function calls to generate recommendations. **The cost to find MAX-VALUE memory settings for all 12 of our functions, was $3.22 for brute force, $0.10 for the AWS-CO, and $0.008 for CPU-TAMS.**

To evaluate the linear search, binary search, and gradient descent algorithms we utilized the brute force search dataset walking through the data to emulate each search algorithm on AWS Lambda. Linear search had the lowest average cost of $0.58, likely attributed to most functions having a low MAX-VALUE memory setting, in contrast to functions such as Sysbench that required linear search to traverse the entire memory setting range. Binary search also performed poorly, costing on average the most of any search technique at $2.28. The Binary Search algorithm always required many steps to repeatedly divide the memory range in half. Gradient descent performed slightly better, costing $1.50. **Compared to the worst case cost example for CPU-TAMS, other techniques exhibited between 3.8 to 15x higher cost.** Figure VI shows the number of required invocations, cost, goal memory setting, and whether each memory selection method results in a approximate recommendation or optimal selections.

V. CONCLUSIONS

This paper has introduced our novel CPU-TAMS memory selection method and presented results of our evaluation to find recommended memory configurations across multiple serverless FaaS platforms. **RQ-1:** To evaluate CPU-TAMS we leveraged workloads with diverse CPU, disk, and network utilization to understand how resource performance scales with function memory across platforms. On AWS Lambda, disk read and network throughput scaled with function memory

up to 2GB, while CPU timeshare scaled linearly across all memory settings. Google Cloud Functions allocated vCPUs in a tiered approach doubling vCPU capacity for every doubling of memory where vCPU capacity remained constant for each tier. IBM Cloud Functions and DigitalOcean functions did not couple resource shares to memory settings, but instead allowed co-located functions to compete for resources. **RQ-2:** CPU-TAMS was shown to accurately predict memory settings to provide MAX-VALUE recommendations that offer high performance and low cost. Using a single profiling run, CPU-TAMS predicted MAX-VALUE function memory configurations with only 5% cost, and 8% runtime error on AWS Lambda. CPU-TAMS found MAX-VALUE memory settings on Google Cloud Functions, IBM Cloud Functions, and DigitalOcean functions with no error by leveraging distinct characteristics of each platform's vCPU-to-memory scaling policy. Compared to other search techniques, CPU-TAMS required significantly less profiling data, resulting in 3.8 to 15x lower cost to apply. Each FaaS platform provided unique challenges from different platform scaling policies that CPU-TAMS had to account for. Our efforts demonstrate that a one-size-fits-all approach to find optimal FaaS function memory configurations for every platform is not possible as accounting for platform heterogeneity is required.

ACKNOWLEDGMENTS

This research is supported by the NSF Advanced Cyber-infrastructure Research Program (OAC-1849970), NIH grant R01GM126019, and AWS Cloud Credits for Research.

REFERENCES

[1] AWS, "AWS Lambda – Serverless Compute - Amazon Web Services," http://aws.amazon.com/lambda/, 2021.
[2] Microsoft Azure, "Azure Functions," http://azure.microsoft.com/en-us/services/functions/s, 2021.
[3] IBM, "IBM Cloud Functions," http://ibm.com/cloud/functions, 2021.
[4] Google Cloud, "Google Cloud Function: Event-Driven Serverless Compute Platform," http://cloud.google.com/functions, 2021.
[5] DigitalOcean, "DigitalOcean —Run functions on demand." https://www.digitalocean.com/products/functions, 2022.
[6] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," *2018 USENIX Annual Technical Conf. (USENIX ATC 18)*, 2018.
[7] Datadog, "The state of serverless," Feb 2020. [Online]. Available: http://datadoghq.com/state-of-serverless-2020/
[8] R. Cordingly, H. Yu, D. P. Varik Hoang, D. Foster, Z. Sadeghi, R. Hatchett, and W. J. Lloyd, "Implications of Programming Language Selection for Serverless Data Processing Pipelines," in *2020 6th IEEE Int. Conf. on Cloud and Big Data Computing (CBDCOM 2020)*, 2020.
[9] A. Eivy and J. Weinman, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.
[10] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: the prospect of serverless scientific computing and HPC," in *Comm in Computer and Information Science*, 2018.
[11] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," 2017.
[12] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.
[13] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018.
[14] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Proc - 2018 IEEE Int. Conf. on Cloud Engineering, IC2E 2018*, 2018.
[15] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services," in *2019 IEEE Int. Conf. on Cloud Engineering (IC2E)*, jun 2019, pp. 23–33.
[16] M. Fotouhi, D. Chen, and W. J. Lloyd, "Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application," in *Proc of the 5th Int. Workshop on Serverless Computing*, 2019, pp. 49–54.
[17] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring Serverless Computing for Neural Network Training," in *IEEE Int. Conf. on Cloud Computing, CLOUD*, 2018.
[18] R. Cordingly, N. Heydari, H. Yu, V. Hoang, Z. Sadeghi, and W. Lloyd, "Enhancing observability of serverless computing with the serverless application analytics framework," in *Companion of the 2021 ACM/SPEC Int. Conf. on Performance Engineering, Tutorial*, 2021.
[19] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video processing with serverless computing: A measurement study," in *Proc of the 29th ACM workshop on network and operating systems support for digital audio and video*, 2019, pp. 61–66.
[20] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," *arXiv preprint arXiv:2012.14132*, 2020.
[21] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th Int. Conf. on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.
[22] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proc of the 11th ACM Symp. on Cloud Computing*, 2020, pp. 30–44.
[23] A. Casalboni, "Aws lambda power tuning," 2020. [Online]. Available: http://github.com/alexcasalboni/aws-lambda-power-tuning
[24] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," *arXiv preprint arXiv:2010.15162*, 2020.
[25] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM 2020-IEEE Conf. on Computer Comm.* IEEE, 2020, pp. 129–138.
[26] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting Performance and Cost of Serverless Computing Functions with SAAF," in *6th IEEE Int. Conf. on Cloud and Big Data Computing (CBDCOM 2020)*, 2020.
[27] R. Cordingly, "Serverless performance modeling with cpu time accounting and the serverless application analytics framework," 2021.
[28] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Trans on Cloud Computing*, 2020.
[29] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proc of the ACM/SPEC Int. Conf. on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–276.
[30] T. Zubko, A. Jindal, M. Chadha, and M. Gerndt, "Maff: Self-adaptive memory optimization for serverless functions," in *European Conf on Service-Oriented and Cloud Computing*. Springer, 2022, pp. 137–154.
[31] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "Slam: Slo-aware memory optimization for serverless applications," *arXiv preprint arXiv:2207.06183*, 2022.
[32] W. J. Lloyd, S. Pallickara, O. David, M. Arabi, T. Wible, J. Ditty, and K. Rojas, "Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives," *IEEE Trans on Cloud Computing*, 2015.
[33] "Stress(1)," 2012. [Online]. Available: http://linux.die.net/man/1/stress
[34] A. W. Services, "Aws compute optimizer," 2021. [Online]. Available: http://aws.amazon.com/compute-optimizer/
[35] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, and W. Lloyd, "The serverless application analytics framework: Enabling design trade-off evaluation for serverless software," in *Proc of the 2020 Sixth Int. Workshop on Serverless Computing*, 2020, pp. 67–72.
[36] "Apache OpenWhisk," http://openwhisk.apache.org, 2021.
[37] Fn Project, "Fn Project – The Container Native Serverless Framework," http://fnproject.io/, 2021.
[38] N. Mahmoudi and H. Khazaei, "Performance modeling of metric-based serverless computing platforms," *IEEE Transactions on Cloud Computing*, 2022.