Serverless Performance Modeling with CPU Time Accounting and the Serverless Application Analytics Framework

Robert Cordingly

A thesis
submitted in partial fulfillment of the
requires for the degree of

Master of Computer Science and Systems

University of Washington

2021

Committee:

Wes J. Lloyd

Ka Yee Yeung

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

Serverless Performance Modeling with CPU Time Accounting and the Serverless Application Analytics Framework

Robert Cordingly

Chair of the Supervisory Committee:
Assistant Professor Wes J. Lloyd
School of Engineering and Technology

Recently serverless computing platforms have emerged to provide appealing options to developers for deploying cloud native applications. One of the most popular serverless computing delivery paradigms known as Function-as-a-Service (FaaS) offers many desirable features including high availability, fault tolerance, and automatic application scaling. Understanding performance of FaaS workloads in the cloud involves new challenges as FaaS workloads are billed to the nearest millisecond, infrastructure is temporary, and observability of hardware configuration, load balancing, and function tenancy is notoriously obscure.

To better understand FaaS workload performance to enable accurate performance predictions, we propose a novel performance modeling approach that leverages Linux CPU Time Accounting (CPU-TA). We introduce the Serverless Application Analytics Framework (SAAF), our tool used to collect information about the infrastructure running FaaS platforms, and use this to train models to predict FaaS workload runtime. Using 10 different serverless applications, we compare our CPU-TA modeling approach to baseline performance modeling approaches that directly predict runtime. Enabling accurate performance predictions on FaaS platforms allows developers and researches to make informed deployment decisions resulting in faster performance and reduced hosting cost.

We found our runtime prediction technique was able to make accurate performance predictions across 448 different FaaS configuration scenarios, achieving average prediction accuracy of over 95%.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

Serverless computing recently has emerged as a compelling approach for hosting applications in the cloud [1] [2] [3]. Serverless computing platforms promise autonomous fine-grained scaling of computational resources, high availability (24/7), fault tolerance, and billing only for actual compute time while requiring minimal setup and configuration. To realize these capabilities, serverless platforms leverage ephemeral infrastructure such as MicroVMs or application containers. The serverless architectural paradigm shift ultimately promises better server utilization as cloud providers can more easily consolidate user workloads to occupy available capacity, while deallocating unused servers, to ultimately save energy [4] [5] [6]. Rearchitecting applications for the serverless model promises reduced hosting costs as fine-grained resources are provisioned on demand and charges reflect only actual compute time.

Function-as-a-Service (FaaS) platforms leverage serverless infrastructure to deploy, host, and scale resources on demand for individual functions known as "microservices" [7]. With FaaS platforms, applications are decomposed and hosted using collections of independent microservices differing from application hosting with Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) cloud platforms. On FaaS platforms, temporary infrastructure containing user code plus dependent libraries are created and managed to provide granular infrastructure for each service [8]. Cloud providers create, destroy, and load balance service requests across available server resources. Users are billed based on the total number of function invocations, memory utilization, and total runtime using fine-grained measurements as small as milliseconds on some platforms [9] [10]. Serverless platforms have arisen to support highly scalable, event-driven applications consisting of short-running, stateless functions triggered by events generated from middleware, sensors, microservices, or users [11].

Common use cases include: multimedia processing, data processing pipelines, IoT data collection, chatbots, short batch jobs/scheduled tasks, REST APIs, mobile backends, and continuous integration pipelines.

Despite the many advantages with serverless computing, several important challenges have not been addressed [12] [13]. FaaS platform complexities including heterogeneous CPUs, variable function tenancy, the coupling of memory setting to CPU performance, and microservice composition lead to considerable *performance variation* of application performance. All of these factors lead to unpredictable performance and cost of application deployments. Unlike IaaS clouds, where cost accounting is as simple as tracking the number of VM instances and their uptime, serverless billing models are *multi-dimensional*. Software deployments consist of many microservices which must be individually tracked [14]. FaaS platforms exhibit *performance variation* and this directly translates to cost variability. Functions can execute across *heterogeneous CPUs* on servers that host a variable number of co-located function instances referred to as *function tenancy* producing *resource contention*. Understanding hosting costs for FaaS applications requires understanding the implications of *microservice composition*, where applications composed of multiple functions are deployed and scaled separately.

The coupling of cost to runtime on FaaS platforms motivates developers to optimize their applications to reduce cost. FaaS platforms presently lack effective tools to support estimating the costs of hosting applications. Current cloud pricing calculators from public cloud providers (e.g. AWS and Azure), and commercial tools (e.g. Intel Cloud Finder, RankCloudz, Cloudorado) primarily provide IaaS compute and storage cost estimates based on average performance [15] [16] [17]. Recently, FaaS calculators have appeared, but they are limited to generating cost estimates based on average runtime and memory size [18] [19] [20]. These calculators do not consider how FaaS function runtime scales relative to the memory reservation size, a feature coupled to CPU power on several FaaS platforms [9] [21], how FaaS platforms utilize heterogeneous CPUs, or how factors such as multi-tenancy, impact performance.

To understand FaaS performance variation and optimize applications to reduce hosting costs, developers must profile their applications and estimate runtime for different FaaS

configurations under different system conditions. To make these predictions, developers need a tool to collect information about their functions and a technique to train runtime prediction models built around serverless platforms. As every millisecond of runtime has a cost on FaaS platforms, developers need to know the minimum amount of training data they need to collect, what features provide the most accurate predictions.

In this thesis, to address the challenges associated with pricing obfuscation on serverless platforms, we focus on the problem of modeling runtime of FaaS functions. Runtime predictions can then be directly translated into cost estimates by applying the platform's pricing policy. We offer a novel approach combining CPU time accounting, multiple regression, and random forest regression to provide highly accurate FaaS function runtime predictions. Our approach involves profiling CPU metrics of multiple FaaS function deployments run using different configurations and hardware (e.g. AWS Lambda with 256, 512, 1024 MB to Intel Xeon E5-2680v2, E5-2676v3, E5-2686v4). We train regression models to predict how CPU metrics (e.g. CPU user mode time, CPU kernel mode time) scale across alternate function deployments with different CPUs and memory settings, and even to different cloud providers. By applying Linux CPU time accounting principles, we can then calculate FaaS function runtime on any CPUs (e.g. Intel Xeon E5-2686v4), with any memory size (e.g. 1024 MB), on any cloud (e.g. IBM Cloud Functions [22]). We implement our approach using both multiple linear regression and random forest regression to train models and compare our predictions against baseline methods that predict function runtime directly. We evaluate our approach using compute bound functions, five different workloads, and 448 different FaaS configurations. Predictions are evaluated across different CPUs, memory configurations, and with and without function multi-tenancy on different FaaS platforms.

We found workload runtime could be estimated within ~4.9% mean absolute percentage error (MAPE) or better by our CPU Time Accounting approach across all modeling scenarios. Our approach can help developers predict FaaS workload costs to make informed deployment decisions. These advancements can enable developers to better evaluate deployment and design alternatives, while understanding performance implications to achieve more efficient serverless software implementations.

1.2 Research Questions

This thesis investigates the following research questions:

RQ-1: (Performance Variation Evaluation) To what extent do factors such as heterogeneous CPUs or function instances multi-tenancy, contribute most to performance variation on FaaS platforms?

RQ-2: (CPU Time Accounting Performance Modeling) How effective is our CPU time accounting performance modeling approach in yielding accurate runtime estimates of workloads across different FaaS configurations (e.g. memory settings or heterogeneous CPUs)? What is the accuracy compared with runtime predictions derived from baseline performance modeling approaches?

RQ-3: (Training Data Size) How accurately can we predict the runtime of FaaS workloads with limited profiling data? What are the tradeoffs between prediction accuracy vs. the size of available training data?

RQ-4: (Feature Selection) How effective are the individual metrics collected by SAAF (e.g. CPU metrics, page faults, context switches) as features to support predicting FaaS function runtime?

RQ-5: (Workload Evaluation) How accurately can we predict function runtime using our CPU time accounting across a variety of different workloads? How do workload characteristics such as CPU utilization, disk utilization, and parallelism influence predictability of runtime?

1.3 Thesis Contributions

This thesis makes the following research contributions:

1. We introduce and utilize the Serverless Application Analytics Framework (SAAF), a

profiling framework designed for FaaS cloud computing platforms. SAAF collects a variety of metrics about the infrastructure used to host serverless applications and how applications perform.

2. We evaluate the consistency of performance on FaaS platforms. We build workloads to measure the performance impact of unique characteristics of FaaS platforms such as CPU hardware heterogeneity and function instance multi-tenancy. By understanding aspects of platforms that lead to high performance variation, we can adapt our models to better account for them.

3. We describe our Linux CPU Time Accounting approach that leverages CPU metrics to predict FaaS function runtime for workloads deployed with different configurations (e.g. memory or CPU). We then present results from our investigation on the accuracy of our Linux CPU Time Accounting modeling approach compared with baseline modeling approaches that leverage multiple regression and random forest to directly predict runtime.

4. We investigate the accuracy vs. training data size tradeoff for FaaS performance modeling. Specifically we are interested in understanding the additional accuracy afforded by collecting additional model training data on FaaS platforms. By understanding the accuracy vs. training data size tradeoff, we can better direct profiling efforts to save costs when collecting performance model training data.

5. We investigate the utility of available features from SAAF to support FaaS function runtime predictions. We are interested in learning which features provide the most predictive value in performance models. Our goal is to reduce profiling overhead by not assessing features that do not contribute towards making better function runtime predictions.

6. We deploy a suite of ten functions to AWS Lambda, to evaluate our CPU Time Accounting approach for a variety of different workloads. We investigate the impact

of specific workload types (e.g. CPU or disk bound workloads) and application features (e.g. multithreading) on workload predictability.

Chapter 2

BACKGROUND

The challenge of performance prediction on serverless platforms, including the need to address performance variation resulting from hardware heterogeneity is identified in [12]. The authors identify how pay-as-you-go pricing models, and the complexity of serverless application deployments, leads to the key pitfall: "Serverless computing can have unpredictable costs". In contrast to application hosting with VMs, serverless platforms complicate budgeting as organizations must predict service utilization to estimate hosting costs. Performance variation of serverless workloads and accuracy of runtime predictions are invariably linked. We review related work on cloud performance variation, performance modeling, and performance evaluation of serverless platforms highlighting relationships to our research goals.

2.1 Performance Variation of Cloud Systems

In the public cloud, key factors often responsible for producing performance variation include hardware heterogeneity, provisioning variation, and resource contention. Ou and Farley identified the existence of heterogeneous CPUs that host identically labeled VM types on Amazon EC2, leading to IaaS cloud performance variation [23] [24]. Rehman et al. identified the problem of "provisioning variation" in IaaS clouds in [25]. Provisioning variation is the random nature of VM placements that generates varying multi-tenancy across physical servers producing performance variation from differences in resource contention. Schad et al. showed the unpredictability of Amazon EC2 VM performance resulting from provisioning variation and resource contention from VM multi-tenancy in [26]. Ayodele et al. and Lloyd et al. demonstrated how resource contention from multi-tenant VMs can be identified using the cpuSteal metric in [27] [28].

On serverless FaaS platforms Jonas et al. identified heterogeneous CPUs and noted their

potential to complicate performance modeling in [12]. Wang et al. identified heterogeneous VM types on FaaS platforms from AWS, Azure, and Google in [5]. They observed 4 CPU types and 5 VM configurations (AWS Lambda), 3 CPU types x 3 VM configurations (Azure functions), and 4 CPU types (Google Cloud Functions). Their efforts did not evaluate the extent of performance variation possible from heterogeneous CPUs.

Previous research has identified how provisioning variation results in varying degrees of function tenancy on FaaS platforms [4] [5] [4]. We identified how the number of function "tenants" on VMs, called "function instances" by Wang, increased when scaling up the number of concurrent requests on AWS Lambda [4]. Conversely, increasing function memory reduced the number of tenants on a VM. Wang observed that function instance placement across VMs on AWS Lambda used greedy placement, where concurrent requests are packed onto individual VMs until available memory (3328MB) is exhausted. Multiple functions from a single user account were found to share VMs, but VMs did not appear to be shared with other users. On Azure Functions, the maximum observed tenancy of function executions did not exceed 8, while up to 4 user accounts shared VMs. While these efforts identified the multi-tenancy, they did not evaluate performance implications from resource contention.

2.2 Performance Modeling of Cloud Systems

On IaaS clouds, domain specific approaches have been developed to model workload performance by incorporating specific metadata regarding the tasks [29] [30] [31] [32]. Recently, offline and online machine learning approaches have been applied to model runtime of multi-stage, batch-oriented, scientific workflows. The use of task metadata and resource utilization metrics as features supported accuracy improvements [33] [34] [35]. Other efforts at IaaS cloud performance and cost modeling have focused on cost-aware VM scheduling to support infrastructure management for VM placement [36] [37] [38] [39]. Efforts to save costs by leveraging reduced-priced cloud VMs available through auction based pricing mechanisms, such as Amazon EC2 spot instances, have spurred considerable research [39] [40] [41]. In summary, existing approaches have focused on runtime predictions for batch-oriented workloads that execute across homogeneous cloud VMs. While other efforts have focused on

performance modeling for resource management, to optimize the use of auction based VMs, or to help select an appropriate VM type.

2.3 Performance Modeling of Serverless Platforms

A significant amount of prior research on serverless platforms has focused on evaluating performance of FaaS platforms for hosting a variety of workloads and not performance modeling. Several efforts have investigated performance implications for hosting scientific computing workflows [42] [43] [44] [45]. Other efforts have evaluated FaaS performance for machine learning inferencing [46] [47], NLP inferencing [48], and even neural network training [49]. To support cost comparison of serverless computing vs. IaaS cloud, Boza et al. developed CloudCal, a tool to estimate hosting costs for service-oriented workloads on IaaS (reserved), IaaS (On Demand), and FaaS platforms [50]. CloudCal determines the minimum number of VMs to maintain a specified average request latency to compare hosting costs to FaaS deployments. FaaS resources, however, were assumed to provide identical performance as IaaS VMs when functions were allocated 128 MB RAM. Wang et al. identified AWS Lambda performance at 128 MB as only ~1/10th of 1-core VM performance in [5] which more recently is estimated to be ~1/12th of 1-core performance [51] suggesting inaccuracies with CloudCal. Other efforts have conducted case studies comparing costs of hosting application workloads on IaaS vs. FaaS [4] [52], and FaaS vs. PaaS [53].

More recently, a few efforts have focused on predicting FaaS function performance. In [54], the authors proposed techniques to model performance of serverless platforms focusing on optimization of infrastructure state (cold vs warm) and the infrastructure lifecycle of serverless platforms. In [55], the authors predicted performance of serverless workflows using mixture density networks and continuous model learning to predict FaaS round trip time. In this thesis, we extend our evaluation of our novel CPU Time Accounting approach to predict runtime of serverless functions initially described in [56].

Chapter 3

METHODOLOGY

In this section, we detail tools and techniques used to investigate our research questions (RQ-1, RQ-2, RQ-3, RQ-4, RQ-5). Section 3.1 describes our suite of serverless profiling tools including the Serverless Application Analytics Framework (SAAF) and FaaS Runner. Section 3.2 describes our performance prediction techniques, including Linux CPU time accounting principles and how multiple linear regression and random forest-based regression are leveraged to make predictions. Section 3.3 describes our experimental workloads, applications, and data sets.

3.1 Supporting Tools

To enable a better understanding of the performance implications for serverless Function-as-a-Service platforms we developed the Serverless Application Analytics Framework (SAAF) [57] [58]. SAAF is a multi-language framework which is included in the deployment packages of functions deployed to multiple commercial FaaS platforms. SAAF supports analysis of functions deployed to AWS Lambda, Google Cloud Functions, Azure Cloud Functions, IBM Cloud Functions, and the OpenFaaS FaaS platforms [9] [10] [59] [60]. SAAF provides native versions to enable analysis of FaaS functions written in Java, Node.js, Python, Go, and BASH on AWS Lambda custom runtimes.

SAAF collects metrics from multiple sources inside the Linux operating system including the /proc filesystem, local files in /tmp, and environment variables created by the FaaS platform. These metrics are then aggregated and appended onto the JSON data returned by the function. SAAF's design allows all of these 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 commercial FaaS platform (e.g. AWS Lambda, IBM Cloud Functions) exposes or hides different metadata about the underlying Linux

environments that run functions. SAAF is built specifically for FaaS platforms so it is able to run on all supported platforms and collect as much information as each FaaS platform provides.

To determine function tenancy and potential resource contention, SAAF supports uniquely identifying VMs that host one or more function instances by implementing platform specific mechanisms. IBM Cloud Functions runs Xen 4.7 allowing the unique Xen hypervisor ID that is available from /sys/hypervisor/uuid [61] to be used as a method of VM identification. VMs can be uniquely identified on AWS Lambda with the sandbox-root ID in /proc/$$/cgroup [5].

To automate complex experiments on FaaS platforms, we created a secondary application called FaaS Runner. FaaS Runner provides a client-side application that is used in conjunction with SAAF. FaaS Runner automates FaaS experiments by using functions and experiment files that define how FaaS functions should be executed and how the results from SAAF should be processed. Experiment files define how functions should be executed; synchronously or asynchronously, in parallel across many threads or sequentially, which Payloads should be used and how to distribute them, or whether multiple functions in a pipeline should be executed in a certain order. FaaS Runner supports executing functions, and automatically changing function configurations on all of SAAF's supported platforms. After an experiment is completed, FaaS Runner will automatically aggregate results and compile a report for quick data analysis, or allow the data to be easily imported into another tool such as R, a Jupyter Notebook, or a spreadsheet. Figure 3.1 shows the workflow of defining an experiment file, FaaS Runner executing functions across multiple platforms, and compiling the results.

Finally, we include many other helpful tools for developing serverless applications. Functions created using the SAAF project structure can be automatically packaged and deployed to all supported FaaS platforms using our publish scripts. This enables applications to be created that are entirely platform neutral, a single code base can be written and automatically deployed to all of SAAFs supported platforms without any code changes. Both SAAF and FaaS Runner are invaluable tools for scientists and practitioners to profile applications and execute experiments on FaaS platforms.

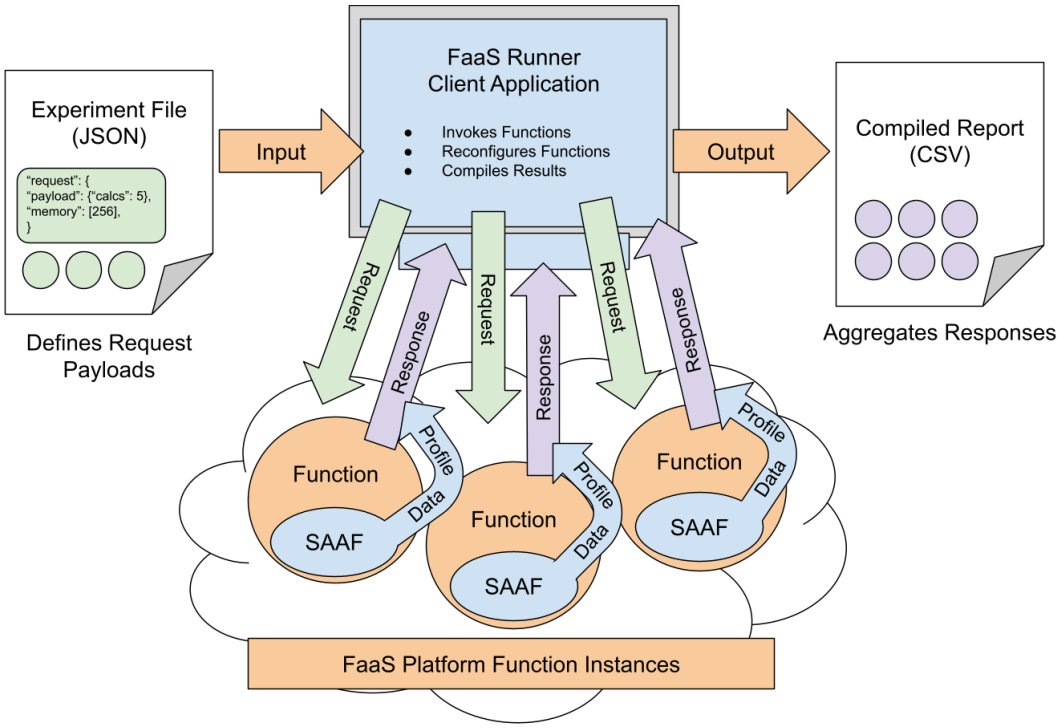


Figure 3.1: Function-as-a-Service experiment execution with FaaS runner and profiling with SAAF. SAAF profiles function instances and appends profiling data to the response payload

3.2 CPU Time Performance Profiling

In this thesis, we extend our previous work with Linux CPU time accounting performance modeling [62] [56]. In particular, we compare accuracy with other performance prediction methods on serverless platforms while discussing improvements and challenges. In contrast to traditional performance modeling techniques described in chapter 2, our approach leverages CPU time accounting metrics to train multiple models to predict each CPU time accounting metric rather than directly predicting runtime. The Linux /proc filesystem provides many metrics that detail the time the CPU spends executing in different modes. The SAAF framework returns these metrics, such as time spent in user mode (CpuUsr), kernel mode (CpuKrn), idle (CpuIdle), waiting for I/O (CpuIOWait), or serving interrupts (CpuIntSrvc and). After training model to predict individual metrics, the total wall clock runtime of a workload can be calculated by summing all of the metrics and dividing by the number of CPU cores of the host server:

$$Runtime = \frac{\begin{array}{c} cpuUsr + cpuKrn + cpuIdle + cpuIOWait + \\ cpuIntSrvc + cpuSoftIntSrvc \end{array}}{\#\ of\ CPU\ cores}$$

In previous work with Linux CPU time accounting performance modeling, multiple linear regression was used to make runtime predictions using only CPU bound workloads. In this thesis, we extend the previous work by investigating the efficacy of using stepwise multiple regression and random forest regression to predict a CPU profile to estimate runtime. Now that our predictions have expanded beyond CPU bound workloads, the number of time accounting metrics we need to predict has increased. Previously, it was sufficient to only account for CPU User and idle time in our predictions. Now our CPU Time Accounting prediction approach incorporates CPU kernel time and CPU I/O wait time alongside using many more metrics returned by SAAF as features in each of our models. This introduces new challenges as some metrics, such as CPU Kernel time, may not be normally distributed or have nonlinear relationships to the input data. Since the input data usually determines the runtime of a function, and runtime is primarily user and idle time, our previous predictions were not impacted by these challenges. Our new workloads are more complex and need models that take into account more features as our original prediction models would not

result in accurate predictions using only CPU user and idle time. If a function had 20% of its runtime as Kernel time, our original prediction method would be off by more than 20%. The need to build more complex models that take into account many feature of a workload's profile required our CPU Time Accounting approach to expand into utilizing more methods of feature selection and regression. We compare our CPU time accounting approaches with baseline approaches including standard multiple regression and random forest regression that directly predict runtime. We perform these evaluations by harnessing different workloads and different commercial FaaS platforms.

FaaS platforms present unique challenges when predicting performance with Linux CPU time accounting metrics. Depending on the memory configuration of a FaaS function, the cloud provider often adjusts the number of CPU cores and timeshare the function will receive. The virtualization hypervisor used to host function instances may behave differently. For example, the Firecracker microVM completely isolates CPU time accounting metrics from each function instance on AWS Lambda, whereas on IBM Cloud Functions the CPU timeshare is not adjusted based on memory. As a result, Linux CPU time accounting metrics need to be normalized by the number of function instances that share the host prior to training models. Alongside that, FaaS platforms can sometimes exhibit inconsistent performance. Function invocations with identical workloads may be significantly faster or slower with no clear reason why. We created workloads with increasing complexity to measure the performance variation of FaaS platforms, and create models to predict performance of different workloads.

3.3 Experimental Workloads and Datasets

To evaluate our performance modeling techniques, we developed a compute-bound function known as the "Calcs Service" (https://github.com/wlloyduw/CalcsService). This microservice produces workloads where a variable number of calculations are performed using the formula ($a \times b \div c$) with operands stored in separate arrays on the heap. For each calculation, a random index in the array is chosen to store random numbers for use as operands. Using large arrays to store operands, in contrast to primitive local variables, induces memory stress on the workload by causing page faults. To vary the degree of memory stress,

the size of the arrays are specified in the function's payload. The Calcs function was used in our workloads to perform between 30,000,000 to 60,000,000 calculations to provide a variety of function runtimes to support training performance models. To ensure deterministic behavior, we used a fixed seed across all function invocations to ensure that "random" numbers led to identical workloads. A child thread was added to the function to create a multi-threaded workload where the child thread performed 1/2 the number of calcs to ensure finishing before the parent. The second thread generates CPU contention while the parent thread dictates the function's overall runtime.

Using the Calcs Service, we profiled alternative function configurations (e.g. CPU, memory, platform) using multiple workloads described in II and III below. Each subsequent workload is designed to add additional complexity and memory stress.

Each of our workloads were executed on different versions of AWS Lambda to produce two primary datasets: an older dataset from when AWS Lambda used Xen VMs to host functions, and a recent dataset where AWS Lambda has adopted Firecracker to host functions using microVMs. We also executed our workloads on IBM Cloud Functions. We had developed our Calcs Service initially in Java and at the time of our initial testing, neither Google Cloud Functions nor Azure Functions supported Java functions using Linux. Due to this limitation, we focused our testing on AWS Lambda and IBM Cloud Functions.

We trained regression and random forest models with over 70,000 function invocations to convert individual CPU metrics from one FaaS configuration scenario to another. Both AWS Lambda and IBM Cloud Functions provide the option to configure a memory setting that impacts the performance of functions. On AWS Lambda, this setting directly changes the CPU time share allocated to functions. On IBM Cloud Functions the memory setting alters the maximum number of function instances that can share the same host; higher memory settings result in less resource contention which can result in better performance. Given that a function's memory reservation size directly impacts the performance and cost of a FaaS workload, we created regression models to predict the runtime of one memory setting based off of another. We created models to predict runtime using 256MBs, 512MBs, 1GB, and 2GBs on both AWS Lambda and IBM Cloud Functions.

In addition to the Calcs Service, we deployed nine additional functions to AWS Lambda

to evaluate predictions across different memory settings. These functions were deployed recently so we do not have data from when AWS Lambda was based on the Xen hypervisor to compare with. Consequently we cannot perform CPU to CPU configuration predictions as AWS Lambda no longer exposes the identity of the host CPU under Firecracker. These additional functions provide a variety of CPU utilization profiles. Of the nine new functions, four process scientific workloads. The MST, BFS, and Page Rank functions generate graphs and process minimum spanning tree, breadth first search, and page rank respectively. The DNA function downloads a DNA sequence file from Amazon S3 and processes visualization data. Two of our functions stress the storage volume of function instances. The Writer function generates data in memory and repeatedly saves the data to disk and deletes it for a given number of iterations. Similarly, our Compress function generates files to fill the temporary volume and then repeatedly compresses them into a Zip archive. Alongside our CPU and disk bound functions, the Resize function is primarily a network bound function. Resize downloads an image from S3, scales the image to a variety of different resolutions, and then uploads the images back to S3. The Sleep function sleeps for a given amount of time and does nothing. Finally, we deployed a wrapper function to execute Sysbench on AWS Lambda. Sysbench is a popular Linux benchmark application. We leveraged Sysbench's prime number generation benchmark to model performance. Unlike all of our other functions which only utilize two or less threads, Sysbench utilizes all of the vCPUs allocated to function instances. All of the functions we used are defined in table 3.2. Five of these functions were derived from the publicly available SeBS: Serverless Benchmarking Suite [63] with minimal changes to implement SAAF into them. By using these functions, we are able to further evaluate our Linux Time Accounting approach on a variety of different workloads.

Another feature of FaaS platforms that impacts performance is the use of heterogeneous CPUs. Prior to AWS Lambda's implementation of the Firecracker microVM, we observed that AWS Lambda leveraged at least three different CPUs to execute function instances. CPUs were randomly assigned to function instances so the user had no control over which CPU they were assigned. This same CPU heterogeneity can be still observed on IBM Cloud Functions. Presently, Firecracker appears to execute functions with homogeneous CPUs

Table 3.1: Calcs Service Function Workload Names and Definitions

Name	Description
NMT2	Fixed # of Calcs, **N**o **M**emory Stress, **2** **T**hreads, Concurrent calls
NMT2-seq	Fixed # of Calcs, **N**o **M**emory Stress, **2** **T**hreads, **Seq**uential calls
SCNMT2	**S**caling **C**alcs, **N**o **M**emory Stress, **2** **T**hreads, Concurrent calls
SCMT2	**S**caling **C**alcs, **M**emory Stress, **2** **T**hreads, Concurrent calls
SCSMT2	**S**caling **C**alcs, **S**caling **M**emory Stress, **2** **T**hreads, Concurrent calls

Table 3.2: Function Names and Descriptions

Name	Description
Calcs Service	A function that executes random math using arrays as memory stress.
Sysbench	A popular benchmarking tool used to generate prime numbers.
Sleep	A function that sleeps for a specified duration.
MST	A function that generates a graph and calculates the min spanning tree.
BFS	A function that generates a graph and processes a breadth first search.
Page Rank	A function that generates a graph and processes page rank of each node.
Writer	A function that generates text and repeatedly writes it to disk and deletes.
Compress	A function that generates files and compresses them into a zip file.
Resize	A function pulls an image from S3, resizes it and saves it back to S3.
DNA	A function pulls DNA sequence from S3 and creates visualization data.

Table 3.3: Function Names and CPU Time Profiles

Name	CPU User Time	Idle Time	CPU Kernel Time	Average Runtime
Calcs Service	30%	70%	0%	3750 ms
Sysbench	90%	9%	1%	3800 ms
Sleep	0%	100%	0%	10000 ms
MST	17%	80%	3%	4080 ms
BFS	16%	80%	4%	6540 ms
Page Rank	19%	80%	1%	4050 ms
Writer	4%	80%	16%	1920 ms
Compress	18%	80%	2%	5740 ms
Resize	20%	80%	0%	5220 ms
DNA	16%	81%	3%	870 ms

as observed over thousands of function invocations where only "Intel Xeon @ 2.5Ghz" was returned by SAAF. Firecracker, as a derivative of KVM, does not expose the real CPU model name, whereas previously AWS Lambda based on Xen exposed the full CPU model name enabling evaluation of performance implications of function execution on heterogeneous CPUs.

Alongside our memory setting models we created models to predict the performance of a workload across different CPUs used by FaaS platforms. Since the Firecracker microVM no longer reveals CPU model information we can only build CPU models for AWS Lambda using the data from 2019. We can build models using all data from IBM Cloud Functions, and we can use the masked CPU returned by AWS Lambda Firecracker functions. Training models that predict performance of a CPU based off another CPU allows us to create predictions that account for hardware heterogeneity on FaaS platforms. If the random distribution of CPUs that a FaaS platform uses is known, we can predict the performance and cost of a large batch of function invocations and apply the random distribution of CPUs to a batch.

Given that the distribution of CPUs for function invocations is random, we needed to make thousands of function calls to accumulate enough data to train models for each CPU. This problem revealed one of the most significant challenges of modeling performance of FaaS platforms. Collecting data for training performance models can be both time consuming and expensive. To address this problem we investigated the trade-off between the volume of training data and performance modeling accuracy. Even though we performed tens of thousands of function invocations, we occasionally would only collect a few hundred samples for a specific CPU. The random nature of CPU allocation on FaaS platforms makes it more difficult to train models as we always have inconsistent amounts of training data in our source and target data sets. To establish ground truth and measure the accuracy of our predictions, we have created a process of establishing ground truth.

Table 3.4: Calcs Service Experimental Workload Configurations

Name	Calculations	Memory Stress	Threads	Tenancy
NMT2	40m	No	2	n
NMT2-seq	40m	No	2	1
SCNMT2	$30 \rightarrow 60m$	No	2	n
SCMT2	$30 \rightarrow 60m$	Fixed 1m	2	n
SCSMT2	$30 \rightarrow 60m$	$1 \rightarrow 1m$	2	n

3.4 Experiment Execution and Data Processing

Each of our experimental workloads were executed in the same manner using FaaS Runner. To evaluate our CPU Time Accounting method using Calcs Service, we defined 10 different values of calculations (ranging from 30,000,000 to 60,000,000 with steps of 3,000,000) with 10 different array sizes (1, to 1,000,000 with steps of 100,000). Then we executed batches of 100 function invocations in parallel: repeating 12 times. The first two batches were

Table 3.5: Observed Ratios of CPU Types on AWS Lambda and IBM Cloud Functions FaaS Platforms

Platform	Intel Xeon CPU	VM	%
AWS	E5-2680v2 @ 2.8 GHz, 10 core	c3	67.5
AWS	E5-2676v3 @ 2.4 GHz, 12 core	m4	19.9
AWS	E5-2686v4 @ 2.3 GHz, 18 core	r4	12.5
IBM	E5-2683v3 @ 2.0 GHz, 14 core	unseen	18.4
IBM	E5-2683v4 @ 2.1 GHz, 16 core	b1	66.1
IBM	E5-2650v4 @ 2.2 GHz, 12 core	u1	3.8
IBM	E5-2690v4 @ 2.6 GHz, 14 core	c1	7.2
IBM	Gold 6140 @ 2.3 GHz, 18 core	unseen	4.5

removed to ensure all of our infrastructure measurements reflected a warm state. This process was then repeated for four memory settings memory settings (256MBs, 512MBs, 1024MBs, and 2048MBs) on both AWS Lambda and IBM Cloud Functions. After this process was complete, we had on average 250 function invocations for each CPU and Memory configuration using a variety of workloads. Given that CPUs are randomly distributed, we found at times we would only observe a few invocations on a specific CPU, or a specific calculation/array size tuple was missing completely for a specific CPU. This inconsistency of FaaS platforms meant we could not simplify establish ground truth based on the workload parameters. We may get multiple runs with the same parameters or runs could be missing.

We compiled all the data from this experiment into multiple different workloads defined in table 3.4. Each workload varies the complexity of the Calc Service function. For example, in SCNMT2 we take data with any calculation value but only take runs with an array size of 1. Then we filter the data by the specific FaaS configuration we want: consisting of a specific CPU, memory setting, and/or function tenancy value. After the data is filtered,

we then split the data into 5 folds to perform cross-fold validation. One fold will be used for testing data, while the other 4 will be used for training. We repeat the training/testing process across all folds, meaning all data is used for both training and testing at some point. To establish ground truth, we pair runs based off their z-score in the overall distribution. For each run the z-score value of the source data set is mapped to a theoretical value in the target data set based upon the mean and standard deviation of the target data. This approach established ground truth and allows for the target and source data sets to have inconsistent amounts of data. To compare the accuracy of our models we use testing data to calculate mean absolute percent error (MAPE). To compare different methods of prediction against each other we also used Wasserstein distance to compare which prediction method reproduced a more similar distribution of function invocations compared to our test data [55]:

$$(u, v) = \int_{-\infty}^{\infty} |U - V|$$

For each iteration of a fold, we create models using CPU time accounting metrics and multiple linear regression or random forest. For both of these methods, we train 3 models, one to predict CPU user time, one to predict CPU idle time, and one to predict CPU kernel time. For all three of these models, we use 8 features returned by SAAF (CPU User, CPU Idle, CPU Kernel, context switches, page faults, major page faults, CPU steal, and free memory) and use stepwise multiple regression to select features optimizing R^2. For baseline runtime predictions we simply predict runtime directly (treating runtime as the only dependent variable). After the training is complete, we test the models on the testing fold, by using our pairing method we are then able to compare our predicted values to real values of the alternate FaaS configuration. After all folds are complete, we evaluate the accuracy of our models based on the overall mean absolute percent error (MAPE), degrees of freedom (DF), and coefficient of variation (CV), of all folds.

We used the same experimental method used for the Calcs Service for our other nine functions. We defined 10 different workloads to vary runtime and executed 12 batches of 100 function invocations (excluding the first 2) across different memory settings. We expanded the number of memory settings used for our workload evaluation from 5 to 7 (256MBs,

512MBs, 1024MBs, 2048MBs, 4096MBs, and 8192MBs). All of our functions were deployed to AWS Lambda and pinned to the same availability zone using a VPC.

Since we are now using multiple functions, each function needed a unique set of parameters to define our workloads. Our graph based functions (MST, BFS, and Page Rank) generated a deterministic tree data structure made up of 500,000 nodes and executed each algorithm over 50 to 150 loops to achieve 30 to 120 seconds of runtime. Due to the limited memory of FaaS platforms at low memory configurations, we could not continuously expand the size of the tree data structure to increase function runtime. Function instances would quickly run out of memory before achieving considerable runtime. To mitigate this, we instead implemented loops in our functions where they would instead repeatedly execute their algorithm. The sleep function simply slept between 10 and 20 seconds. The resize function pulls a single 1MB image from Amazon S3 and resizes it to half of its original size 50 to 150 times and then uploads the resized image back to S3. The writer and compress functions generate about 250MBs of text and either repeatedly write it to disk and delete it, or write to disk and repeatedly compress the file into a zip archive 10 to 20 times. The DNA visualization function downloads a 1MB to 2MB DNA sequence file and generates visualization data. Using the prime number generation benchmark included in our Sysbench function, we created payloads to calculate from 40,000 to 4,000,000 prime numbers in steps of 40,000. Once all of our experiments were executed, we used the same data processing methodology used on our Calcs Service experiments across all nine of these functions.

Chapter 4

EXPERIMENTAL RESULTS

To evaluate our research questions, we profiled the workloads described in the Tables 3.1 and 3.4. We deployed the Calcs Service function to AWS Lambda on the Virginia region. We pinned all Lambda functions to execute within a virtual private cloud (VPC) on the same availability zone (e.g. us-east-1b). This configuration reduces hardware heterogeneity as different availability zones have been shown to use different infrastructure [56]. We also deployed our functions to the us-south Dallas region on IBM Cloud Functions. To investigate the accuracy of our novel CPU Time Accounting performance modeling techniques, we first assessed the performance variation of each FaaS platform using the NMT2 workload (RQ-1). Then we evaluated the accuracy of our CPU Time Accounting approach to performance modeling across a variety of FaaS hardware scenarios: different CPUs, memory configurations, tenancy, and platforms (RQ-2). See table 4.1 for all scenario groups. We evaluate the impact of dataset size on the accuracy of predictions (RQ-3), and address the process of data collection and feature selection for building accurate models (RQ-4). Finally, we expand the evaluation of our performance modeling approach to include nine other functions to assess its accuracy over varying workloads and memory configurations (RQ-5).

4.1 Performance Variation Evaluation of FaaS Platforms

To investigate performance variation inherent to FaaS platforms, we executed the NMT2 workload on AWS Lambda and IBM Cloud Functions. This workload involved performing a fixed number of calculations without memory stress. Combined all runs of this workload provided a deterministic benchmark. Prior to the advent of Firecracker, we observed that AWS Lambda had a coefficient of variation (CV) of 10% when all functions were pinned to the same us-east-1c subnet using a VPC. We replicated this test on AWS Lambda which

now uses the Firecracker hypervisor, and the same function deployment now results in just 1.2% CV. When filtering data by CPU type (a capability available prior to the advent of Firecracker on AWS Lambda), we measured CV of 0.72% and 0.73% CV for the E5-2690v2 and E5-2686v4 CPUs respectively. This difference in performance variation strongly suggests that AWS Lambda, now supported by the Firecracker hypervisor, may employ homogeneous CPUs as the CV is significantly closer to an individual CPU's CV than the platform's overall CV prior to Firecracker. See Figure 4.1 for a comparison of CV between different CPUs on the Xen hypervisor compared to the Firecracker microVM. When AWS Lambda Firecracker functions are not pinned to a VPC, but instead let to run on any availability zone within the region, performance variation with Firecracker increased to 3.3%. Finally, performance variation was significantly higher on IBM Cloud Functions- over 27%. This is likely due to the differences between CPU sharing policies provided by AWS Lambda and IBM Cloud Functions. On AWS Lambda, functions are assigned a fixed amount of CPU time based on memory settings, while CPU timeshare is not restricted on IBM Cloud Functions [56]. We can highlight this difference by comparing the CV when functions are executed in parallel and sequentially. Figure 4.2 shows the impact of multi-tenancy on performance variation of both AWS Lambda and IBM Cloud Functions. Both platforms exhibit an increase in CV when functions are executed in parallel, AWS Lambda ranging from 100% increase in CV to nearly 0% at 2048MBs. This is due to the maximum tenancy at 2048MBs being 1 on AWS Lambda, resulting in no multi-tenancy. IBM Cloud Functions shows much higher CV at all memory settings, with parallelism adding between 50% to over 200% CV, resulting in significantly less consistent runtime when functions are executed in batches.

4.2 CPU Time Accounting Evaluation

In [56] we evaluated our CPU time accounting (CPU-TA) prediction approach with 77 different scenarios, predicting runtime with heterogeneous CPUs, different memory configurations, and on AWS Lambda and IBM Cloud Functions. We executed the same workloads and generated runtime predictions using random forest regression using both CPU time accounting principles and baseline runtime predictions.

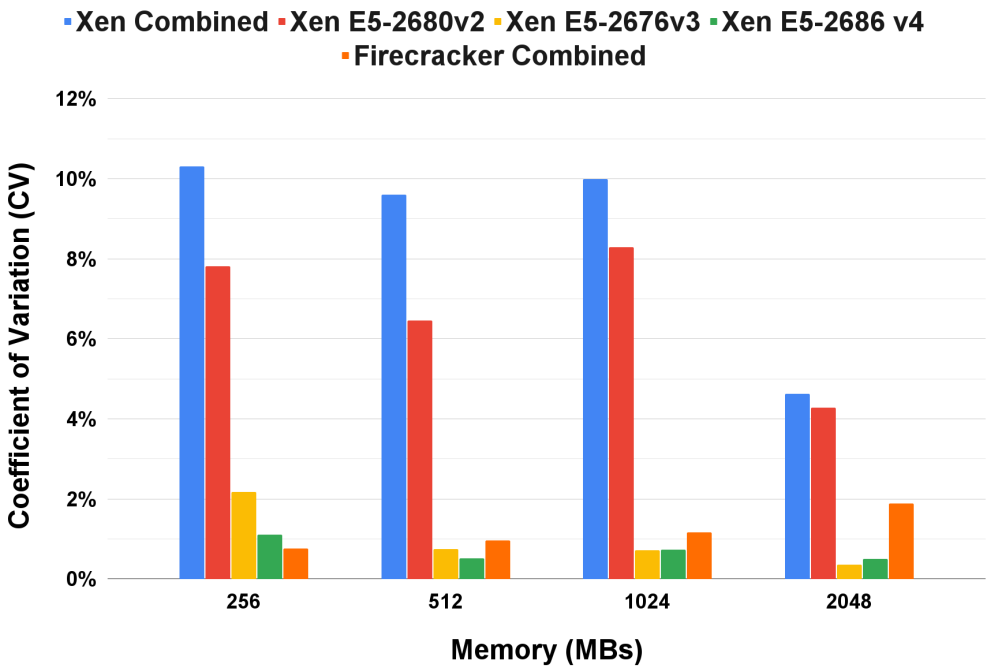


Figure 4.1: Comparison of Coefficient of Variation Between Different CPUs and Hypervisors on AWS Lambda

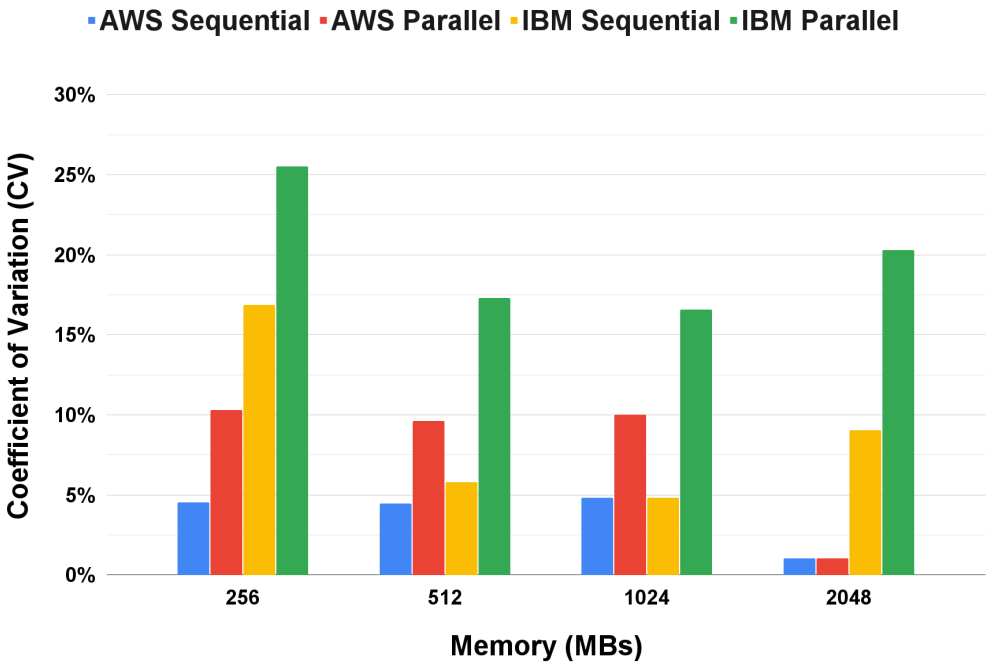


Figure 4.2: Comparison of Coefficient of Variation Between Functions Executing in Parallel or Sequentially on AWS Lambda (Xen) and IBM Cloud Functions

Table 4.1: Categories and Examples of Prediction Scenarios

Category	Description	Scenarios
AWS CPU	Predict between one CPU to another on AWS Lambda. Fixed memory settings. Ex: AWS E5-2680v2→E5-2676v3 at 256MBs	74
AWS Memory	Predict between one memory setting to another on Lambda. Fixed CPUs. Ex: AWS E5-2680v2 at 256MBs→512MBs	96
AWS to IBM CPUs	Predict one CPU on IBM using one from AWS Lambda. Fixed memory settings. AWS E5-2680v2→IBM E5-2683v3 both at 256MBs	24
IBM CPU	Predict between one CPU to another on AWS Lambda. Fixed memory settings. Ex: IBM E5-2650v4→E5-2690v4 at 256MBs	111
IBM Memory	Predict between one memory setting to another on IBM. Fixed CPUs. Ex: Ex: IBM E5-2683v4 at 512MBs→2048MBs	143
AWS Firecracker	Predict between one memory setting to another on AWS Lambda using Firecracker. Ex: 256MBs→512MBs using Resize Function	300

The SCNMT2 workload showed the largest margin in performance prediction improvement when comparing runtime predictions for different hardware configurations. Of the 20 prediction scenarios, CPU time accounting based predictions out preformed the baseline runtime prediction 14 times. Prediction accuracy was compared using mean absolute percent error (MAPE) and Wasserstein distance. For functions that ran with 256MB memory, predictions from the Intel Xeon E5-2680v2 to the E5-2686v4 processor using CPU time accounting and random forest saw 3.3% less error when compared to the baseline runtime prediction approach with random forest. For the winning scenarios, t-tests show the prediction improvement to be significant in 10/14 of the scenarios with p-values ranging from 0.04 to <0.01 with degrees of freedom ranging from 50 to 500 samples. For the more complex SCMT2 and SCSMT2 workloads, CPU time accounting prediction and the baseline runtime prediction approaches performed nearly identically using multiple linear regression and random forest. CPU time accounting and runtime based predictions had a difference of only 0.14% and 1.2% MAPE for SCNMT2 and SCMT2 respectively with an average of 0.68% MAPE difference across the remaining 40 CPU prediction workloads.

In our previous work [56], memory prediction scenarios were limited to only predicting from 256MBs to higher memory settings on equivalent CPUs. We have expanded the number of memory scenarios to include all possible permutations of memory settings, resulting in a total of 96 prediction scenarios with our three Calcs Service workloads and four modeling techniques. Making predictions across memory settings proved to be more challenging than CPUs scenarios because there are two distinct factors that underpin the performance of these scenarios: CPU time share allocation and variable function tenancy of hosts. Out of the 96 scenarios, CPU time accounting "won" by providing superior predictions with lower MAPE in 37 scenarios with significant p-values less than 0.05. For the other 59 scenarios, 44 had similar error where the difference in MAPE between the CPU time accounting models and our baseline predictions was not statistically significant. In these settings, MAPE ranged from 0.01% to 0.93%. Wassterstein distance also identified the similarity of these models, in this case it ranges from as low as 1 to 50 where models with significant prediction improvement ($p<0.05$) have a difference of more than ± 100 Wassterstein distance. CPU time accounting provided statistically equivalent or better predictions for the majority of

scenarios when compared with our baseline approaches. CPU time accounting only "lost" by providing predictions with higher MAPE in 15 scenarios with significance ($p<0.02$). These 15 scenarios tended to occur when predictions were made across large memory deltas (e.g. 256MBs to 2048MBs with the Intel Xeon E5-2686v3 CPU).

Prediction accuracy differences between CPU time accounting methods and baseline approaches that predict runtime directly can likely be attributed to an inherent observability challenge of FaaS platforms: multi-tenancy. As discussed earlier, on some platforms such as AWS Lambda prior to Firecracker, and on IBM Cloud Functions, CPU metrics such as CPU user time and CPU idle time must be normalized by the tenancy of the function to make accurate runtime predictions. SAAF and FaaS Runner attempt to accurately calculate the tenancy of function instances by counting the number of concurrent function instances that executed in parallel. This metric ultimately becomes the maximum tenancy of a function over the duration of its runtime. Tenancy is a dynamic metric. For example, if a function invocation spends 90% of its runtime executing in isolation (tenancy of 1), and during the last 10% of the runtime 10 new function instances are created on the same host, the tenancy value results in 10. This occurs even though the tenancy was only 1 for the majority of the runtime. For this scenario CPU time accounting methods would not make accurate predictions because the CPU metrics will not accurately total the correct runtime using the CPU time accounting equation. To address this problem we have extended SAAF and FaaS Runner to now calculate a new metric to characterize function tenancy, which we call runtimeOverlap. We plan to leverage runtimeOverlap in the future to improve our predictions by better accounting for function tenancy.

Similar to the execution of CPU prediction scenarios, we also attempted to predict performance across FaaS platforms. We used data from AWS Lambda to predict performance of different CPUs on IBM Cloud Functions. For this smaller test, we used the SCNMT2 workload to evaluate 24 prediction scenarios. Like for memory predictions, CPU time accounting provided predictions with significantly ($p<0.05$) lower MAPE in 8 of the 24 prediction scenarios. MAPE was higher for 7 scenarios compared to baseline runtime predictions, and the remaining 9 scenarios could not be statistically differentiated.

In addition to predicting IBM Cloud Functions runtime based on AWS Lambda, we

predicted performance across different CPUs on IBM Cloud Functions with mixed tenancy. This was likely our most complex scenario attempted as IBM Cloud Function's performance varies greatly depending on the tenancy of a function [56]. While AWS Lambda assigns a fixed CPU timeshare based on a function's memory setting, IBM Cloud Functions are not constrained but allowed to compete for the host's available CPU timeshare. The memory setting simply limits the number of concurrent function instances that can share a host. In this dataset, we have a total of 254 prediction scenarios across different CPUs, different memory settings, and different tenancy values. In these scenarios, neither CPU time accounting based predictions nor runtime based predictions were very accurate. MAPE predictions ranged from as low as 7% up to 17%, much higher than any of the AWS Lambda predictions.

Across all of our prediction groups, we evaluated a total of 448 different prediction scenarios. Of those, CPU time accounting predictions out performed baseline runtime prediction approaches 149 times, producing less MAPE with significance established by t-test ($p<0.05$). Across all scenarios, CPU Time Accounting averaged 4.9% MAPE, while the baseline runtime predictions had 5.1% MAPE. In 184 scenarios, CPU time accounting and baseline runtime predictions performed similarly, where there was no statistically significant difference observed between the methods. Comparing predictions between multiple linear regression and random forest revealed that both methods perform similarly for both CPU-TA or baseline runtime predictions. No single prediction scenario revealed a significant difference between predictions made with multiple linear regression vs. random forest, resulting in a difference of only 0.05% MAPE across all scenarios in favor of multiple linear regression. Table 4.2 shows the results of each prediction scenario and accuracy of our CPU Time Accounting approach. For a complete summary of results for all of our prediction scenarios see [64]. Figure 4.3 shows the number of scenarios, across RQ-2 prediction categories, where CPU-TA outperformed the baseline method, performed equivalently, or under performed.

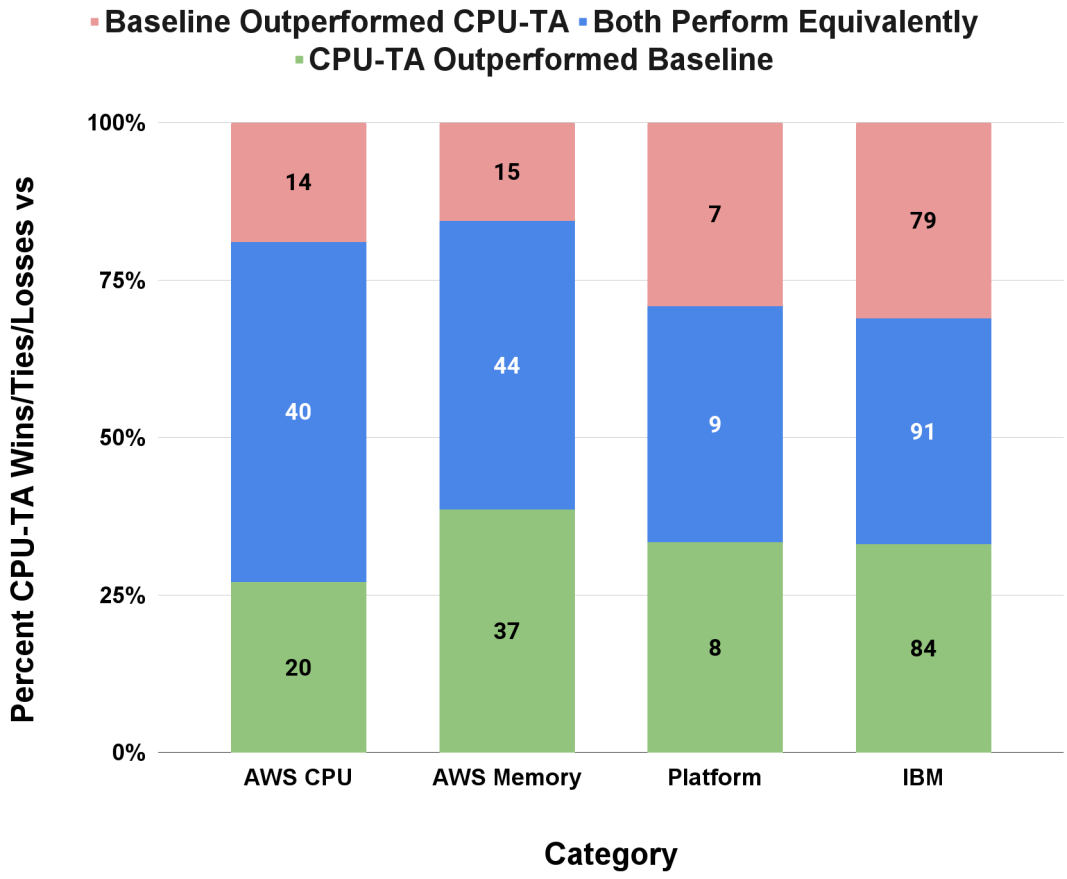


Figure 4.3: RQ-2: Count and percent of scenarios where CPU Time Accounting outperformed, performed equivalently, or under performed in prediction accuracy compared to baseline prediction method

Table 4.2: Runtime prediction comparison: CPU Time Accounting (CPU-TA) vs. Baseline Runtime Prediction Approaches (BASE) using MAPE.

Prediction Group	Scenarios	TA>Base ($p<0.05$)	TA≈Base (n.s.)	TA<Base ($p<0.05$)	MAPE Range
AWS CPU	74	20	40	14	1-4%
AWS Memory	96	37	44	15	1.5-5%
AWS to IBM	24	8	9	7	3-8%
IBM	254	84	91	79	7-17%
Total	**448**	**149**	**184**	**115**	

4.3 Data Size Evaluation

One of the challenges on FaaS platforms when building performance prediction models is knowing how much data we need to collect to have accurate predictions. Every millisecond of runtime on FaaS platforms has a cost so knowing the amount of data needed to create accurate models is important.

To evaluate the accuracy of our models as the number of samples is changed, we executed 100 iterations of each prediction scenario, incrementing the volume of data used for training. The first iteration was given 1% of the data up to 100% in steps of 1%. Figure 4.4 shows a scatter plot of each of these iterations, comparing the dataset degrees of freedom to its prediction MAPE. With less than 100 samples, prediction accuracy varies immensely (min: 1%, average: 10%, max: 40%). Models with this little data are not reliable. After 1000 samples, this range drops considerably (min: 1%, average: 4%, max: 15%). On FaaS platforms collecting thousands of samples is not particularly difficult. Applications such as FaaS Runner can automate experiments to perform thousands of function invocations in only a few minutes. Depending on the workload and memory configuration, this may be a very inexpensive task. Profiling workloads that are more demanding, however, that require

high memory such as 10GB (now supported on AWS Lambda) may be very expensive to profile to make accurate predictions.

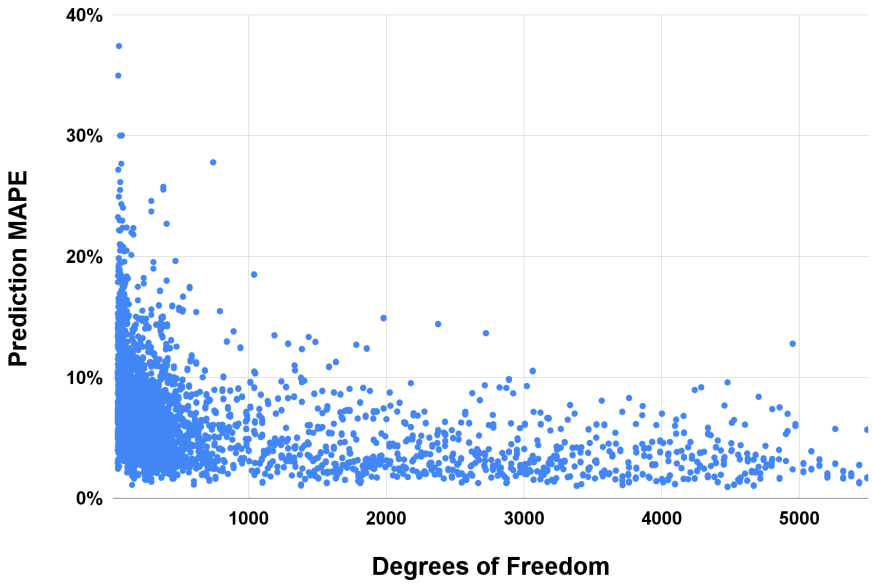


Figure 4.4: Mean Absolute Percent Error as Degrees of Freedom is Changed

For both CPU time accounting and baseline runtime predictions there was a consistent trend where approximately 1,500 samples were required to provide below 10% MAPE for all prediction scenarios as shown in Figure 4.5. This trend varied where some scenarios were less sensitive to the lack of data than others resulting in lower MAPE. Conversely as the amount of available data increased, the error in our predictions decreased.

4.4 Accessing Feature Selection

Applications on serverless FaaS platforms are billed based on the actual runtime of the application, and recently cloud providers have improved the granularity of billing to the nearest millisecond [9]. For frequently used functions, this advancement encourages developers to optimize function performance even if by just a few milliseconds as the costs will add up over time. The Severless Application Analytics Framework builds around this

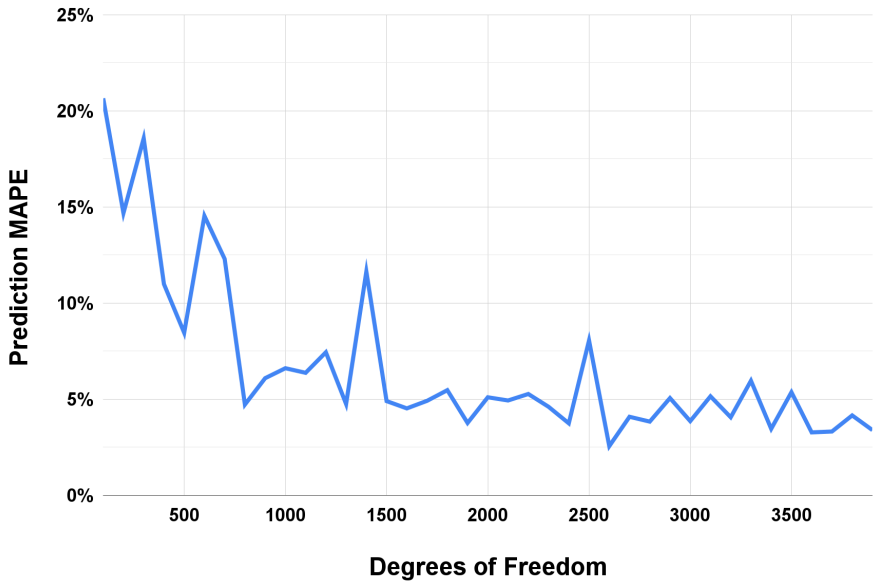


Figure 4.5: Rolling Average of Prediction Accuracy (MAPE) Compared to Data Set Degrees of Freedom

design philosophy by providing developers with fine-grained control over the profiling data collected and returned. With this in mind, to reduce profiling overhead for performance modeling, developers need to know which features are most important for training accurate performance models.

While SAAF provides potentially many useful features to introspect about the functionality of FaaS platforms, we are only required to model a select few for use in our CPU time accounting prediction models (i.e. cpuUsr, cpuKrn, and cpuIdle). For the baseline runtime predictions, the only independent variable was runtime. Feature selection becomes more complex when using our CPU time accounting method. Each of our models that predicts a single CPU metric requires assessment of the CPU metric itself alongside a selection of other metrics. We found that for our workloads, eight SAAF metrics could be used to predict runtime: CPU User, CPU Idle, CPU Kernel, context switches, page faults, major page faults, CPU steal, and free memory. To perform feature selection we utilized stepwise

multiple regression to iterate over each model, comparing the R^2 value as new metrics were added. This resulted in hundreds of different combinations of features for which features were used in our models. Similarly, we use the gini importance metric to determine the most important features with our random forest regression models. Using both of these approaches to feature selection, we found that the free memory, context switches, and page fault metrics were the most commonly selected features when training models that complemented the metric being predicted. Table 4.3 shows the number of times each feature was placed at each rank (1-8) using gini importance.

Rank	CPU User	CPU Idle	CPU Kernel	Contxt Swch	Page Faults	CPU Steal	Free Mem	Maj Pg Fault
1	287	79	0	4	0	3	47	0
2	121	131	0	75	4	9	80	0
3	12	137	0	198	1	34	38	0
4	0	34	7	69	77	21	212	0
5	0	24	55	60	230	15	36	0
6	0	8	278	5	87	25	7	0
7	0	7	80	9	21	277	0	26
8	0	0	0	0	0	26	0	394
Average Rank	**1.35**	**2.63**	**6.03**	**3.37**	**5.09**	**6.14**	**3.31**	**7.94**

Table 4.3: Feature Importance Rank count of each metric: CPU User, CPU Idle, CPU Kernel, Context Switches, Page Faults, CPU Steal, Free Memory and Major Page Faults.

4.5 Workload Evaluation

To build upon the Calc Service prediction scenarios, we added 9 other functions with varying workloads. By using the metrics collected by SAAF we are able to profile each function by observing the distribution of CPU time metrics. CPU-bound functions exhibited mostly CPU User and CPU Idle time. While disk-bound functions exhibited more CPU Kernel or CPU IO Wait time. Figure 4.8 and Figure 4.9 depicts the profile of each function at each of the 6 different memory settings. Even though all of our functions perform different tasks,

their CPU time profile can show similarities and differences in each workload. For example, our writer function and compress function (Figure 4.9 a and b) both do fundamentally the same thing, they generate data and write it to disk. By observing their profile, we can see significant differences. Writer exhibits a large amount of CPU kernel time, nearly 200ms, and very little CPU user time, while compress has almost no kernel time and much more CPU user time. This difference is likely because the task of compressing data requires more CPU user time than simply writing data to disk. Disk I/O alone for the Writer function involves primarily kernel time due to system calls for I/O scheduling, caching, and I/O Protection. The DNA visualization function and Resize function (Figure 4.9 c and d) both utilize S3 to process data over the network. They process data in very different ways but have similar profiles, the only difference being that DNA visualization requires a small amount of kernel time. The MST, BFS, and Page Rank profiles all look very similar as they function in a similar manner (Figure 4.8 c, d, and e) becasue they are all single threaded CPU bound workloads. Their CPU profiles are dominated by idle time with a fixed amount of CPU User time across all memory settings. This observation is due to how AWS Lambda couples CPU time share to memory setting. As the memory settings are decreased, the platform adjusts the CPU time share to inject more idle time into the CPU profile of the function. Idle time is decreased linearly with memory until functions reach a point where function instances are allocated more CPU cores. From 128MBs to about 3GB, function instances are allotted 2 vCPU cores. After 3GBs of RAM, additional cores are added roughly every 2GBs. Figure 4.6 depicts the number of CPU cores for each memory setting alongside the theoretical and real performance speedup of function instances.

Additional vCPU cores require more CPU Idle time to be added by the FaaS Platform to maintain linear performance scaling. The increase in vCPUs results in the profiles of our single threaded workloads looking like they have much higher runtime as the memory setting is increased. Our Sleep function highlights this perfectly in Figure 4.8 f as these function invocations have 1000ms of runtime. At 256MBs through 2048MBs, function instances have 2 cores so there is exactly 2000ms of CPU idle time, 1000ms for each vCPU across these memory settings. At 4096MBs, the function instances now have 3 cores so we see 3000ms of CPU idle time even though the runtime of the function was only 1000ms. This trend

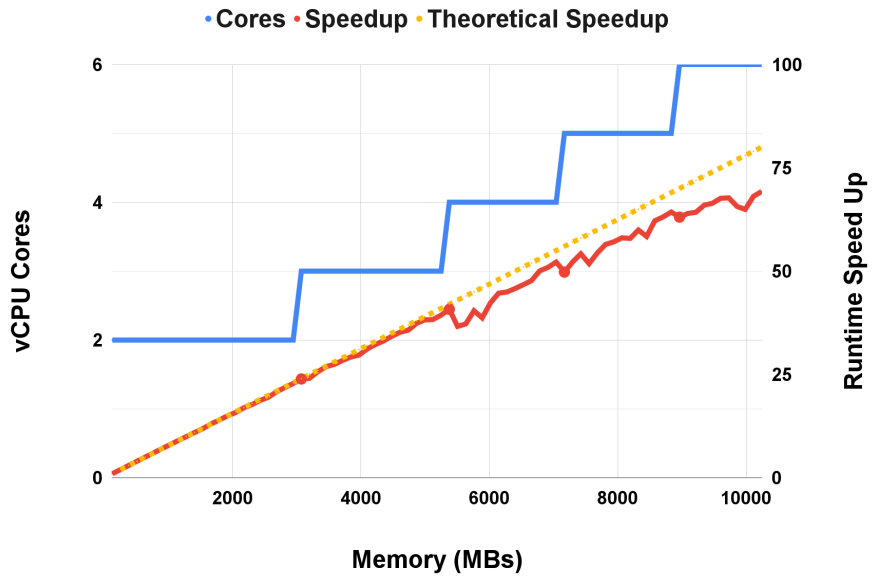


Figure 4.6: Allocation of vCPU Cores as Memory Setting is Increased

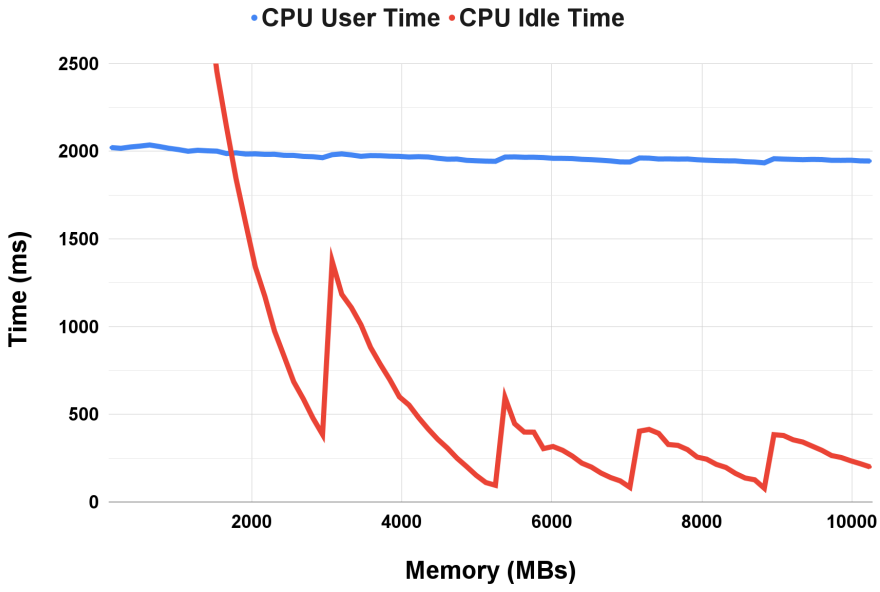
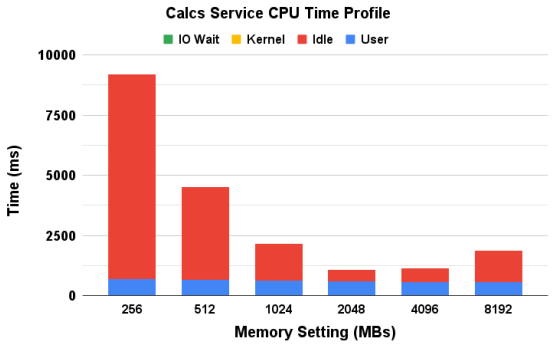


Figure 4.7: Sysbench CPU User and Idle Time as Memory Changes

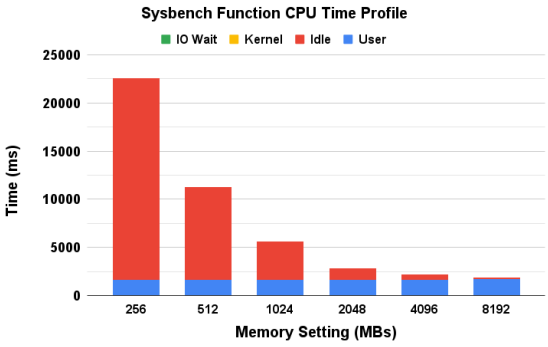
continues with 8192MBs of memory where we now have 5 cores and see 5000ms of idle time. For multi-threaded workloads such as Sysbench that leverage all of the available CPU cores, we do not see this increase in idle time but instead see a linear decrease in idle time across increasing memory settings, approaching less than 100ms at 8192MBs. If we execute a Sysbench workload across a wider number of memory settings and plot CPU user and idle we can clearly see jumps in idle time at the points where additional CPU cores are allocated. Figure 4.7 shows the CPU user and idle time at every available memory setting in steps of 128MBs on AWS Lambda.

To further evaluate our runtime predictions for FaaS functions deployed on AWS Lambda with the Firecracker hypervisor, we deployed our suite of 10 functions and repeated the same memory based predictions performed previously with AWS Lambda using Xen. For each function, we created 30 prediction scenarios where we made predictions across all increasing combinations of the 256MB, 512MB, 1GB, 2GB, 4GB, and 8GB memory settings. For example, predicting the performance of the Compress function at 8GBs based off its performance at 512MBs or predicting Sysbench performance at 4GBs based off 2GBs. For all 10 functions, this resulted in a total of 300 new prediction scenarios. Diversifying to these new workloads revealed scenarios where CPU time accounting predicted performance accurately while also revealing some weaknesses.

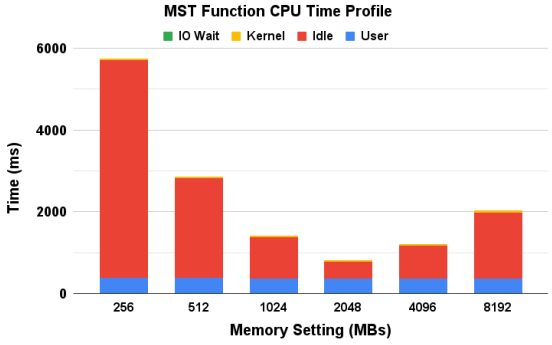
Figure 4.10 shows the average MAPE using CPU time accounting to predict runtime of each workload across varying ranges of memory settings. All of the functions except Sysbench perform close to each other, ranging from as low as 1% error to less than 10% across all memory ranges. Sysbench performs similarly until the gap in memory reaches a difference of around 2GBs and then predictions get significantly worse. Prediction MAPE jumps from about ~10% to ~25% and then after the 6GB memory gap MAPE increases another 10%. This is a unique observation that we can expand upon based on the profile of each function. As we saw earlier, most of our functions are single threaded while we over provisioned Sysbench to use 12 threads. Since AWS Lambda only provisions function instances with a maximum of 6 vCPU cores, creating a workload with 12 threads will require the operating system to context switch between threads more often as not all threads are able to be executed in parallel. What is unique about these jumps in MAPE, observed in



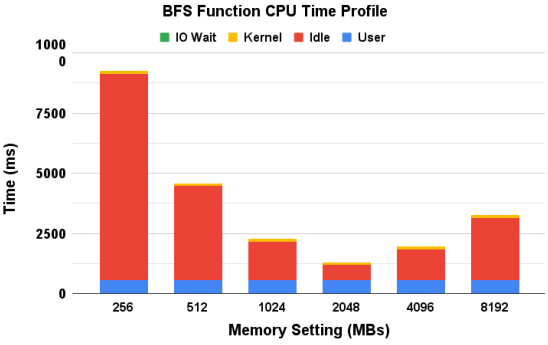
(a) Calcs Service Function Profile



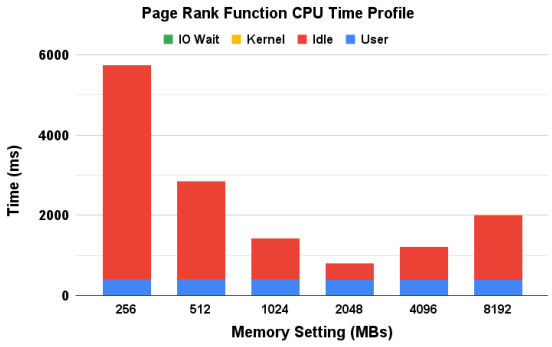
(b) Sysbench Function Profile



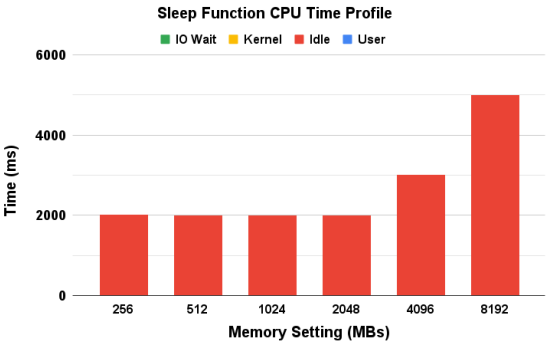
(c) MST Function Profile



(d) BFS Function Profile

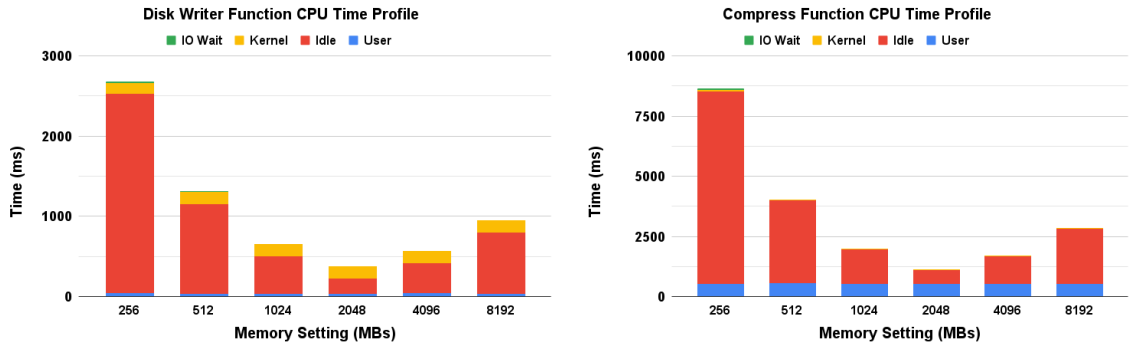


(e) Page Rank Function Profile



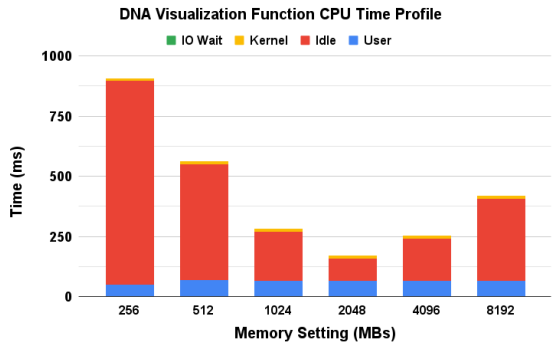
(f) Sleep Function Profile

Figure 4.8: CPU Time Profile of Functions

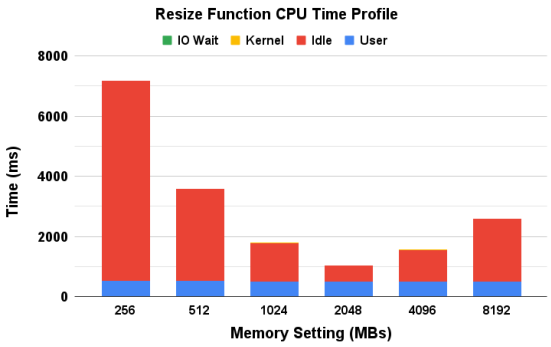


(a) Writer Function Profile

(b) Compress Function Profile



(c) DNA Visualization Function Profile



(d) Resize Function Profile

Figure 4.9: CPU Time Profile of Functions Continued

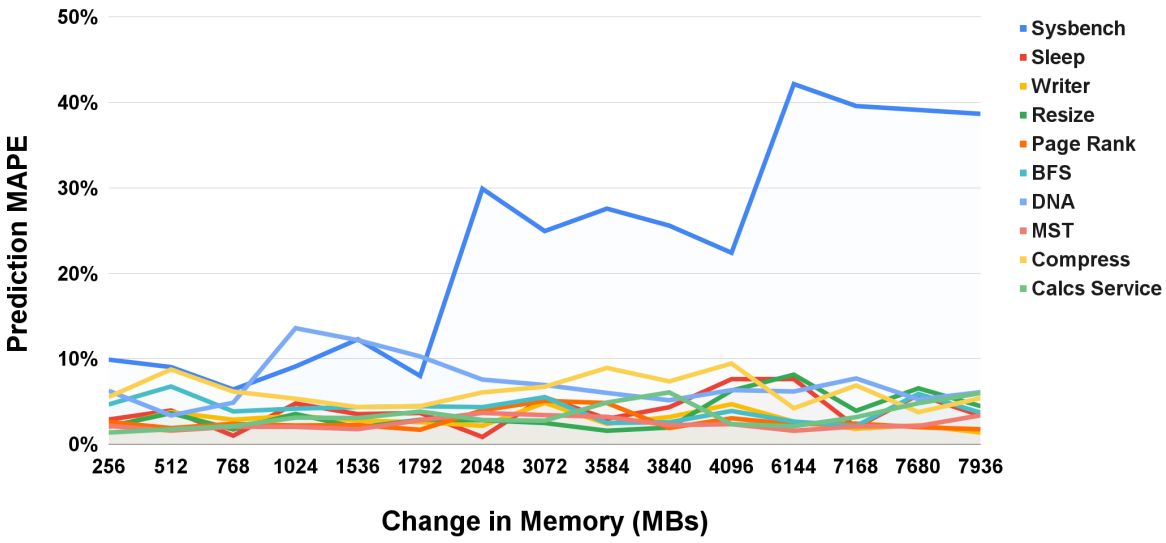


Figure 4.10: CPU Time Accounting Prediction Mean Absolute Percent Error for Each Workload Across Different Ranges of Memory Settings

Figure 4.10, is that they occur when the change in memory settings span across function instances with a different number of vCPUs cores. As long as both the source and target scenario have the same number of allocated vCPUs, our predictions are as accurate as our single threaded workloads. Since Calcs Service never takes advantage of more than 2 threads, as is not over provisioned we never saw this behavior. Multithreaded applications are inherently less predictable as scheduling by the operating system can introduce randomness compared to singled threaded applications.

Alongside the challenge revealed by Sysbench, CPU metrics such as CPU User, CPU Idle, and CPU Kernel time are not recorded in milliseconds, but centiseconds. This provided insufficient profiling granularity as measurements failed to observe ~10ms of the function's execution. For functions with a runtime of a few seconds or more, this delay becomes insignificant. For our testing, however, some of our function invocations, most notably with the DNA visualization function, completed in under 100ms of total runtime. This alongside the unknown tenancy of Firecracker function instances resulted in CPU time accounting

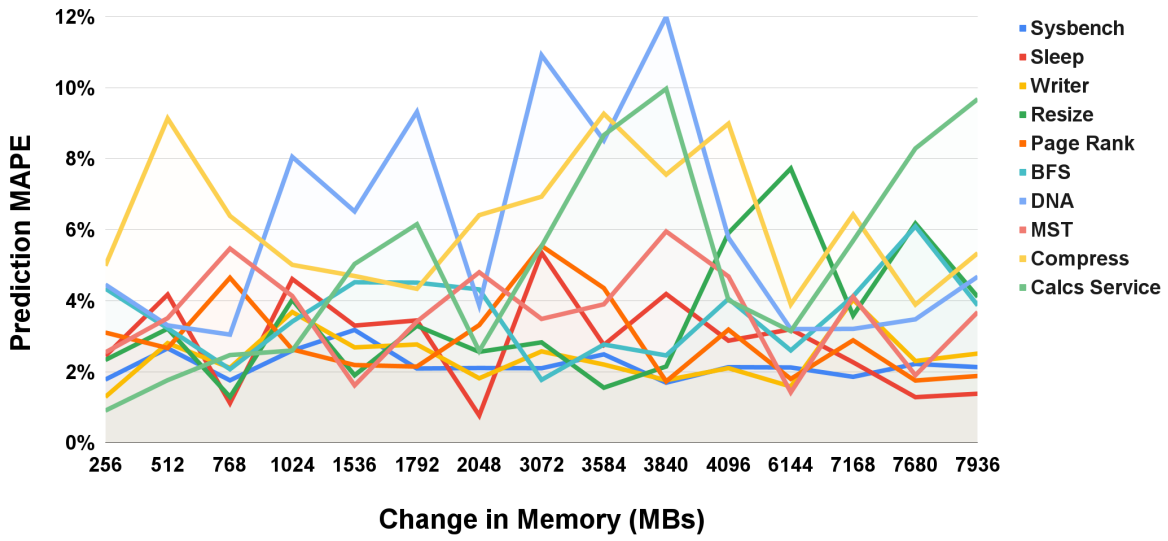


Figure 4.11: Runtime Modeling Prediction Mean Absolute Percent Error for Each Workload Across Different Ranges of Memory Settings

producing inaccurate predictions with up to ~12% MAPE.

Comparing our CPU Time Accounting runtime prediction method to the baseline methods we see that once again on average CPU time accounting out performs the baseline method in more scenarios. Figure 4.11 shows the MAPE of the runtime based predictions. As the figure shows, the baseline method was able to handle Sysbench much better. Of the 300 scenarios containing all 10 of these functions, CPU time accounting was able to achieve 134 significant "wins" ($p<0.02$) where MAPE was lower than the baseline method. We observed 75 scenarios where both methods performed similarly and 91 scenarios where CPU time accounting lost. To no surprise, CPU time accounting was outperformed in all 30 of the scenarios using Sysbench. Figure 4.12 shows the number of scenarios, across all prediction categories, where CPU-TA outperformed the baseline method, performed equivalently, or under performed.

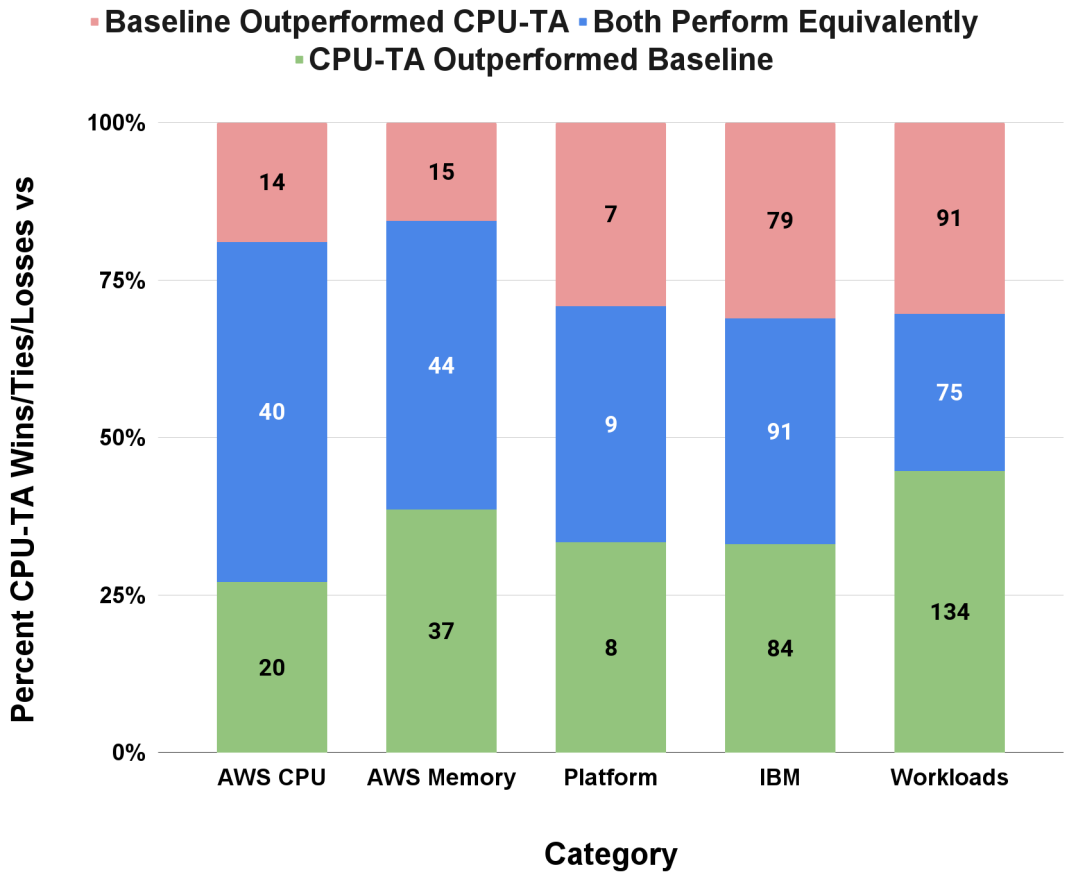


Figure 4.12: RQ-4: Count and percent of scenarios where CPU Time Accounting outperformed, performed equivalently, or under performed in prediction accuracy compared to baseline prediction method

Chapter 5

CONCLUSIONS

In this thesis, we introduced our novel CPU Time Accounting performance modeling approach for predicting FaaS function runtime. We compared performance modeling accuracy of our CPU Time accounting modeling approach to prediction methods that directly predict runtime using 448 FaaS prediction scenarios, for example predicting between different CPU types or memory configurations. **RQ-1:** Using a simple workload, we observed that CPU heterogeneity caused up to 27% variation on IBM Cloud Functions, and 8.2% on AWS Lambda prior to using the Firecracker microVM. After implementing Firecracker, AWS Lambda showed similar performance variation to individual CPUs using the Xen hypervisor, suggesting that AWS Lambda now uses homogeneous CPUs. **RQ-2:** We implemented both CPU Time accounting and baseline models with multiple linear regression and random forest regression. For the 448 scenarios, 333 scenarios either showed superior performance predictions with CPU Time Accounting, or equivalent performance to the baseline method. In the majority of scenarios, CPU Time Accounting was able to make accurate performance predictions within 94% to 99% of the target scenario. Across all scenarios, CPU Time Accounting averaged 4.9% MAPE while the baseline runtime predictions had 5.1% [64]. **RQ-3:** We observed the impact of varying the data set size on runtime predictions. We found that around ~1500 function invocations were required to stabilize prediction error below 10% on AWS Lambda. This equates to only $0.12 for a function with 10 seconds of runtime at 512MBs of memory. **RQ-4:** We evaluated feature selection using gini importance in random forest to identify the most informative metrics collected from SAAF. Alongside CPU User, Idle, and Kernel time that are required by our CPU Time Accounting approach, we also found that context switches, free memory, and page faults were important features for our performance predictions as ranked using gini importance. **RQ-5:** We evaluate our CPU time accounting prediction approach on a variety of different functions

and workloads. Using our suite of 10 functions, we observed more accurate or equivalent performance predictions in 209 out of 300 scenarios. Using this diverse set of workloads revealed multi-threading to cause significantly less accurate predictions when source and target function instances had a different number of vCPUs allocated to them. With further development and more training data, the CPU time accounting method may be able to account for this challenge.

APPENDIX

Glossary

Calcs Random math calculations used in the Calcs Service Function. 15

Calcs Service A serverless function that executes random math calculations using arrays to invoke CPU and memory stress. 14

CPU-TA Our novel Linux CPU Time Accounting performance prediction approach. 24

cpuIdle Time spent by the CPU idling. 13

cpuIntSrvc Time spent by the CPU servicing hardware interrupts. 13

cpuIOWait Time spent by the CPU waiting for I/O to complete. 13

cpuKrn Time spent by the CPU executing processes in Kernel Mode. 13

cpuUsr Time spent by the CPU executing processes in User Mode. 13

Experiment files JSON files used by the FaaS Runner that define the input JSON payloads, memory settings, and other attributes used to execute an experiment on a FaaS platform. 11

function instances The environment used to execute functions on FaaS platforms. Function instances can be containers, virtual machines, or a microVM. The specific implementation of a function instance depends on the platform. 4

JSON Javascript Object Notation, the main data notation used to send information to and from FaaS microservices. 10

multi-tenancy When multiple function instances share the same host infrastructure. 3

payloads The JSON Object sent to or received from a microservice. 11

SAAF The Serverless Application Analytics Framework, a library included inside FaaS functions. Used to profile hardware and performance of function instances. 4

scenarios A source and target configuration that are used to make predictions across. 24

workload A set of JSON payloads used to define the work that will be executed on the FaaS platform. 3

BIBLIOGRAPHY

[1] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, 2016, p. 5.

[2] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[3] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2017*, 2017.

[4] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018.

[5] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[6] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. the USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[7] A. Sill, "The Design and Architecture of Microservices," *IEEE Cloud Computing*, 2016.

[8] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean Lambdas with Large Libraries," in *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, 2017.

[9] AWS, "AWS Lambda – Serverless Compute - Amazon Web Services," https://aws.amazon.com/lambda/, [Online; accessed 12-Apr-2021].

[10] Microsoft Azure, "Azure Functions," https://azure.microsoft.com/en-us/services/functions/s.

[11] I. Baldini, K. Chang, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, V. Ishakian, and P. Suter, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, 2017.

[12] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, and Others, "Cloud programming simplified: a berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[13] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[14] A. Eivy, "Be Wary of the Economics of 'Serverless' Cloud Computing," *IEEE Cloud Computing*, 2017.

[15] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," *IEEE Cloud Computing*, 2016.

[16] M. Eisa, M. Younas, K. Basu, and H. Zhu, "Trends and directions in cloud service selection," in *Proceedings - 2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016*, 2016.

[17] M. Eisa, M. Younas, and K. Basu, "Analysis and representation of QoS attributes in cloud service selection," in *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 2018.

[18] AWS, "AWS Lambda Pricing Calculator," https://s3.amazonaws.com/lambda-tools/pricing-calculator.html.

[19] Peter Sbarski, "Serverless Cost Calculator," http://serverlesscalc.com/.

[20] IOpipe.com, "Servers.LOL: Serverless Cost Calculator for AWS Lambda - IOpipe," .

[21] Google Cloud, "Google Cloud Function:Event-Driven Serverless Compute Platform," https://cloud.google.com/functions.

[22] IBM, "IBM Cloud Functions," https://www.ibm.com/cloud/functions.

[23] Z. Ou, H. Zhuang, A. Lukyanenko, J. K. Nurminen, P. Hui, V. Mazalov, and A. Yla-Jaaski, "Is the Same Instance Type Created Equal? Exploiting Heterogeneity of Public Clouds," *IEEE Transactions on Cloud Computing*, vol. 1, pp. 201–214, 2013.

[24] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: Exploiting Performance Heterogeneity in Public Clouds," in *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*, 2012, pp. 1–14.

[25] M. S. Rehman and M. F. Sakr, "Initial findings for provisioning variation in cloud computing," in *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, 2010, pp. 473–479.

[26] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, pp. 460–471, 2010.

[27] A. O. Ayodele, J. Rao, and T. E. Boult, "Performance Measurement and Interference Profiling in Multi-tenant Clouds," in *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*, 2015.

[28] W. Lloyd, S. Pallickara, O. David, M. Arabi, and K. Rojas, "Mitigating resource contention and heterogeneity in public clouds for scientific modeling services," in *Proceedings - 2017 IEEE International Conference on Cloud Engineering, IC2E 2017*, 2017.

[29] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, H*, 2015.

[30] J. White, M. Matalka, W. F. Fricke, and S. Angiuoli, "Cunningham: a BLAST Runtime Estimator," *Nature Precedings*, 2011.

[31] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *Proceedings - International Conference on Data Engineering*, 2009.

[32] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *Proceedings - International Conference on Data Engineering*, 2010.

[33] M. Hafizhuddin Hilman, M. A. Rodriguez, and R. Buyya, "Task runtime prediction in scientific workflows using an online incremental learning approach," in *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2018*, 2019.

[34] R. F. Da Silva, G. Juve, M. Rynge, E. Deelman, and M. Livny, "Online Task Resource Consumption Prediction for Scientific Workflows," in *Parallel Processing Letters*, 2015.

[35] T. P. Pham, J. J. Durillo, and T. Fahringer, "Predicting Workflow Task Execution Time in the Cloud using A Two-Stage Machine Learning Approach," 2017.

[36] R. Ghosh, F. Longo, V. K. Naik, and K. S. Trivedi, "Modeling and performance analysis of large scale IaaS clouds," *Future Generation Computer Systems*, 2013.

[37] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proceedings - International Conference on Distributed Computing Systems*, 2011, pp. 559–570.

[38] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Proceedings - 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, 2012, pp. 612–619.

[39] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud," in *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010*, 2010, pp. 236–243.

[40] A. Andrzejak, D. Kondo, and S. Yi, "Decision Model for Cloud Computing under SLA Constraints," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 257–266.

[41] Q. Zhang, Q. Zhu, and R. Boutaba, "Dynamic Resource Allocation for Spot Markets in Cloud Computing Environments," *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pp. 178–185, 2011.

[42] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: the prospect of serverless scientific computing and HPC," in *Communications in Computer and Information Science*, 2018.

[43] 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.

[44] 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.

[45] 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.

[46] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018.

[47] 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 International Conference on Cloud Engineering (IC2E)*, jun 2019, pp. 23–33.

[48] M. Fotouhi, D. Chen, and W. J. Lloyd, "Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 49–54.

[49] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring Serverless Computing for Neural Network Training," in *IEEE International Conference on Cloud Computing, CLOUD*, 2018.

[50] E. F. Boza, C. L. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza, "Reserved, on demand or serverless: Model-based simulations for cloud budget planning," in *2017 IEEE 2nd Ecuador Technical Chapters Meeting, ETCM 2017*, 2018.

[51] 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 International Conference on Performance Engineering, Tutorial*, 2021.

[52] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, 2016.

[53] L. F. A. Jr, F. S. Ferraz, R. F. A. P. Oliveira, and S. M. L. Galdino, "Function-as-a-Service X Platform-as-a-Service : Towards a Comparative Study on FaaS and PaaS," *The Twelfth International Conference on Software Engineering Advances Function-as-a-Service*, 2017.

[54] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, 2020.

[55] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–276. [Online]. Available: https://doi.org/10.1145/3358960.3379133

[56] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting Performance and Cost of Serverless Computing Functions with SAAF," in *6th IEEE International Conference on Cloud and Big Data Computing (CBDCOM 2020)*, 2020.

[57] 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 *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pp. 67–72.

[58] R. Cordingly, N. Heydari, H. Yu, V. Hoang, Z. Sadeghi, and W. Lloyd, "Enhancing observability of serverless computing with the serverless application analytics framework," 2021.

[59] Apache OpenWhisk, "Apache OpenWhisk Official Website," https://openwhisk.apache.org, [Online; accessed 1-May-2018].

[60] "Fn Project – The Container Native Serverless Framework," https://fnproject.io/.

[61] 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 International Conference on Cloud and Big Data Computing (CBDCOM 2020)*, 2020.

[62] 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 Transactions on Cloud Computing*, 2015.

[63] 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.

[64] Robert Cordingly, "Complete Performance Modeling Results," github.com/wlloyduw/SAAF/tree/master/research/performance_modeling.