Predicting ARM64 Serverless Functions Runtime: Leveraging function profiling for generalized performance models

Xinghan Chen, Robert Cordingly, Ling-Hong Hung, Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma, Washington, USA
kirito20, rcording, lhhung, wlloyd@uw.edu

***Abstract*—In this paper, we investigate efficacy of cross-architecture performance models for serverless Function-as-a-Service (FaaS) platforms. Specifically, we create and evaluate models that predict serverless function runtime for functions executed on ARM64 processors, by utilizing resource utilization profiling data from function execution on x86_64 processors. We train regression based function-specific, and also generalized performance models using Linux CPU time accounting profiling data. We evaluate accuracy of serverless function runtime predictions for both seen and unseen functions, those not included as training data. We leveraged 18 distinct serverless function workloads, including 11 seen and 7 unseen, in total encompassing over 144,000 serverless function calls. We evaluate three different generalized performance models for unseen predictions: All-in-one, where all training data is combined into one model, Resource-bound, where separate models are trained for CPU vs. I/O bound functions, and ARM-speed models, where three separate models are trained based on ARM64 relative speed vs. x86_64. Using a separate classification model, we automate selection of the appropriate ARM-speed model to make predictions. For seen workloads on ARM64 processors, we predict function runtime with a mean absolute percentage error (MAPE) of only $\sim$1.17. Using our ARM-speed generalized performance models, we predict function runtime with MAPE of only $\sim$10.29 for unseen workloads, and $\sim$3.04 for seen workloads. Our performance modeling techniques can be leveraged to support creating a broadly applicable tool that predicts serverless function runtime on ARM64 processors by profiling unseen functions on x86_64 to provide inference data for model inputs.**

***Index Terms*—FaaS, Serverless Computing, Performance Modeling, Cloud Computing, Cross-Architecture Analysis**

I. Introduction

Function-as-a-Service (FaaS) is a cloud computing delivery model that allows developers to deploy and run code in a serverless environment. FaaS platforms execute code as serverless functions, providing scalability and cost-effectiveness compared to traditional virtual machine (VM) and container hosting solutions. FaaS infrastructure automatically adapts to fluctuating workloads, relieving developers from management burden while efficiently handling traffic spikes. FaaS platform charges are based on actual resource consumption, making them particularly economical for applications with variable workloads. FaaS providers only charge for the amount of resources a function consumes, not for entire servers or VMs enabling developers to save considerable costs, especially when hosting applications with variable server utilization. By leveraging the benefits of serverless computing, developers can build and deploy robust, scalable, and cost-effective applications that are highly available and responsive to traffic spikes.

As cloud computing platforms evolve, 64-bit ARM processors are gaining traction for their energy efficiency and high performance, presenting a viable alternative to traditionally dominant x86_64 processors. Cloud providers, including Amazon Web Services (AWS), have incorporated ARM64 processors in their data centers, offering them alongside x86_64 processors. Amazon's FaaS platform AWS Lambda, for example, provides ARM64 Graviton2 processors for 20% lower cost than Intel Xeon x86_64 CPUs [1]. Despite the appealing energy and cost benefits of ARM64, challenges exist regarding adoption, including code migration, software, and tool availability necessitating additional effort in code migration and testing. Developers and organizations looking to adopt ARM processors can gain insights from understanding the performance implications of their workloads on these processors.When code migration requires extensive refactoring and developer effort, it is helpful to first understand performance and cost implications to help developers and practitioners prioritize and plan code migration efforts. Understanding ARM function runtime is also important in edge and fog environments with a FaaS delivery model that feature these processors to enable informed scheduling decisions [2]–[4].

This paper extends previous research on predicting serverless function runtime for functions executed on different x86_64 processors. In [5], the authors investigated efficacy of serverless function performance models to predict runtime of compute-bound functions on seven different x86_64 Intel Xeon processors across two cloud providers. In this paper, we extend previous work in important ways. First, in [5], only one compute-bound serverless function was leveraged to create and evaluate performance models. In this paper, we leverage eighteen distinct serverless functions from SeBS and FunctionBench [6], [7]. We also created custom serverless function wrappers to instrument Linux Sysbench workloads [8]. Our functions feature diverse resource utilization characteristics and are described in Table I and Figure 1. Using these functions, we assess performance over forty distinct time steps to analyze 720 unique function configurations with runtime spanning from $\sim$3 to 140 seconds. Second, our study investigates the efficacy of resource utilization performance modeling across CPU architectures (x86_64$\rightarrow$ARM64). In [5], performance models only predicted function runtime on processors with the same architecture (Intel Xeon x86_64). Third, in [5], performance models were only trained for specific functions. In this study, we create both function-specific and generalized serverless function performance models. Generalized models are particularly important because they can predict function runtime for unseen functions not included in the training dataset. Generalized models can be immediately reused without retraining to estimate function runtime for new functions. Here we investigate multiple and random forest regression models in isolation, or combined with Linux CPU time accounting principles that are introduced in [9]. We leverage CPU time accounting metrics, collected via the Serverless Application Analytics Framework (SAAF) as features to predict function runtime on ARM processors [10].

First, we assessed the accuracy of function-specific performance models, where function profiling data was included in training datasets. Next, we trained generalized performance models and evaluated their accuracy at predicting function runtime for unseen functions not included as training data. For generalized performance models, we evaluate three distinct modeling approaches: (1) a single combined general model using all training data, **All-in-one**, (2) a set of models trained with specific resource-bound workloads (e.g. CPU-bound, I/O-bound) known as **Resource-bound**, and (3) a set of models trained with workloads having specific ARM64 runtime behavior (e.g. faster than x86_64, slower than x86_64, or similar to x86_64) known as **ARM-speed**. To automate pairing an **ARM-speed** model with an unseen workload, we trained performance classifiers to categorize ARM64 performance relative to x86_64. This approach automates selection of the best **ARM-speed** model for function runtime prediction, enabling creation of a fully automatic tool to predict ARM64 function runtime for unseen workloads based on x86_64 profiling data. Such a tool can support developers to analyze serverless functions and prioritize code migrations to ARM64 where cost and runtime benefits are greatest. Cloud providers presently do not provide function runtime predictions, forcing developers to migrate functions and benchmark ARM64 performance.

A. *Research Questions*

This paper investigates the following research questions:

RQ-1: (Function-Specific Performance Modeling): What is the accuracy of ARM64 function runtime predictions for FaaS functions based on profiling on x86_64 processors where training data includes functions being predicted?

RQ-2: (Generalized Function Performance Modeling): What is the accuracy of ARM64 function runtime predictions for unseen FaaS functions not included as training data for models, where models are trained using carefully selected workloads having a range of resource utilization characteristics (e.g. CPU, memory, disk, network)?

RQ-3: (ARM Performance Classification): How accurate are ARM64 serverless function runtime performance classifications using classifiers trained with x86_64 profiling data? Are performance classifications (i.e. ARM-faster, ARM-slower, and ARM-similar) sufficient for pairing pre-trained models to provide runtime predictions? Which metrics best support ARM performance classification?

RQ-4: (ARM Performance Modeling without FaaS): Outside a FaaS platform, what is the accuracy of ARM64 function runtime predictions using models trained by running functions on x86_64 VMs? We seek to validate that our x86_64$\rightarrow$ARM64 runtime prediction approach is generalizable outside the context of AWS Lambda.

II. Background and Related Work

A. *Towards Adoption of ARM64 Processors for FaaS*

ARM64 servers present an enticing alternative for improving system flexibility, performance isolation, and resource utilization in serverless computing. In late 2021, AWS Lambda, a predominant public FaaS platform, began offering ARM64 processors as an alternative to x86_64. To encourage adoption, ARM64 processors are offered at a 20% discount [1]. Prior investigations have primarily benchmarked ARM64 performance for hosting FaaS platforms or workloads [11]–[15].

Xie et al. compared the use of x86_64 Intel Xeon processors to the ARM Phytium 2000+ processor for hosting serverless FaaS platforms by deploying the Kubernetes-based Knative and OpenFaaS frameworks [11]. Both frameworks were deployed using one master node and eight worker nodes with 8 virtual CPUs (vCPUs) and 16GB memory each. Xie investigated implications of cold-start initialization, auto-scaling, and performance isolation of co-located function instance containers. Xie found that ARM64 processors exhibited greater startup latency, but similar scaling responsiveness while being more susceptible to resource contention when under intense pressure from many concurrent function requests. In [12], Javed et al. compared performance of OpenFaaS, Apache OpenWhisk, and AWS Greengrass hosted on a cluster of four ARM-based Raspberry Pis, in contrast to AWS Lambda and Azure Functions. The authors used three functions where each stressed one resource (e.g., CPU, memory, and disk) and found that OpenFaaS was most suitable to deploy and run on edge devices, and that network latency to the cloud made executing functions locally faster for their use cases.

Redacted authors et al. compared x86_64 vs. ARM64 performance for a natural language processing (NLP) pipeline on AWS Lambda [13]. The pipeline runtime averaged 1.7% slower on x86_64 processors than ARM64, but this performance loss was likely due to resource contention. Running the pipeline continuously for 24 hours on both processors, the fastest observed runtime was on x86_64 (13.53% faster), while the slowest runtime was also on x86_64 (17.10% slower). Runtime variation was 3x greater on x86_64 than ARM64, while ARM64 was projected to be $\sim$21.4% less expensive for 10,000 NLP pipeline executions. Park et al. profiled performance of a deep neural network inferencing suite on

AWS Lambda using ARM64 and x86_64 processors [14]. Park tested various model optimization heuristics, finding that ARM64 optimization libraries did not deliver equivalent performance enhancements compared to x86_64. Park concluded that ARM64 hardware is not yet as efficient due to immaturity of the development ecosystem. In our earlier work, Redacted author et al. deployed 18 distinct serverless functions and compared runtime using x86_64 and ARM64 processors on AWS Lambda and found that while only 7 functions were faster on ARM64, 15 were less expensive due to favorable ARM64 pricing [15]. Function runtime variation on ARM64 was less than x86_64 potentially from ARM64's lack of hyperthreading and potentially lower resource contention from less user demand for ARM64 CPUs on AWS Lambda.

B. Towards Serverless Performance Modeling

Serverless computing platforms including FaaS and Backend-as-a-Service (BaaS) are revolutionizing how cloud-based applications are developed and deployed. A big challenge with serverless platforms is their variable performance. Schleier-Smith et al. note that serverless providers use statistical multiplexing that creates challenges in predicting how long serverless functions will take to execute, or the extent of resources they will consume [16]. Redacted author et al. observed runtimes for an identical NLP data processing workload on AWS Lambda over 24 hours varied by 45.1% on x86_64, and 14.5% on ARM64 processors [13]. This variability can be a major problem for applications that require predictable performance or that have strict service level agreements (SLAs) [17]. On serverless platforms, use of heterogeneous processors can increase runtime variability, making accurate function runtime predictions more difficult [5]. This trade-off exposes the challenge for cloud providers between maximizing resource usage and ensuring predictability. [18]

In addition to performance-related challenges with serverless computing, cost-performance trade-offs have been identified. While serverless platforms offer highly scalable and flexible architecture, they can be more expensive than traditional cloud platforms, especially for hosting long-running workloads or high volume service endpoints. Developers and organizations must carefully evaluate cost and performance requirements for serverless deployments. Predicting customer costs for serverless computing is not trivial, as understanding hosting cost implications across multiple providers is a challenging issue identified for further research [19]–[21].

Analytical performance models have long been used to predict and improve the performance of distributed computing systems. However, there is a lack of such models for serverless computing platforms. Infrastructure abstraction on commercial serverless platforms reduces observability making creation of performance models more difficult [10]. Unique characteristics and policies of serverless platforms make it difficult to apply performance models developed for other systems directly. The lack of transparency and modeling capabilities make it challenging for developers to optimize performance and cost of their serverless applications. New approaches and models are needed to accurately estimate performance and cost of serverless applications [22]–[25]. Initial efforts have focused on modeling the performance and cost of serverless functions and workflows. In [5], Redacted authors et al. modeled serverless function runtime across heterogeneous processors and memory settings. This effort considered only x86_64 processors, and leveraged a single compute-bound function in developing the overall methodology. In [26], Lin et al. predict serverless application runtime and cost distributions based on profiling functions 10,000 times on AWS Lambda. Their effort did not consider how different CPUs impact performance, but instead focused on estimating the broad range of possible runtime and cost outcomes. Up to 16 serverless functions were utilized, but details regarding these functions and the rationale for their selection is limited.

Other efforts have modeled runtime and cost of serverless workflows [25], [27]. Eismann et al. presented a methodology for predicting serverless workflow costs by modeling function response times and outputs [25]. Runan et al. used static code analysis to extract dependencies among functions to improve workflow runtime predictions [27]. In this paper, in contrast to earlier efforts that model performance for sets of static functions or workflows, we contribute generalized performance models capable of predicting function runtime for unseen functions not included in training datasets. Earlier efforts have also largely ignored performance differences from heterogeneous CPUs which are commonplace in the cloud. We contribute models that predict function performance across CPUs with different architectures.

III. Methodology

A. Benchmarking Environment

In this paper, we deployed and profiled serverless functions on AWS Lambda, Amazon's versatile FaaS platform [28]. AWS Lambda supports a wide array of programming languages, as well as the deployment of custom runtimes and container-packaged functions. AWS Lambda was chosen in our study because among commercially available serverless FaaS platforms, it is currently the only platform that offers both ARM64 and x86_64 processors [1], [13].

To profile serverless functions on AWS Lambda run on both x86_64 and ARM64 processors, we utilized the Serverless Application Analytics Framework (SAAF) [10], [29]. SAAF helps developers characterize workload performance, resource utilization, and infrastructure and is instrumental in understanding and optimizing serverless application performance. SAAF supports collection of 48 distinct metrics to profile function CPU, memory, and I/O utilization while monitoring infrastructure state (e.g., cold vs. warm). SAAF supports reproducible testing to enable repeatable measurements of function performance and scalability over time, across different platforms and configurations. SAAF supports numerous programming languages, seamlessly integrating with serverless function packages, making it deployable across various commercial and open-source FaaS platforms. By enhancing observability of function deployments, SAAF plays a crucial role in enabling serverless function runtime predictions [5].

B. Serverless Functions

To support our research, in this paper, we leveraged the 18 serverless functions described in table I. We utilized functions from FunctionBench, which provides a diverse set of functions tailored for benchmarking FaaS platforms [7]. We deployed four FunctionBench functions including: linpack, json_dumps, chameleon, and float. In addition, we also leveraged the Serverless Benchmark Suite (SeBS), which provides functions for benchmarking FaaS platforms [6]. We leveraged five SeBS functions including: graph-pagerank, graph-mst, graph-bfs, compression, and video-processing.

We created four custom serverless functions to wrap existing Linux performance benchmarks. Three functions were created by wrapping Linux Sysbench benchmarks: primenumber (sysbench-cpu), thread (sysbench-threads), and readmemory (sysbench-memory) [8]. The readdisk function was created by wrapping the fio - flexible I/O tester benchmark [30].

We created five new serverless functions to stress specific aspects of the system that are not covered by other benchmarks. Chacha20 used the openssl encryption libraries to encode an 8MB file n times [31]. For chacha20, we disabled acceleration, Neon on ARM, and AVX on x86_64 in our testing. Sqlite executes random queries against an embedded SQLite file-based database. Filehandle opens and closes a scalable number of file handles. We scaled the number of file handles from 100,000 to 12,000,000 to scale the function's runtime. The socket function opened and closed a socket n times to scale runtime. Readwritememory performed n iterations of creating a 1 KB byte array, writing 1,024 bytes, and deleting the array in memory. Collectively, we used these 18 functions because they provide a diverse range of CPU profiles, as shown in figure 1. Two functions used multiple threads, while five others featured considerable CPU kernel time consistent with a high volume of I/O operations and/or kernel API calls. Our functions encompassed tasks ranging from compute-intensive to I/O-bound workloads, enabling us to build and evaluate performance models for a broad range of use cases to consider serverless system performance for both architectures.

C. Model Development and Evaluation

1) Predicting Serverless Function Runtime: For our runtime prediction models, we employed three distinct approaches: simple linear regression (SLR) using only runtime, multiple linear regression (MLR), and Linux CPU time accounting (LTA). Multiple linear regression and Linux CPU time accounting used cpuUser, cpuKernel, and cpuIdle as features. We implemented each approach with simple/multiple linear regression (SLR, MLR, LTA) and random forest (SLR-RF, MLR-RF, LTA-RF) to compare accuracy.

Runtime Linear Regression (SLR, SLR-RF): This method performs simple linear regression using runtime data from function executions without incorporating additional features

TABLE I
FUNCTION DESCRIPTIONS AND SHORT NAMES GROUPED BY THE PREDOMINANTLY STRESSED RESOURCE.

	Function Name	Source	Description
cpuUser	chacha20*†	openssl	Repeatedly perform openssl encryption of 8MB file n times
	graph-bfs†	sebs	Breadth-first search (BFS) implementation with igraph.
	graph-mst†	sebs	Minimum spanning tree (MST) implementation with igraph.
	graph-pagerank†	sebs	PageRank implementation with igraph.
	primenumber*†	sysbench	Prime number generator
	chameleon	FunctionBench	Create HTML table of n rows and M columns
	csv	[redacted] [32]	Generates a large CSV file and performs calculates on columns.
	float	FunctionBench	Perform sin, cos, sqrt ops
	json_dumps	FunctionBench	JSON deserialization using a downloaded JSON-encoded string dataset
	sqlite	original	Execute n random SELECT queries on a 10*1000 SQLite database
	video-processing*	sebs	Convert PNG to GIF n times
cpuKernel	filehandle†	original	Open and close file handles
	socket†	original	Open and close socket n times
	thread†	sysbench	Create thread, put locks and release thread
Memory	readmemory*†	sysbench	N sequential reads of 1GB memory block
	readwritememory†	original	Allowcate 1MByte of memory, write 0x42 into it and release
I/O	readdisk*†	fio	Test random read speed on a 1GB block
	compression	sebs	Create a .gz file for a file

cpuUser group: Runtime dominated by CPU user time (blue), **cpuKernel group**: Runtime with higher CPU kernel time (yellow), **Memory group**: Workload is memory intensive (orange), and **I/O group**: Workload is I/O intensive (grey).
*: Function executes external binary program (non-Python)
†: Function used to train models

as predictors. It serves as a baseline to assess the effectiveness of more complex modeling approaches.

Multi-Regression Analysis (MLR, MLR-RF): This approach integrates multiple features derived from SAAF profiling, specifically CPU User, CPU Kernel, and CPU Idle time into regression models. We focused on these features, because profiling serverless functions on AWS Lambda indicated little to no CPU time spent in other modes.

Linux CPU Time Accounting (LTA, LTA-RF): This approach leverages Linux CPU time accounting to incorporate CPU timing metrics to aid performance modeling [5], [9]. The premise of Linux CPU time accounting is that for every second of function runtime, each vCPU provides one second of CPU time divided across all CPU modes. Linux CPU time accounting captures CPU time spent in distinct modes: **CPU User** is when the CPU executes code in user mode, **CPU Kernel** is when the CPU executes in kernel mode, **CPU IO wait** is when the CPU waits for I/O to complete, **CPU Sft Int**

Srvc is when the CPU waits for soft interrupts, and **CPU Idle** is when the CPU is idle. We model each CPU time component independently to improve accuracy. We capture the CPU profile on one platform, and create models to predict the CPU profile on a target platform. Profiling effectiveness can be verified because for every second, the total observed CPU time for all modes combined must equal 1 second for each vCPU. Linux CPU time accounting can provide improvements over traditional regression-based performance models because it enables modeling each component of workload behavior separately (i.e. user code, kernel code, I/O wait, interrupts, and idle time). In this paper, we apply Linux CPU time accounting to build x86_64 to ARM64 serverless function performance models using multiple features derived from SAAF profiling to generate multiple linear regression models to predict CPU profile metrics (e.g. CPU User, CPU Idle, etc.). A total of 21 features (e.g. totalMemory, freeMemory, pageFaults, frameworkRuntime, userRuntime, cpuUserDelta, cpuNiceDelta, cpuKernelDelta, cpuIdleDelta, cpuIOWaitDelta, cpuIrqDelta, cpuSoftIrqDelta, cpuStealDelta, cpuGuestDelta, cpuGuestNiceDelta, pageFaultsDelta, availableCPUs, utilizedCPUs, recommendedMemory, frameworkRuntimeDeltas, and runtime) were used to predict each CPU metric.

We evaluated the accuracy of our models against actual observed runtimes on AWS Lambda with ARM64 processors. For each of our 18 serverless functions, we collect a comprehensive dataset to characterize function runtime. We scale function runtime up from ~3 to 140 seconds in 40 distinct steps by adjusting a configuration parameter. The parameter adjusts the number of iterations or the volume of work a function performs to enable profiling function runtime over an increasing time span. For each step, we collect 100 runtime samples, for a total of 4,000 samples for each

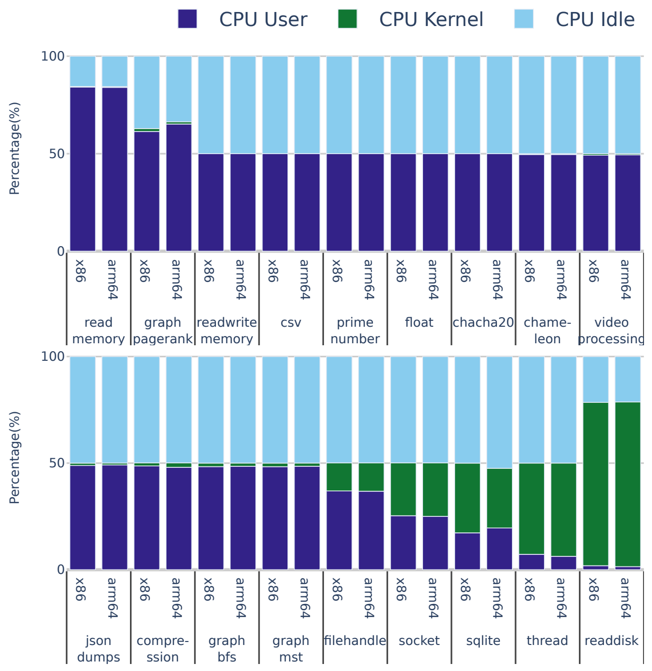


Fig. 1. CPU mode time percentage of each function

CPU (i.e. ARM64, x86_64) per function. To reduce x86_64 runtime variance, we only use runtime samples from the Intel Xeon 8259CL x86_64 processor identified by its 2.5 GHz clock and 36608KB cache size, and perform additional runs when failing to obtain this processor. Recently three distinct x86_64 CPUs were identified on AWS Lambda [32]. For this paper, we performed over 115,000 x86_64 function calls and identified 5 distinct x86_64 processors (see III.D). To establish ground truth for function runtime on ARM64, we observe the inference sample's percentile position in the x86_64 dataset, and map this to the equivalent percentile position in the corresponding ARM64 dataset. This method allows us to estimate an expected ARM64 runtime to pair with each x86_64 runtime observation. We then evaluate performance model accuracy by calculating the mean absolute percentage error (MAPE) across all observations. For runtime prediction in cloud computing, we note that ground truth must also be estimated because all samples are simply observations that fall somewhere along a distribution. Runtime of identical serverless function calls always exhibits some variability.

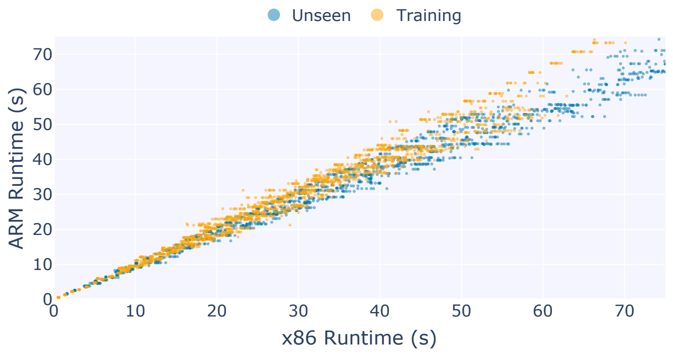


Fig. 2. Generalized Performance Models Input Space Coverage: Serverless Function Runtime - Training vs. Testing Data

2) Generalized Serverless Function Performance Models: We investigated creating generalized performance models to predict ARM64 serverless function runtime for any serverless function. While prior efforts at predicting function runtime have focused on predicting runtime only for observed workloads [5], [25], in this paper, we investigate performance models to make runtime predictions for unseen functions, not included in model training datasets. Generalized performance models are needed to create a tool that predicts function runtime on ARM64 processors for unseen functions that can help developers prioritize ARM64 function migration decisions. While creating models that predict function runtime for unseen workloads with perfect accuracy is likely intractable, we seek to create models that can provide ”good” estimates of runtime for unseen functions, sufficient to be of value to developers and practitioners.

To train generalized performance models, we used a diverse set of 11 functions having different resource utilization characteristics with a runtime input space spanning from ∼ 3 to 140 seconds as described in table I and III. Figure 2 depicts training vs. unseen function runtime overlap for our datasets up to 75 seconds. Specifically, we used the functions: primenumber,

readmemory, readdisk, chacha20, readwritememory, filehandle, thread, graph-pagerank, graph-mst, graph-bfs, and socket to train generalized models. We investigated three distinct approaches for creating generalized performance models:
All-in-one: uses all available training workloads to train a combined model to predict ARM64 runtime for unseen functions. The advantage with this approach is that there is only one model, so it is easy to train and use to make predictions.
Resource-bound: involves creating resource-specific models trained with a subset of the available workloads having particular characteristics. For our **Resource-bound** approach, we created separate CPU-User and CPU-Kernel bound models. Functions with more than 10% CPU kernel time were used to train the CPU-Kernel model; otherwise, the remaining functions were used to train the CPU-User model.
ARM-speed: involves training a series of performance models by combining workloads with similar ARM64 runtime performance relative to x86_64. We created three models by combining training workloads together based on their performance behavior: ARM-faster, ARM-slower, and ARM-similar. Here, ARM-faster indicates that ARM64 had $>15\%$ faster runtime than x86_64, ARM-slower meant ARM64 had $>15\%$ slower runtime, and for ARM-similar, ARM64 runtime was within +/-15% of the x86_64 runtime.

3) Classification Models for Characterizing ARM64 Performance to Support ARM-speed Model Selection: Our ***ARM-speed*** generalized performance modeling approach involved training separate models to predict serverless function runtime for workloads exhibiting ARM-faster, ARM-slower, or ARM-similar performance. Selecting the best ***ARM-speed*** model for an unseen workload represents a new problem. It is necessary to rapidly identify the best generalized model (i.e. ARM-faster, ARM-slower, or ARM-similar) to make runtime predictions for new unseen serverless workloads. To address this challenge, we trained classification models to classify ARM function runtime into the three categories (i.e. faster, slower, similar) using x86_64 profiling data. The classification can then be used to select the best generalized ***ARM-speed*** model for runtime predictions. A total of 21 features were used to generate the prediction models. We explored the following classification algorithms: *Random Forest Classifier*, *Ada Boost Classifier*, *MLP (Multi-Layer Perception) Classifier*, *Decision Tree Classifier*, *KNeighbors Classifier*, *Gaussian Process Classifier*, and *Quadratic Discriminant Analysis*. Each classifier was trained using x86_64 and ARM64 performance data with ground-truth performance categories assigned using the 15% thresholds described above. When collecting training and test data, we filtered out cold function calls. Cold calls were not included because they are notoriously slower than warm calls, introducing runtime variability that can skew the model's predictions. We tested our classifiers ability to accurately classify ARM function runtime into subgroups: ARM-faster, ARM-slower, or ARM-similar. While our primary aim was to identify classifiers that could infer the performance subgroup of unseen workloads (**RQ-3**), we note our classifiers can also be applied to help developers rapidly identify workloads most likely to benefit from faster execution on ARM64 processors, i.e. workloads classified as ”ARM-faster”.

TABLE II
CPU MODEL COMPARISON: AWS LAMBDA X86_64 AND ARM64 CPUS

CPU	**AWS Graviton 2**	**Platinum 8259CL**
Clock Speed:	2.5 GHz	2.5/3.5/1.2 GHz base/turbo/low
Cores:	64 (1 Socket)	24 (48 Threads) (8 Socket)
Core/Arch:	Neoverse N1	Cascade Lake-SP
TDP	80-110w	210w
Node:	TSMC 7nm	Intel 14nm
Cache:	48K instruc/c, 64K data/c 1M L2/c, 32M L3	32k instruc/c, 32k data/c 1M L2/c, 35.75M L3
Memory:	DDR4 8 channel/chip	DDR4 6 channel/chip
CPU Cluster:	4	1

4) Tools and Frameworks: We used the sklearn Python library to create and evaluate models [33]. Sklearn offers many tools and algorithms for machine learning, making it well suited for performance modeling. It provides functionalities for data prepossessing, model training, cross-validation, and performance evaluation, which are crucial for the robust development and assessment of our prediction models.

5) Evaluation Metrics: To evaluate runtime models, we employed standard evaluation metrics including MAPE and R-squared (R^2) for regression models. For our classification models we performed ternary classification and evaluated accuracy as a percentage. These metrics provide a comprehensive view of model performance, allowing us to assess, not only the accuracy of predictions, but also the models' ability to generalize across different workloads and conditions.

D. Experimental Approach

Our analysis was conducted on AWS Lambda in the us-west-2 (Oregon) region. We provisioned our AWS Lambda functions with 3 GB memory to ensure full access to 2 vCPUs [34]. This is the smallest memory size that allows full access to 2 vCPUs. Any smaller memory size results in a fractional share of CPU time equivalent to less than 2 vCPUs. Under-provisioning vCPUs could lower performance and increase runtime performance variance, skewing the results of our comparisons. Functions were tested using identical configurations (e.g. memory and vCPUs) on x86_64 and ARM64 to ensure consistency for accurate benchmarking and analysis. Currently, there are no other commercially available FaaS platforms using ARM processors, limiting our study to AWS Lambda.

During our study we discovered that AWS Lambda employed a variety of different x86_64 compatible CPUs to execute functions. By analyzing over 115,000 x86_64 function calls made in April 2024 on us-west-2, we identified five different x86_64 CPU types. Notably, we observed x86_64 function executions on the following CPUs:

- Intel Xeon 8529CL 2.50GHz 36608KB Cache (91.1515%)
- Intel Xeon 8275CL 3.00GHz 36608KB Cache (6.5606%)
- Intel Xeon 8375C 2.90GHz 55296KB Cache (2.0984%)
- AMD EPYC 2.25GHz (0.0985%)
- AMD EPYC 2.65GHz (0.0909%)

To minimize x86_64 serverless function runtime variance we concentrated our modeling efforts exclusively on the Intel Xeon Platinum 8259CL CPU, featuring a 2.50GHz base frequency and 36,608 KB cache as described in table II. To identify the specific CPU in use on AWS Lambda, we first matched the most likely CPU based on clock speed and cache size to those offered on Amazon EC2 VMs. Using Amazon EC2, we then installed the Firecracker microVM hypervisor on an m5dn.metal ec2 instance. We applied the T2CL Firecracker CPU template, a template that supports creating microVMs using different x86_64 processors while masking some of the CPU differences to create a homogenized Intel Cascade Lake compatible microVM [35]. Amazon has developed this technique to mask x86_64 CPU differences to allow mixing, for example, Cascade Lake and Ice Lake processors while providing the illusion that all CPUs are Cascade Lake on AWS Lambda. Using the T2CL template on our firecracker microVM, we were able to precisely match Linux cpuinfo and CPU flags observed on AWS Lambda on our own microVM to confirm the exact x86_64 CPU most commonly used on AWS Lambda (Xeon 8259CL). For this paper, we focus exclusively on the Xeon 8259CL processor because it was the most common CPU observed in over 90% of x86_64 serverless function executions on AWS Lambda. We filtered out all other CPUs from our datasets to homogenize our x86_64 profiling data to one CPU. For each workload, we performed 100 runs across each of 40 steps on this CPU for a total of 4,000 runs. Filtering other x86_64 CPUs required increasing the total number of profiling runs by about $\sim$10%.

For I/O operation tests on serverless functions, we provisioned a 5 GB ephemeral disk to increase available /tmp space [36]. An ephemeral disk is a temporary cloud disk provided to the function instance for an additional charge. We deployed the Bonnie++ disk I/O benchmark to compare AWS Lambda disk I/O performance on x86_64 and ARM64 processors. X86_64 functions exhibited marginally lower I/O performance for sequential input per character but performed comparably in other categories vs. ARM64 [37]. ARM64 however, demonstrated reduced latency in multiple scenarios. Bonnie++ performance was consistent on AWS Lambda for both 3GB and 10GB function memory configurations. We conclude that I/O performance on ARM64 has lower latency than x86_64 but overall throughput is quite similar.

We scaled up function runtime using 40 distinct steps resulting in average runtime spanning from approximately $\sim$3 seconds to 140 seconds on x86_64 processors. Each function was profiled 100 times per step on both processors. This approach enabled us to profile functions across a wide range of runtimes, from a few seconds to several minutes, providing a thorough understanding of how each architecture responded for different workloads and durations.

To extend our evaluation beyond AWS Lambda, because we know of no other commercial FaaS platform having ARM64 support, for **(RQ-4)**, we investigated predicting function runtime on ARM64 processors based on x86_64 profiling using Amazon EC2 virtual machine instances. This evaluation helps us verify if our x86_64$\rightarrow$ARM64 performance modeling approach can be generalized for use outside of AWS Lambda on self-hosted FaaS platforms or on future commercial FaaS platforms with ARM64 support. For these tests, we utilized the c5.xlarge (x86_64 architecture) and m6g.xlarge (ARM64 architecture) Amazon EC2 instances types. These instances allowed us to train performance models using function profiling data from x86_64 and ARM64 VMs to predict function runtime on ARM64 VMs.

A subset of our benchmarks were selected including: chacha20, primenumber, and graph-pagerank. These workloads were chosen as they were primarily CPU-bound workloads with runtime dominated by CPU User mode time. We trained function-specific performance models for each function using the same approach as for **(RQ-1)**.

To train and evaluate performance models on EC2, each function utilized the same 40 distinct steps as on AWS Lambda with 50 runs per step on both processors for a total of 4,000 calls per function.

IV. Results and Evaluation

A. *ARM Performance Modeling*

To investigate **(RQ-1)**, we leveraged 11 serverless functions with different ARM64 runtime behavior shown in figure 3 to train and evaluate function-specific performance models. We trained models to predict ARM64 function runtime using data from profiling functions on x86_64 processors. We trained models for each of the functions: chacha20, primenumber, readdisk, filehandle, readwritememory, readmemory, socket, graph-pagerank, graph-mst, graph-bfs, and thread.

Our analysis revealed that function-specific models demonstrated very good accuracy. We calculated average MAPE for function-specific performance models trained for the first 11 functions shown in Table III. Simple linear regression (SLR) achieved average MAPE of 2.67. Multiple linear regression (MLR) achieved lower average MAPE of 1.77 with an R-squared value (R^2) of $\sim$0.99. Linux CPU time accounting (LTA) improved accuracy further to 1.36 MAPE. **Random forest multiple regression (MLR-RF) provided the lowest average MAPE of just 1.17**, while using random forest for Linux CPU time accounting (LTA-RF) resulted in higher MAPE of 1.20. We found that all of our random forest regression models attained an R^2 of 0.99. Overall, function-

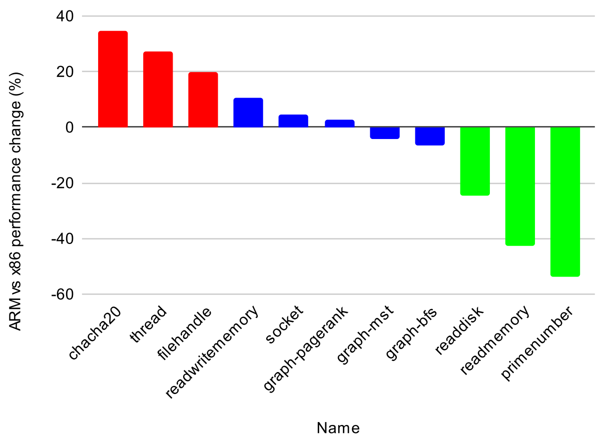


Fig. 3. ARM64 vs. x86_64 Function Performance of Training Workloads

TABLE III
TRAINING AND TESTING FUNCTION'S RUNTIME, COEFFICIENT OF VARIATION (CV), AND MEAN ABSOLUTE PERCENTAGE ERROR (MAPE)

Function name	**Min runtime** x86_64 (sec)	**Min runtime** ARM64 (sec)	**Max runtime** x86_64 (sec)	**Max runtime** ARM64 (sec)	**CV(%)** x86_64	**CV(%)** ARM64	**MAPE** fn-specific$^{1\,2}$	**MAPE** All-in-One^{1}	**MAPE** ARM-speed^{1}
primenumber	6.00	5.27	120.92	108.73	0.72	0.58	0.83	28.55	0.18
readmemory	3.15	3.85	132.68	106.40	2.17	3.51	1.2	7.02	2.15
readdisk	6.77	7.46	135.01	114.11	2.05	1.79	2.17	16.47	1.76
chacha20	4.70	4.53	118.90	144.54	0.73	0.23	0.2	27.93	7.42
readwritememory	5.08	3.89	123.16	134.82	1.28	2.32	1.44	8.93	5.41
filehandle	4.69	8.87	109.33	132.41	1.88	0.95	2.84	5.26	2.49
thread	4.46	5.38	128.17	135.75	0.63	0.56	0.96	18.82	1.75
graph-pagerank	5.58	6.15	58.69	61.45	0.60	0.57	0.98	9.32	2.15
graph-mst	6.83	3.40	65.03	56.15	0.63	0.56	0.96	3.05	2.46
graph-bfs	4.25	8.77	64.10	67.49	0.94	0.84	0.39	4.02	3.94
socket	7.82	6.91	125.99	130.18	2.31	3.08	0.97	1.51	3.72
video-processing	3.01	3.17	139.54	135.75	0.42	1.07	1.79	25.26	8.32
json dumps	5.30	8.71	128.80	134.02	1.59	1.45	0.64	5.23	7.83
sqlite	6.28	4.25	134.92	121.42	1.06	0.82	0.97	18.79	6.96
chameleon	5.12	8.29	112.96	101.62	1.09	0.74	1.13	13.07	10.60
compression	8.21	7.48	135.76	122.41	1.80	0.46	0.52	15.26	11.93
float	4.19	8.63	122.40	135.99	3.26	2.14	0.85	24.04	14.30
csv	8.87	8.90	136.81	124.68	1.22	0.94	2.17	29.72	12.10
Avg-training	5.39	5.86	107.45	108.37	1.27	1.36	1.17	11.90	3.04
Avg-unseen	5.85	7.06	130.17	125.13	1.49	1.09	1.15	18.77	10.29
Average	5.57	6.33	116.29	114.88	1.35	1.26	1.16	14.57	5.86

^{1}-random forest regression w/ multi-features, ^{2}-evaluated w/ 2nd independent 4k sample/fn dataset

specific models consistently predicted ARM64 runtime function runtime with high accuracy (<3% mean error) when leveraging profiling data from functions run on x86_64.

B. Generalized ARM Performance Modeling

For **(RQ-2)**, we trained generalized performance models to predict ARM64 function runtime for unseen workloads not included in the training dataset. We trained these models using functions with runtime spanning from $\sim$3 to 140 seconds, the first 11 functions at the top of table III. We investigated three different modeling approaches for predicting unseen function runtime on AWS Lambda with ARM64 processors, based on x86_64 profiling data. Each modeling approach investigated a different method for generalized performance modeling.

All-in-one Model: This modeling approach aggregates all training data into a single common model providing the advantage of simplicity. The user needs only to perform simple inferencing with a single model to generate a runtime prediction. The one-model-fits-all approach, however, may not capture the nuances of specific resource-intensive workloads as effectively as other approaches.

Resource-bound Model: We investigated training two distinct models based on the workload's primary resource requirement: CPU-User intensive (i.e. primarily runs user intensive code) and CPU-Kernel intensive (i.e. >10% CPU time spent executing kernel instructions - indicating intensive I/O or kernel API use) as shown in figure 1. This simple delineation aimed to improve accuracy of runtime predictions for functions with clear CPU utilization differences. Nevertheless, our results for this approach did not provide good ARM64 runtime prediction accuracy. In fact, this approach was less accurate than the **All-in-one** model.

ARM-speed Model: We investigated training a set of three models, known as the **ARM-speed** models, which group together training workloads having similar runtime behavior. We investigate whether models are more accurate when the input space is constrained to workloads with similar runtime behavior. Training functions are grouped into the models:

- *ARM-faster:* ARM64 runtime $\geq$ 15% faster than x86.
- *ARM-slower:* ARM64 runtime $\leq$ 15% slower than x86.
- *ARM-similar:* ARM64 and x86_64 runtime within +/-15%.

While defining more performance categories is possible, categorization is limited by the need to have a critical mass of workloads in each category. Applying ARM-speed models to generate runtime predictions for unseen workloads additionally requires identifying the best models to pair with unseen workloads to generate runtime predictions. This challenge is addressed by **(RQ-3)**, where we investigate classification models to support automatically selecting the best **ARM-speed** model. We used readwritememory in both the ARM-slower and ARM-similar models because its runtime performance was right on the threshold between ARM-slower and ARM-similar.

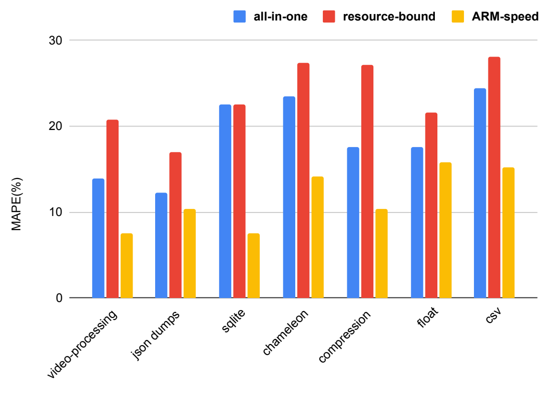


Fig. 4. Mean Absolute Percentage Error for Unseen workloads

Figure 4 shows average MAPE for unseen workloads using our three generalized modeling approaches. **ARM-speed** models provided the best accuracy in contrast to **All-in-one** and **Resource-bound** models for ARM64 serverless function runtime prediction. By categorizing unseen function runtime into

TABLE IV
CLASSIFIER ACCURACY COMPARISON

Classfier	**Accuracy** (%)
Random Forest Classifier	93.35
DecisionTree Classifier	91.65
Gaussian Process Classifier	83.63
AdaBoost Classifier	78.78
KNeighbors Classifier	74.55
MLP Classifier	65.83
Quadratic Discriminant Analysis	62.05

a specific performance group (e.g. ARM-faster, ARM-slower, and ARM-similar) and then making runtime predictions using a category specific model, our **ARM-speed** models helped tailor predictions more closely by considering their unique performance behavior. Detailed results of ARM64 runtime prediction for all eighteen functions is shown in table III.

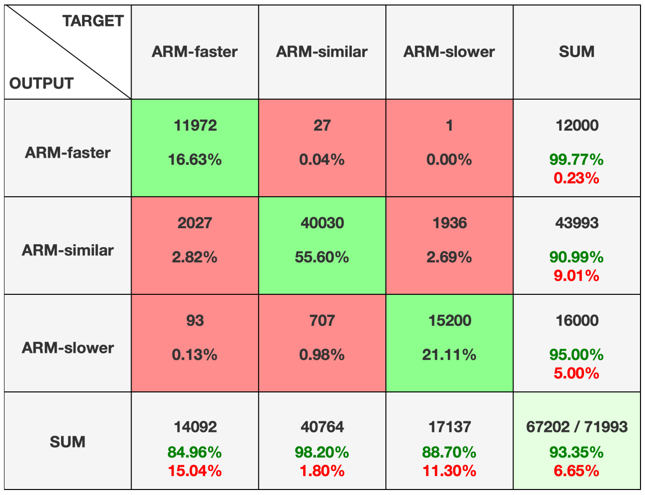
C. Unseen Workload Runtime Classification

The **ARM-speed** models require pairing unseen workloads with the appropriate generalized model trained to generate predictions for their specific runtime behavior (i.e. ARM-faster, ARM-slower, and ARM-similar). For **(RQ-3)**, we investigated classification models shown in table IV to automatically classify the ARM64 runtime category based on x86_64 profiling.

TABLE V
RANDOM FOREST CLASSIFIER - STRONGEST FEATURES

Feature	**Importance**	**Information Gain**
pageFaultsDelta	0.1227	0.2177
utilizedCPUs	0.0730	0.0882
freeMemory	0.0700	0.1499
cpuUserDelta	0.0680	0.0968
cpuKernelDelta	0.0652	0.1049
frameworkRuntime	0.0063	0.1355

Feature importance shown is permutation importance. pageFaultsDelta: # of pagefaults, frameworkRuntime: profiling time spent by SAAF, utilizedCPUs: estimated CPUs actually used by the function, freeMemory: free memory space in megabytes, cpuUserDelta: CPU user mode time during function execution, cpuKernelDelta: CPU kernel time during function execution



TARGET / OUTPUT	ARM-faster	ARM-similar	ARM-slower	SUM
ARM-faster	11972 16.63%	27 0.04%	1 0.00%	12000 99.77% 0.23%
ARM-similar	2027 2.82%	40030 55.60%	1936 2.69%	43993 90.99% 9.01%
ARM-slower	93 0.13%	707 0.98%	15200 21.11%	16000 95.00% 5.00%
SUM	14092 84.96% 15.04%	40764 98.20% 1.80%	17137 88.70% 11.30%	67202 / 71993 93.35% 6.65%

Fig. 5. Workload Performance Classification Confusion Matrix

Accuracy of our different performance classifiers is shown in table IV, table V describes important features, while Figure 5 provides a ternary confusion matrix detailing our random forest classifier accuracy. Our random forest classification model offered 93.35% accuracy when classifying expected ARM64 performance using individual x86_64 function runs as model input. With this high single sample accuracy, for unseen workloads, classifying 10 function runtime samples and then assuming the most common classification is correct reduces the probability of misclassifying performance to just 0.025%. Only a small number of test samples (i.e. 10) are needed to quickly classify unseen function ARM64 performance. The 'chameleon' workload provided a more challenging scenario with single sample classification accuracy of just ∼60%. In this challenging scenario, by expanding the analysis to 100 samples, and then assuming the most common classification is correct, the probability of misclassification is just ∼2.7%.

Supported by our random forest classifier, our **ARM-speed** models can generate ARM64 runtime predictions with average MAPE of 10.29 for unseen functions, and 5.86 for all functions. This marks a promising advancement towards providing a methodology to build an automated tool to predict ARM64 serverless function runtime based on profiling on X86_64 processors. As future work, we plan to expand the number of training functions used in generalized models. Enriching the datasets to cover a broader input space has potential to enhance our ability to classify a larger array of serverless functions to support improved ARM64 runtime predictions.

D. Performance Estimation on Non-FaaS Platforms

To investigate **(RQ-4)**, we conducted experiments using AWS EC2 instances, specifically c5.xlarge (x86_64 architecture) and m6g.xlarge (ARM64 architecture). We chose these VM types because they enabled us to train a VM performance model for an Intel Xeon x86_64 CPU to predict runtime on the Graviton2 ARM64 CPU. We selected three functions for this purpose: chacha20, primenumber, and graph-bfs. We ran function code inside the Podman containers restricted to 2 vCPUs to evaluate function-specific x86_64→ARM64 performance predictions using the same approaches as with **(RQ-1)**. Due to the significant time and cost involved in profiling functions on VMs, for each function we performed 50 runs (not 100) across 40 steps using 20 VMs in parallel, resulting in 6,000 total profiling runs for each CPU architecture. This is half of the profiling data compared to function-specific models trained on AWS Lambda for **(RQ-1)**. For our three functions average MAPE was 1.41 for ARM64 runtime predictions on EC2. This included chacha20 (MAPE: 0.93), primenumber (MAPE: 1.87), and graph-pagerank (MAPE: 1.42). These results demonstrate that our x86_64 to ARM64 performance modeling approach is robust and adaptable in contexts outside AWS Lambda. Future work can investigate function runtime predictions in open-source FaaS environments like Apache OpenWhisk and OpenFaaS deployed on ARM64 hardware.

V. CONCLUSIONS

In this paper, we present function-specific and generalized performance models to support predicting ARM64 serverless function runtime. For **(RQ-1)**, using random forest regression models we demonstrated our ability to predict ARM64 serverless function runtime with on average only ∼1.17 MAPE using x86_64 profiling data. For **(RQ-2)**, we investigated three approaches for training generalized performance models to predict ARM64 function runtime for unseen workloads not included in model training datasets. Our **ARM-speed** model provided ARM64 runtime predictions with only 3.04 MAPE for training functions, 10.29 MAPE for unseen functions, and 5.86 MAPE for all functions on average. To automatically select the best **ARM-speed** model to make ARM64 runtime predictions for unseen functions **(RQ-3)**, we trained a series of classifiers to predict the behavioral category (i.e. ARM-faster, ARM-slower, or ARM-similar). Our random forest classifier achieved 93.35% accuracy at predicting the function's performance category using a single function profiling sample. By taking the most common classification using a set of samples (e.g. 10 to 100), we are able to reliably pair an **ARM-speed** model to an unseen function. For **(RQ-4)**, we demonstrated generalizability of our approach by reproducing ARM64 function runtime predictions using EC2 VMs achieving 1.41 MAPE with only half the volume of training data. Our approaches form the basis to create an automated tool to predict ARM64 serverless function runtime for unseen workloads based on x86_64 profiling. Our results can help developers and practitioners prioritize serverless function migrations to ARM64 processors. Function and modeling code used in our experiments is available on Github [38].

References

[1] D. Poccia, "AWS Lambda Functions Powered by AWS Graviton2 Processor – Run Your Functions on Arm and Get Up to 34 Accessed: 2023-09-30.

[2] G. A. S. Cassel and et al., "Serverless computing for internet of things: A systematic literature review," *Future Generation Computer Systems*, vol. 128, pp. 299–316, 2022.

[3] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *2020 IEEE Intl. Conf. on Fog Comp. (ICFC)*, pp. 17–24, IEEE, 2020.

[4] A. Tzenetopoulos and et al., "Faas and curious: Performance implications of serverless functions on edge computing platforms," in *High Perf. Comp.: ISC High Perf. Digital 2021 Intl. Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*, pp. 428–438, Springer, 2021.

[5] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting performance and cost of serverless computing functions with saaf," in *2020 IEEE Intl Conf on Cloud and Big Data Computing*, pp. 640–649, IEEE, 2020.

[6] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proc. of the 22nd Intl. Middleware Conf.*, pp. 64–78, 2021.

[7] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th Intl. Conf. on Cloud Comp. (CLOUD)*, pp. 502–504, IEEE, 2019.

[8] A. Kopytov, "man sysbench (1): A modular, cross-platform and multi-threaded benchmark tool." https://manpages.org/sysbench. Accessed: 2023-09-30.

[9] W. J. Lloyd and et al., "Demystifying the clouds: Harnessing resource utilization models for cost effective infrastructure alternatives," *IEEE Trans. on Cloud Comp.*, vol. 5, no. 4, pp. 667–680, 2015.

[10] 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 ACM/SPEC Int. Conf. on Perf. Engineering*, pp. 161–164, 2021.

[11] D. Xie, Y. Hu, and L. Qin, "An evaluation of serverless computing on x86 and arm platforms: Performance and design implications," in *IEEE 14th Intl. Conf. on Cloud Comp. (CLOUD)*, pp. 313–321, IEEE, 2021.

[12] H. Javed, A. N. Toosi, and M. S. Aslanpour, "Serverless platforms on the edge: a performance analysis," in *New Frontiers in Cloud Comp. and Internet of Things*, pp. 165–184, Springer, 2022.

[13] D. Lambion, R. Schmitz, R. Cordingly, N. Heydari, and W. Lloyd, "Characterizing x86 and arm serverless performance variation: a natural language processing case study," in *Companion of the 2022 ACM/SPEC Int. Conf. on Perf. Engineering*, pp. 69–75, 2022.

[14] S. Park, J. Choi, and K. Lee, "All-you-can-inference: serverless dnn model inference suite," in *Proc. of the Eighth Intl. Workshop on Serverless Comp.*, pp. 1–6, 2022.

[15] X. Chen, L.-H. Hung, R. Cordingly, and W. Lloyd, "X86 vs. arm64: an investigation of factors influencing serverless performance," in *Proc. of the 9th Int. Workshop on Serverless Comp.*, pp. 7–12, 2023.

[16] J. Schleier-Smith and et al., "What serverless computing is and should become: The next phase of cloud computing," *Comm. of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[17] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures," in *Companion of the 2018 ACM/SPEC Intl. Conf. on Perf. Engineering*, pp. 21–24, 2018.

[18] E. Jonas and et al., "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[19] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Comp. Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[20] E. Van Eyk and et al., "Serverless is more: From paas to present cloud computing," *IEEE Internet Comp.*, vol. 22, no. 5, pp. 8–17, 2018.

[21] A. Eivy and J. Weinman, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Comp.*, vol. 4, no. 2, pp. 6–12, 2017.

[22] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Trams. on Cloud Comp.*, vol. 10, no. 4, pp. 2834–2847, 2020.

[23] N. Mahmoudi and H. Khazaei, "Performance modeling of metric-based serverless computing platforms," *IEEE Trans. on Cloud Comp.*, vol. 11, no. 2, pp. 1899–1910, 2022.

[24] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. on Parallel and Dist. Sys.*, vol. 32, no. 3, pp. 615–632, 2020.

[25] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in *Proc. of the ACM/SPEC Intl. Conf. on Perf. Engineering*, pp. 265–276, 2020.

[26] C. Lin, N. Mahmoudi, C. Fan, and H. Khazaei, "Fine-grained performance and cost modeling and optimization for faas applications," *IEEE Trans. on Parallel and Dist. Sys.*, vol. 34, no. 1, pp. 180–194, 2022.

[27] R. Wang, G. Casale, and A. Filieri, "Enhancing performance modeling of serverless functions via static analysis," in *Intl. Conf. on Service-Oriented Comp.*, pp. 71–88, Springer, 2022.

[28] Amazon Web Services, "AWS Lambda." https://aws.amazon.com/pm/lambda/. Accessed: 2023-09-30.

[29] R. Cordingly and et al., "The serverless application analytics framework: Enabling design trade-off evaluation for serverless software," in *Proc. of the 2020 Sixth Int. Workshop on Serverless Comp.*, pp. 67–72, 2020.

[30] J. Axboe, "1. fio - flexible i/o tester rev. 3.35; fio 3.35-6-g1b4b-dirty documentation." https://fio.readthedocs.io/en/latest/fio_doc.html. Accessed: 2023-09-30.

[31] openssl.org, "Enc - opensslwiki." https://wiki.openssl.org/index.php/Enc. Accessed: 2023-09-30.

[32] R. Cordingly, J. Kaur, D. Dwivedi, and W. Lloyd, "Towards serverless sky computing: An investigation on global workload distribution to mitigate carbon intensity, network latency, and cost," in *2023 IEEE Int. Conf. on Cloud Engineering (IC2E)*, pp. 59–69, IEEE, 2023.

[33] F. Pedregosa and et al., "Scikit-learn: Machine learning in python," vol. 12, pp. 2825–2830, 2011.

[34] R. Cordingly, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with cpu time accounting," in *2022 IEEE Int. Conf. on Cloud Engineering (IC2E)*, pp. 104–115, IEEE, 2022.

[35] a. roypat, zulinx86 et, "firecracker cpu templates." https://github.com/firecracker-microvm/firecracker/blob/main/docs/cpu_templates/cpu-templates.md. Accessed: 2024-08-20.

[36] C. Yun, "AWS Lambda Now Supports Up to 10GB Ephemeral Storage." https://aws.amazon.com/blogs/aws/aws-lambda-now-supports-up-to-10-gb-ephemeral-storage/. Accessed: 2023-09-30.

[37] R. Coker, "Bonnie++ Russell Coker's Documents." https://doc.coker.com.au/projects/bonnie/. Accessed: 2024-08-20.

[38] "Github link." https://github.com/KiritoMiao/ARM-Performance-Predection-model-artifacts.