X86 vs. ARM64: An Investigation of Factors Influencing Serverless Performance

Xinghan Chen
University of Washington
Tacoma, Washington, USA
kirito20@uw.edu

Ling-Hong Hung
University of Washington
Tacoma, Washington, USA
lhhung@uw.edu

Robert Cordingly
University of Washington
Tacoma, Washington, USA
rcording@uw.edu

Wes Lloyd
University of Washington
Tacoma, Washington, USA
wlloyd@uw.edu

ABSTRACT

Function-as-a-Service (FaaS) platforms enable easy deployment and hosting of code known as serverless functions and have gained considerable traction among software developers. Recently, Amazon has offered ARM64 processors as an alternative to x86 for hosting serverless functions on AWS Lambda. To encourage developers to migrate code to ARM64, use of ARM64 processors has been incentivized with a 20% discount. In this paper, we investigate the implications of adopting ARM64 processors for hosting serverless functions. We deploy and benchmark 18 distinct serverless functions that stress a variety of system resources. Additionally, we scaled up the runtime for each of our functions by increasing the work they performed using 40 incremental steps to increase runtime from a few seconds to several minutes. We compared performance differences of x86 vs. ARM64 processors for our serverless function workoads and present our findings summarizing differences in: CPU utilization, function runtime, function runtime variation, and hosting costs. While only 7 of 18 functions ran faster on ARM64, 15 of 18 were less expensive thanks to the cloud provider discount. Performance variation on ARM64 processors was less than half compared to x86 processors potentially from the lack of hyperthreading and lower resource contention for ARM64 CPUs.

KEYWORDS

Function-as-a-Service, Serverless Computing, Performance, ARM

ACM Reference Format:
Xinghan Chen, Ling-Hong Hung, Robert Cordingly, and Wes Lloyd. 2023. X86 vs. ARM64: An Investigation of Factors Influencing Serverless Performance. In *24th ACM/IFIP International Middleware Conference, December 11–15, 2023, Bologna, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware 2023, December 11–15, 2023, Bologna, Italy
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Functions-as-a-Service (FaaS) is a cloud computing delivery model that allows developers to deploy and run their code in a serverless environment. FaaS offers several advantages over traditional container and virtual machine (VM) solutions. One of the main advantages of FaaS is its scalability. With FaaS, the infrastructure automatically scales up or down based on incoming workloads, allowing developers to focus on writing code without worrying about infrastructure management. This enables FaaS to handle sudden traffic spikes without human intervention, where traditional container or VM based cloud infrastructure cannot. Another advantage of FaaS is its cost-effectiveness. FaaS providers only charge for the actual amount of resources a function consumes, not for an entire server or VM. This means developers can save a lot of money by using FaaS, especially for applications with variable workloads. By leveraging the benefits of FaaS, developers can build robust, scalable, and cost-effective applications that can handle sudden traffic spikes and provide a seamless user experience.

While x86 processors have long been dominant, ARM64 processors have recently become more popular for their energy efficiency and suitability for low-power devices. ARM64 processors are designed to lower power consumption while offering high performance. Recently, ARM64 processors have been leveraged by cloud providers in data centers to provide alternatives to x86 for compute platforms. AWS Lambda now offers ARM64-based Graviton2 processors as an alternative to x86 Intel Xeon processors [13] at a 20% discount. While new ARM64 processors offer attractive energy and cost savings, there are potential downsides of concern. One of the major disadvantages of adopting ARM64 processors is the limited availability of software and tools. Since ARM64 processors are less common in the server market compared to x86, there may be fewer software packages and development tools available to organizations. Adoption of ARM64 processors requires the additional effort of x86 code migration and testing on the ARM64 platform. For some use cases, the effort may be trivial, but more effort may be involved for other use cases, particularly those that rely on x86-specific CPU extensions. This can result in longer development times and higher costs, as organizations may need to invest in custom development or software migration to ARM64 platforms [6]. Since ARM64 processors have a different architecture than x86, applications optimized for x86 may not perform as well on the ARM64 platform without major modifications.

In this paper, we investigate the performance differences of x86 vs. ARM64 processors by executing 18 distinct serverless function workloads on AWS Lambda which supports both Intel Xeon and ARM64-based processors. Our goal is to evaluate differences in CPU utilization, function runtime, function runtime variance, and hosting costs. By conducting this study, we aim to offer a detailed evaluation of the two architectures' capabilities within a serverless FaaS environment. The findings can guide developers and organizations considering migration to ARM64-based compute infrastructure to better understand cost vs. performance trade-offs compared to x86-based infrastructure.

1.1 Research Questions

This paper investigates the following research questions:

RQ-1: (CPU Utilization): How do Linux CPU utilization measurements compare for serverless functions run on x86 (Intel) vs. ARM64 (Graviton2) processors? We investigate changes in CPU user mode time, CPU kernel mode time, and CPU idle time.
RQ-2: (Performance): How does serverless function runtime compare on x86 (Intel) vs. ARM64 (Graviton2) processors? Using runtime on x86 processors as a baseline, we identify functions with faster runtime on ARM, similar runtime on ARM, and slower runtime on ARM. In addition, we investigate x86 vs. ARM64 runtime implications when scaling up the work performed by function instances.
RQ-3: (Performance Variance): What is the difference in performance variance of serverless functions executed on x86 (Intel) vs. ARM64 (Graviton2) processors? We calculate and analyze the coefficient of variation of function runtime while scaling the work of function instances using forty distinct steps to increase runtime.
RQ-4: (Cost): What is the cost difference in hosting serverless functions on x86 (Intel) vs. ARM64 (Graviton2) processors? We compare the overall hosting costs of 18 distinct functions while scaling function runtime across forty steps.

1.2 Contributions

This paper provides the following research contributions:

(1) We provide a comparison of executing 18 distinct serverless functions on the ARM64 and x86 architectures. Standard and custom benchmarks were encapsulated into serverless functions to compare CPU utilization, runtime, and cost differences. Functions where ARM64 outperforms x86, x86 outperforms ARM64, and where performance is similar are identified.
(2) We investigate runtime variance on ARM64 vs. x86 processors by leveraging our 18 distinct serverless functions while scaling function runtime across forty steps. The coefficient of variation of function runtime was determined for serverless function workloads. X86 function instances were found to exhibit more runtime variation, which is likely attributed to CPU hyperthreading.

2 RELATED WORK

Recently, ARM processors designed for Internet-of-Things and mobile devices have gained popularity due to their increased power efficiency and low cost. However, migration to the ARM platform can pose considerable challenges for developers. ARM64 processors have emerged as an alternative to x86 processors in data centers and cloud computing. 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. To encourage developers to migrate code to ARM64, these processors are offered at a 20% discount [13].

Xie et al. compared the use of x86 Intel Xeon processors to the ARM-based Phytium 2000+ processor for hosting serverless FaaS platforms by deploying the Kubernetes-based Knative and OpenFaaS serverless frameworks [15]. Both frameworks were deployed using one master node and eight worker nodes with 8 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 for hosting the open-source FaaS frameworks that ARM64 processors had greater startup latency but similar scaling responsiveness while being more susceptible to resource contention under intense pressure from many concurrent function requests. In [7], Javed et al. compared performance of OpenFaaS, Apache OpenWhisk, and AWS Greengrass run on a cluster of four ARM-based Raspberry Pis and compared performance to AWS Lambda and Azure Functions. The authors created 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.

Lambion et al. compared x86 vs. ARM64 performance for a natural language processing pipeline consisting of preprocessing, training, and query functions run on AWS Lambda [10]. The total pipeline runtime averaged 1.7% slower on x86 processors than ARM, but this performance loss was due to resource contention. Running the pipeline continuously over 24 hours on ARM64 and x86, the best-case performance on x86 was 13.53% faster than ARM64, whereas the worst-case performance was 17.10% slower. Resource contention potentially from x86 platform popularity resulted in a 3-fold difference in runtime variation on x86 vs. ARM64. ARM64 was less expensive than x86 as 10,000 NLP pipeline executions cost $245 vs. $311 on x86, a savings of 21.4%. Park et al. profiled performance of a deep neural network inferencing suite on AWS Lambda using ARM64 and x86 processors, extending the work of Lambion [12]. Park tested various model optimization heuristics, finding that ARM64 optimization libraries did not deliver equivalent performance enhancements compared to x86. Park concluded that ARM64 hardware is not yet as efficient due to immaturity of the development eco-system.

3 METHODS

3.1 Systems and Tools

To compare the use of ARM64 vs. x86 processors for the execution of serverless functions, we leveraged the Serverless Application Analytics Framework (SAAF). SAAF is a tool designed to help developers better understand and optimize the performance of their serverless applications [3, 5]. It is a reusable framework that supports multiple programming languages and can be integrated into

a function's package for deployment to multiple commercial or open-source FaaS platforms. SAAF collects forty-eight distinct metrics that enable developers to profile CPU and memory utilization, monitor infrastructure state, and observe platform scalability. By providing improved observability of FaaS function deployments, SAAF enables developers to more accurately predict runtime, performance, and resource utilization, which can help reduce costs and improve application performance. For our measurements with SAAF, we captured userRuntime which provides runtime of the function code without profiling overhead.

We investigated x86 and ARM64 performance on AWS Lambda, a serverless FaaS platform offered by Amazon Web Services (AWS) [14]. AWS Lambda allows developers to run code without provisioning or managing servers by uploading their code in a variety of programming languages, and the service will automatically run code in response to events while automatically managing computing resources. AWS Lambda supports running functions written in a wide range of programming languages, including Python, Java, Go, C#, and Node.js, as well as custom runtimes and functions packaged as container images. For this paper, AWS Lambda was leveraged to support the comparison of x86 vs. ARM64 performance because it is the only public serverless Function-as-a-Service platform currently providing access to ARM64 and x86 processors. [10, 13]

3.2 Serverless Functions

To support our investigation, we leveraged 18 distinct serverless functions as shown in table 1. We utilized functions from the FunctionBench benchmark suite which provides a diverse set of functions tailored for benchmarking FaaS platforms [8]. Functions from FunctionBench include: linpack, json_dumps, chameleon, and float. In addition, we also leveraged the Serverless Benchmark Suite (SeBS), which also provides functions for benchmarking FaaS platforms [2]. Functions from SeBS include: graph-pagerank, graph-mst, graph-bfs, compression, and video-processing.

We created four serverless functions by wrapping existing Linux benchmarks. Three functions were created by wrapping benchmarks from Linux sysbench: primenumber (sysbench-cpu), thread (sysbench-threads), and readmemory (sysbench-memory) [9]. The readdisk function was created by wrapping the fio - flexible I/O tester benchmark [1]. Five serverless functions were created from scratch to stress specific aspects of the system not covered by any other benchmarks. Chacha20 used the openssl encryption libraries to encode an 8MB file n times [11]. For chacha20, we disabled acceleration, Neon on ARM and AVX on x86 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 runtime. Readwritememory performed n iterations of creating a 1 KB byte array, writing 1,024 bytes, and deleting the array. Socket opened and closed a socket n times to scale runtime.

By utilizing these functions, we were able to conduct a comprehensive examination of performance for the x86 and ARM64 architectures in a serverless computing environment. Utilizing existing benchmarks and benchmark suites such as Linux sysbench, FunctionBench, and SeBS provides the benefit of community validation, as these benchmarks are designed to be as comprehensive and unbiased as possible and have already been extensively tested.

	Short Name	Function Name	Description
cpuUser	linpack	python_linpack	Solve linear equations: Ax = b
	chacha20	openssl_encrypt_chacha20	Repeatedly perform openssl encryption of 8MB file n times
	sqlite	python_sqlite_dump	Execute n random SELECT queries on a 10*1000 SQLite database
	video-processing	ffmpeg_sebs_220_gif	Convert PNG to GIF n times
	json_dumps	python_json_dumps	JSON deserialization using a downloaded JSON-encoded string dataset
	graph-pagerank	python_sebs_501_pagerank	PageRank implementation with igraph.
	graph-mst	python_sebs_502_mst	Minimum spanning tree (MST) implementation with igraph.
	float	python_float_operation	Perform sin, cos, sqrt ops
	chameleon	python_chameleon	Create HTML table of n rows and M columns
	graph-bfs	python_sebs_503_bfs	Breadth-first search (BFS) implementation with igraph.
	primenumber	sysbench_cpu_prime	Prime number generator
cpuKernel	thread	sysbench_thread	Create thread, put locks and release thread
	filehandle	python_fopen	Open and close file handles
	socket	python_socket	Open and close socket n times
Memory	readmemory	sysbench_memory	N sequential reads of 1GB memory block
	readwritememory	python_malloc_write	Allowcate 1MByte of memory, write 0x42 into it and release
I/O	readdisk	fio_disk_io_random_read	Test random read speed on a 1GB block
	compression	python_sebs_311_compression	Create a .gz file for a file

Table 1: Function descriptions and short names grouped by the predominantly stressed resource. Group 1: CPU user time (blue), Group 2: CPU kernel time (yellow), Group 3: Memory Intensive (orange) function, and Group 4: I/O (grey)

For this research, we assembled a diverse set of serverless functions encompassing a variety of tasks ranging from compute-intensive to I/O-bound workloads, enabling us to evaluate a broad range of aspects of serverless system performance for both architectures.

3.3 Experiment Environment and Methodology

All tests were conducted on AWS Lambda using the us-west-2 (Oregon) region. We provisioned our AWS Lambda functions with 3 GB memory to ensure full access to two virtual CPUs (2 vCPUs) [4]. This is the smallest memory size that allows full access to two vCPUs. Any smaller memory size would result in a fractional share of CPU time equivalent to having less than two vCPUs. This could hinder performance and increase performance variance, skewing the results of our comparisons. It was imperative to maintain a level playing field to ensure our performance measurements were indicative of the actual capabilities of the respective architectures to remove influences from unequal resource allocations. Functions were tested using identical configurations on x86 and ARM64 to ensure consistency for accurate benchmarking and analysis.

For tests with I/O operations, a temporary, ephemeral disk of 5 GB was allocated [16]. This ensured that any variability in performance was not due to disk space limitations but could be attributed to the architectural differences and workload characteristics.

To gain a comprehensive understanding and to mitigate potential performance variance, we scaled up the runtime of each function using 40 distinct steps and then executed each function ten times per step. The steps were equidistantly spaced to enable profiling the functions over a broad range of runtime spanning from a few seconds to several minutes. This allowed a well-rounded view of how each architecture responded to different workloads and conditions.

4 RESULTS

4.1 ARM64 vs. x86 CPU Utilization Comparison

To investigate **(RQ-1)**, we profiled Linux CPU utilization of our 18 serverless functions on ARM64 and x86 as shown in figure 1. Initially, we captured six distinct CPU metrics using SAAF: cpuUser, cpuKernel, cpuIrq, cpuSoftIrq, cpuIOWait, and cpuIdle [5]. On close examination, values for cpuIrq, cpuSoftIrq, and cpuIOWait were found to be nearly zero. To improve the clarity of figure 1, we omit these metrics and depict the remaining CPU metrics to show the time the CPU executed user instructions (cpuUser), time the CPU executed kernel instructions (cpuKernel), and time the CPU was idle (cpuIdle). As our AWS Lambda functions had access to two vCPUs, one thread running at 100% CPU utilization is shown as 50% cpuUser time in the graph. We found that for most of our functions,

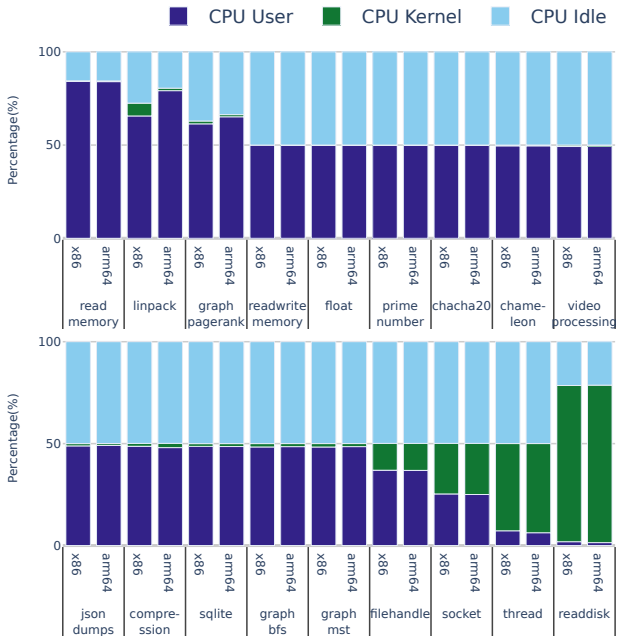

Figure 1: CPU mode time percentage of each function

both architectures exhibited similar CPU utilization. However, two functions (linpack and graph-pagerank) had noticeably higher CPU utilization on ARM64. Both of these functions had higher CPU kernel time on x86 than ARM64 and greater runtime on ARM64 than x86, suggesting that x86-specific kernel functions were used to help optimize performance. Another observation from figure 1 was that most of our functions only used a single thread. Note the large number of functions having approximately 50% CPU idle time on the graph. This observation identifies potential for performance optimization if such workloads can be refactored to leverage both vCPUs to reduce runtime and cost [4].

4.2 ARM64 vs. x86 Performance Comparison

To support investigation of **(RQ-2)**, we executed 18 distinct functions on x86 and ARM64 to compare performance. Runtime performance differences on ARM64 relative to x86 are shown in Figure 2. Figure 3 provides box plots showing x86 and ARM64 function runtime for our functions. We categorize performance of our functions into three groups: ARM-slower (than x86), ARM-similar (to x86), and ARM-faster (than x86). For **ARM-slower**, ARM64 function runtime was >10% than x86. Functions with ARM-slower performance include linpack, chacha20, thread, filehandle, sqlite, and readwritememory. For **ARM-similar**, runtime was within ±10% of x86. Functions with ARM-similar performance include video-processing, json_dumps, socket, graph-pagerank, graph-mst, compression, float, and graph-bfs. For **ARM-faster**, runtime was >10% faster than x86. Functions with ARM-faster performance include chameleon, readdisk, readmemory, and primenumber.

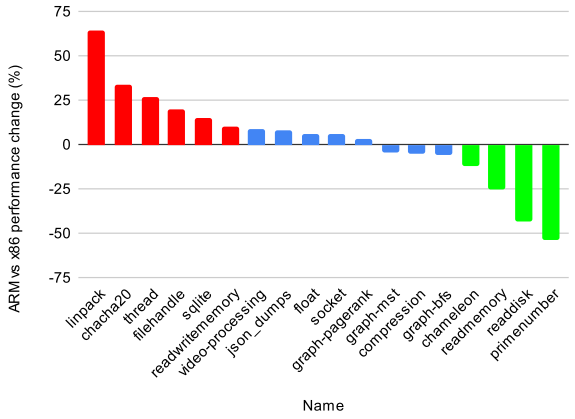

Figure 2: ARM64 Function Performance Difference vs. x86

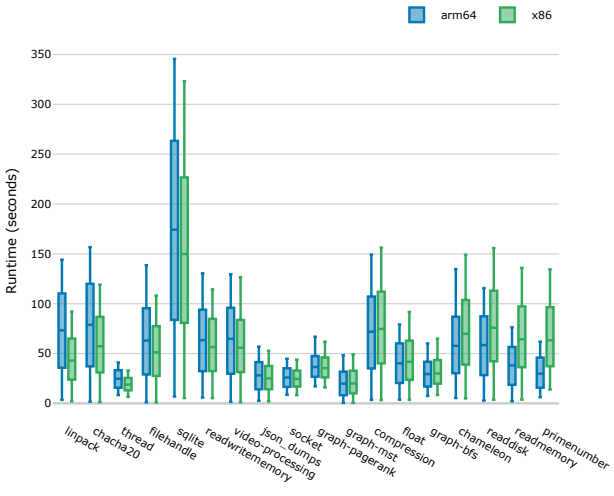

Figure 3: Min and max time for X86 vs. ARM

Our observations revealed notable differences in workload performance between ARM64 and x86 architectures when deployed on AWS Lambda. Figure 2 shows how min-max runtime performance deltas can swing by as much as ±50% depending on the specific workload. An interesting observation was that ARM-slower workloads averaged 9.96% CPU Kernel time, ARM-similar workloads averaged 3.94% CPU Kernel time, and ARM-faster workloads averaged just 0.28% CPU Kernel time except for readdisk, which had high CPU Kernel time (~77%) on both ARM64 and x86. Workloads that required more kernel mode operations appeared to execute slower on ARM, and when kernel mode was avoided, they executed faster.

When scaling runtime of individual workloads, the performance differences between ARM64 and x86 remained relatively constant. For instance, if a particular workload was 20% faster on ARM, this advantage remained whether the function ran for a short or long period. This is likely because our workloads did not perform distinctly different operations when runtime was scaled. Instead, they did more or less of the same operations. ARM64 vs. x86 runtime differences may not hold for non-deterministic functions whose CPU utilization is subject to change based on variable outcomes.

Various factors may have contributed to runtime differences. For example, the differences in micro-architectural designs between ARM64 and x86 could lead to variations in how efficiently certain operations were executed. Additionally, compiler optimizations and hardware acceleration specific to each architecture likely impacted how effectively code was run. For example, video-processing and linpack is using SSE/AVX on X86 and Neon on ARM64. Furthermore, AWS Lambda's resource allocation and management strategies may interact differently with each architecture, contributing to runtime differences.

4.3 ARM64 vs. x86 Performance Variation

To investigate performance variability on x86 vs. ARM64 for **(RQ-3)**, we calculated the average Coefficient of Variation (CV) of our functions as shown in figure 4. We found that ARM64 processors provided less runtime variance than their x86 counterparts on AWS Lambda. Specifically, 12 out of 18 functions tested exhibited lower CV on ARM. The average CV for x86 was more than 2x greater than ARM64, with an average runtime CV of 1.802% for ARM64 and 3.876% for x86. Previously, it was suggested that higher CV for serverless functions on x86 vs. ARM64 may be the result of platform contention for the CPU (where x86 is more popular), and x86 hyperthreading and the subsequent lack of hyperthreading on the ARM-based Graviton2 processor [10]. Our results, where we have observed lower runtime variance for 12 out of 18 functions, add support to this claim. Further, we identify a potential cause to explain higher CV on ARM64 is I/O operations (IOPS). The filehandle and chacha20 workloads had high IOPS and subsequently higher CV on ARM64 than x86.

Figure 5 compares changes in CV while increasing function runtime successively using 40 steps. Functions are shown in separate graphs for our performance groups: ARM-faster, ARM-similar, and ARM-slower. Graphs on the left show CV for ARM, while graphs on the right show CV for x86. The x86 processor exhibited notably higher CV in the ARM-similar performance group. Overall, figure 5 shows the higher CV observed when executing functions on x86.

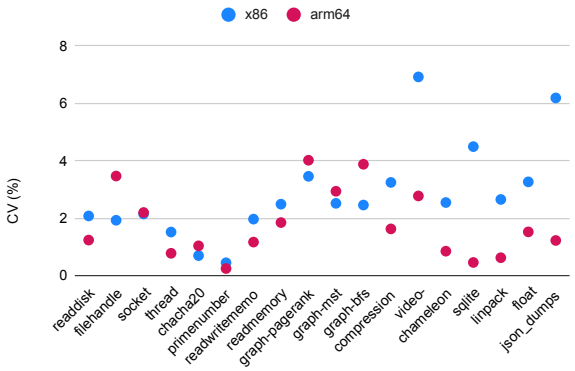

Figure 4: Average CV (%) of function runtime

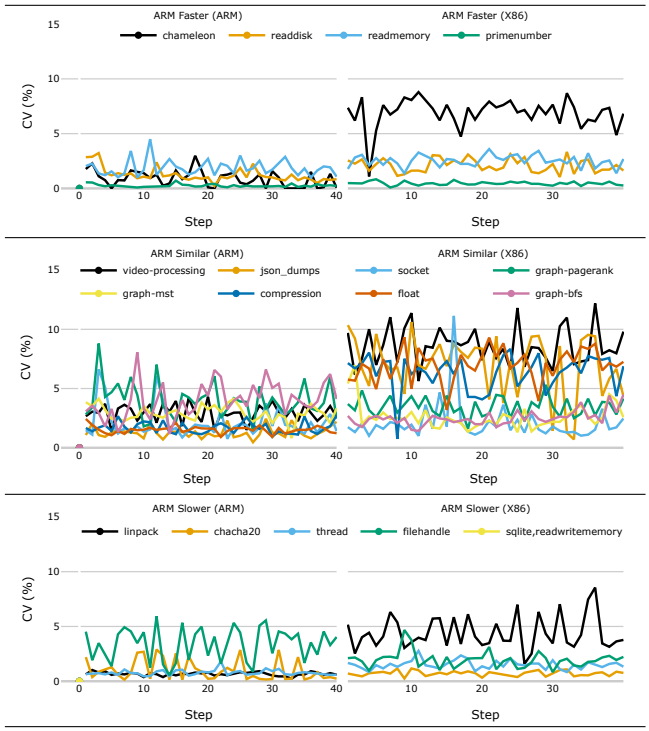

Figure 5: Function runtime: change in CV(%) over 40 steps

4.4 ARM64 vs. x86 Cost Comparison

To investigate **(RQ-4)**, we compared hosting costs of our 18 functions on x86 vs. ARM. Figure 6 depicts cost estimates for 10,000 sets of function calls where run time was scaled up using 40 steps for a total of 400,000 calls per function. Figure 7 shows the % cost difference when comparing function hosting costs on ARM64 vs. x86 processors. Only three functions Linpack, Chacha20, and Thread had higher costs on ARM64 on AWS Lambda. The remaining 15 functions had lower costs on ARM64 than x86. Cost savings was helped by the 20% cost discount Amazon offers when using ARM64 processors on the platform. Without the discount, only 8 of 18 functions would experience a cost savings. The cost savings helps to offset the higher runtime of workloads that execute slower on ARM64 as shown in figure 2.

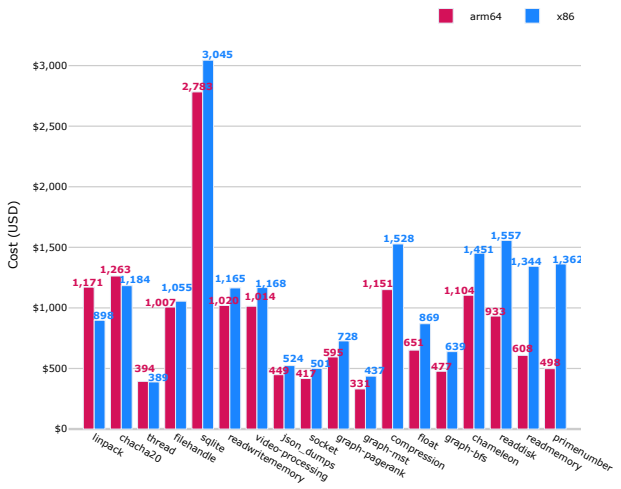

Figure 6: Estimated cost of 400k function calls- x86 vs. ARM

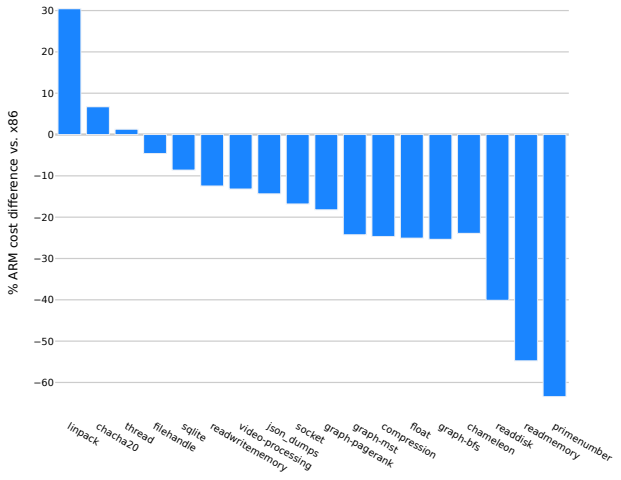

Figure 7: ARM cost difference (+/- %) relative to x86

5 CONCLUSION

In this paper, we compared the performance characteristics of ARM64 and x86 architectures in the context of serverless computing, specifically using AWS Lambda. By leveraging FunctionBench and SeBS as established benchmark suites and custom functions, we profiled performance of CPU, memory, and I/O bound workloads on the two architectures. We summarize our findings of CPU architectural implications as follows:

(RQ-1): While most functions had similar CPU utilization profiles across both architectures, some functions on ARM64 had higher CPU kernel mode utilization. These differences may help detect where x86 vs. ARM64 performance differences are likely occur.

(RQ-2): ARM64 can provide performance advantages for serverless workloads. Of the 18 serverless functions tested, ARM64 provided faster runtime than x86 for 7 out of 18 functions. Four functions were more than 10% faster on ARM64. Runtime improvements appeared highly dependent on the nature of the workload.

(RQ-3): Functions run on x86 on AWS Lambda, exhibit more than twice the runtime variance vs. ARM64 making x86 less reliable for consistent performance.

(RQ-4): ARM64 offers cost savings on AWS Lambda (15 of 18 tested serverless functions). Some of the cost savings are attributed to the 20% cost discount offered by the cloud provider for ARM64 processors.

Given the wide variety of available ARM64 processors, future research can help generalize our results more broadly for ARM-based serverless environments. Future work can also investigate the creation of performance models to predict performance from code migration to ARM64 environments. Performance models have potential to aid developers and practitioners in planning code migrations to prioritize migrations that will offer the best performance and cost trade-offs on ARM64. By shedding light on these performance aspects, our research contributes to the ongoing discourse on optimizing serverless architectures, offering valuable guidance for organizations aiming to make the most out of their serverless computing investments.

REFERENCES

[1] Jens Axboe. [n. d.]. 1. fio - Flexible I/O tester rev. 3.35 8212; fio 3.35-6-g1b4b-dirty documentation. https://fio.readthedocs.io/en/latest/fio_doc.html. Accessed: 2023-09-30.

[2] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*. 64–78.

[3] Robert Cordingly, Navid Heydari, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, and Wes Lloyd. 2021. Enhancing Observability of Serverless Computing with the Serverless Application Analytics Framework. In *Comp. of the 2021 ACM/SPEC Int. Conf. on Performance Engineering, Tutorial*.

[4] Robert Cordingly, Sonia Xu, and Wes Lloyd. 2022. Function Memory Optimization for Heterogeneous Serverless Platforms with CPU Time Accounting. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 104–115.

[5] Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, and Wes Lloyd. 2020. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software. In *Proc. of the 2020 Sixth Int. Wksp on Serverless Computing*. 67–72.

[6] Blake W Ford, Apan Qasem, Jelena Tešić, and Ziliang Zong. 2021. Migrating software from´ x86 to arm architecture: An instruction prediction approach. In *2021 IEEE Int Conf on Networking, Architecture and Storage (NAS)*. IEEE, 1–6.

[7] Hamza Javed, Adel N. Toosi, and Mohammad S. Aslanpour. 2021. Serverless Platforms on the Edge: A Performance Analysis. arXiv:2111.06563 [cs.DC]

[8] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.

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

[10] Danielle Lambion, Robert Schmitz, Robert Cordingly, Navid Heydari, and Wes Lloyd. 2022. Characterizing X86 and ARM Serverless Performance Variation: A Natural Language Processing Case Study. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. 69–75.

[11] openssl.org. [n. d.]. Enc - OpenSSLWiki. https://wiki.openssl.org/index.php/Enc. Accessed: 2023-09-30.

[12] Subin Park, Jaeghang Choi, and Kyungyong Lee. 2022. All-you-can-inference: serverless DNN model inference suite. In *Proceedings of the Eighth International Workshop on Serverless Computing*. 1–6.

[13] Danilo Poccia. [n. d.]. AWS Lambda Functions Powered by AWS Graviton2 Processor – Run Your Functions on Arm and Get Up to 34Performance. https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance/. Accessed: 2023-09-30.

[14] Amazon Web Services. [n. d.]. AWS Lambda. https://aws.amazon.com/pm/lambda/. Accessed: 2023-09-30.

[15] Dong Xie, Yang Hu, and Li Qin. 2021. An evaluation of serverless computing on x86 and arm platforms: Performance and design implications. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 313–321.

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