INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2025, VOL. 71, NO. 3, PP. 1-9
Manuscript received May 08, 2025; revised July 2025. doi: 10.24425/ijet.2025.153619

Evaluation of configurations of AWS Lambda functions

Ilona Bluemke, and Arkadiusz Zdanowski

***Abstract*—AWS Lambda is a widely used serverless computing service that executes code in response to events and automatically manages the underlying hardware resources. Lambda integrates with many AWS services and offers two processor architecture options for running functions: x86_64 (CISC) and arm64 (RISC). Determining the optimal settings for the lowest cost or execution time is not straightforward due to performance variations between processor architectures, the wide range of configuration options, and the workload-dependent nature of function execution efficiency. We developed a tool which we used in experiments examining different configurations and processors architectures for several algorithms. In this paper two of such experiments are presented in detail.**

***Keywords*—serverless computing; Function-as-a-Service; Amazon Web Services; AWS Lambda**

I. Introduction

IN computer science, "the cloud" refers to the on-demand delivery of computing resources over the internet. These resources include servers, storage, databases, networking, and software, which users can access without the need to own or maintain physical infrastructure. The cloud offers scalability, flexibility, and cost efficiency, allowing users to adjust resources as needed and pay only for what they use [1]. Serverless computing [2] eliminates infrastructure management and allows developers to concentrate on writing and executing code. Among serverless platforms, AWS Lambda [3] is one of the most widely used. Optimizing AWS Lambda functions for cost and performance remains challenging. Users can configure memory size and processor architecture, but the impact of these parameters on execution efficiency is not well-documented. Existing benchmarking tools (listed in Section III) provide limited configuration choices or lack comprehensive statistical analysis, making it difficult to compare different configurations systematically.

At the Institute of Computer Science, Warsaw University of Technology a tool, called LambdaLab [4], that examines AWS Lambda performance with a range of configurations, including x86_64 and arm64 processor architectures was designed and implemented. This tool was used to conduct a series of experiments analyzing CPU-intensive workloads on AWS Lambda. In [5] the superiority of arm64, while executing cryptographic hashing SHA-256 algorithm, was described. As the configurations for this experiment were not fully shown we wanted to use the same algorithm and examine a variety of settings. The results of these experiments are presented in Section IV. Our experiment provides valuable insights into performance trade-offs, highlighting the impact of memory size and processor architecture on execution cost.

Section II contains brief theoretical background on cloud services and serverless architectures. AWS Lambda is also presented in more details. Related work i.e. benchmarking tools and experiments are presented in Section III.

In the last section a summary of key findings and potential directions for future research is given. By addressing drawbacks of existing serverless benchmarking tools and experiments this work contributes to a better understanding of AWS Lambda cost and performance optimization and provides a practical solution for developers and researchers.

II. Theoretical Background

Below, basics information on abstract cloud services and AWS Lambda are given.

A. Abstraction layers in cloud services

Cloud computing services are served at different levels of abstraction, offering various models depending on the degree of control and management required by developers and users. Depending on the level, the cloud vendor provides only infrastructure, the platform or the platform along with the necessary infrastructure, or access to software running on a platform, supported by infrastructure. In Fig 1. the three primary cloud service models are shown and the providers and consumers for every model are presented.

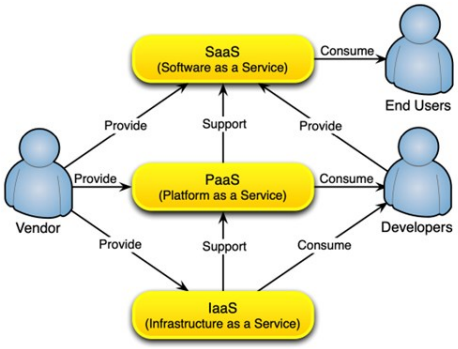

Fig. 1. Cloud services abstraction layers -source: [6].

In this view the cloud vendor provides infrastructure, platform and software, while developers consume infrastructure and platform to provide software, and the end users only consume software.

Authors are with Institute of Computer Science, Warsaw University of Technology, Poland (e-mail: Ilona.Bluemke@pw.edu.pl, arkadiusz.zdanowski.stud@pw.edu.pl).

Infrastructure as a Service (**IaaS**) is the lowest level of cloud service, which offers access to infrastructure hosted by the cloud vendor, including servers, storage, networking [6]. Developers can configure the infrastructure remotely, according to their needs. IaaS model provides flexibility and scalability, enabling to scale resources up or down, as the demand fluctuates. It also benefits from the pay-as-you-go pricing model, which ensures that the users pay only for the resources they consume, reducing upfront capital expenditure and maintaining clear operational costs. IaaS is designed for users who need the ability to install, configure and operate arbitrary low-level software or applications. It is most commonly used for applications running for an extended period of time (or available on 24/7 basis).

Platform as a Service (**PaaS**) provides a higher level of abstraction by offering a managed environment for developing, testing, and deploying applications. It includes runtime environments, development tools, and middleware, all hosted and maintained by the cloud provider. Unlike IaaS, PaaS does not grant users control over the underlying infrastructure, such as server configuration or network management. Instead, applications are developed within a pre-configured platform that handles infrastructure-related tasks. PaaS allows developers to deploy applications without directly managing servers or scaling resources manually, therefore it is particularly useful for scenarios where fast-paced development and deployment is required.

Software as a Service (**SaaS**) is the highest level of abstraction in cloud computing, providing users with access to fully managed software applications over the internet. Unlike IaaS and PaaS, SaaS does not allow developers to control the underlying platform or infrastructure. Instead, users interact directly with pre-built applications provided and maintained by the cloud vendor. SaaS applications are accessed via web browsers, eliminating the need for installation, maintenance, or resource management by the user. All aspects of infrastructure, security, and software updates are managed by the cloud provider. SaaS model is particularly useful for non-technical end-users who require ready-to-use applications without involvement in software development or system administration. The **serverless** computing model enables developers to focus on writing and executing their code without the need to manage anything else. Although "serverless" might suggest no servers are involved, the underlying infrastructure is still managed by the cloud provider, and computational resources are provisioned dynamically. The significant benefit lies in the abstraction of infrastructure management. When compared to the classic cloud service models major differences in serverless approach [7] can be seen:

- Computing provisions resources only for the duration needed to execute a single function, unlike IaaS, where infrastructure is continuously provisioned and requires user management.
- Computing abstracts away the platform management, providing a fully managed environment, unlike PaaS, where developers still need to manage the platform and its environment.
- Computing gives developers the ability to define and deploy individual functions for event-driven architectures, unlike SaaS, where the cloud provider manages the entire software stack.

According to Berkeley View [8], the serverless model combines two key components:

1. Function as a Service (**FaaS**): Provides execution environments for isolated functions, which are invoked in response to events.
2. Backend as a Service (**BaaS**): Complements FaaS by offering services like databases, authentication, storage and logging, which are essential for backend operations but abstracted away from the user.

FaaS functions are designed to operate independently and are triggered by external events, such as HTTP requests, database updates, or scheduled tasks. Once the event arrives, it is passed to a controller for validation. If the event is valid, the controller initiates a function container. A container in serverless computing is a lightweight, isolated environment where the function code is executed. It is designed specifically to start quickly, ensuring minimal latency when a function is invoked. While the container provides isolation and efficiency, it runs on a server managed by the cloud provider, abstracting the infrastructure. The container includes the runtime environment (the chosen programming language engine, libraries and frameworks) and the function's source code.

B. AWS Lambda

AWS Lambda is a widely used serverless compute service that executes code in response to events and automatically manages the underlying resources. As an implementation of the Function as a Service (**FaaS**) model, Lambda allows developers to focus entirely on writing, deploying and executing functions. Lambda integrates seamlessly with many AWS services, which provide Backend as a Service (BaaS) capabilities.

To understand AWS Lambda function, a distinction must be made between a function in the context of FaaS and a function in traditional programming. In traditional programming, a function is a reusable block of code that performs a specific task and can be invoked multiple times within a program. In FaaS, a function refers to a self-contained, stateless unit of execution, which may encompass several programming-level functions [9].

A Lambda function is a cloud-managed object that consists of the executable code and associated configurations. Each function is deployed within a specific AWS Region, meaning it runs only within the selected geographical data center. New AWS Lambda function requires specifying a function name, selecting a runtime environment, and choosing a processor architecture. The runtime environment defines the programming language in which the function will run. AWS Lambda offers two processor architecture options for running functions: x86_64 and arm64:

1. The x86 architecture, originally introduced by Intel in 1978 with the famous 8086 microprocessor [10]. Based on Complex Instruction Set Computing (CISC), x86 processors offer a wide range of instructions, providing versatility and ease of use.
2. In contrast, Graviton2 processors are custom-built by AWS based on the 64-bit arm architecture [11], which uses RISC (Reduced Instruction Set Computing) to simplify the instruction set for better energy efficiency. Graviton2 processors are specifically designed for cloud workloads [12], therefore, according to the AWS Documentation [13], they should deliver higher performance and improved cost-efficiency compared to the x86.

AWS Lambda operates on a pay-as-you-go pricing model, where users are charged based on the number of requests and the duration of function executions. The pricing is provided for a thousand or million invocations. The total cost of an invocation depends on several factors [14]:

1) Request cost: Lambda charges $0.20 per 1 million requests to invoke the function.
2) Computation cost: Lambda charges for a gigabyte-second of computation. The value is calculated by simply converting the following configuration factors to seconds and gigabytes, respectively, and multiplying them.
   - Execution Time: Lambda charges for each millisecond of the time it takes for the function to complete.
   - Memory Allocation: Lambda charges for the amount of memory allocated to the function, in MB.
3) Ephemeral Storage: AWS Lambda provides 512 MB of default short-term storage for free. If more volume is required, users can allocate up to 10 GB of temporary storage at a rate of $0.000000025 per GB-second. The storage is called ephemeral, because it is automatically deleted after function execution, and cannot be reused between multiple invocations.

Finally, the compute cost is multiplied by the ratio which depends on the chosen processor architecture. The ratio for x86_64 is around 20% more expensive than that for arm64. The exact values are also dependent on the region and subject to change over time.

Optimizing AWS Lambda functions for cost and performance is challenging. Users can configure memory size and processor architecture, but the impact of these parameters on execution efficiency is not easily seen. A benchmarking tool is necessary to compare different configurations systematically and choose the optimal option.

III. Related work

Optimizing serverless workloads for performance and cost is a very difficult task. The optimal settings for the lowest cost or execution time depends on many factors -configuration options so systematic benchmarking is necessary to achieve the best performance-cost balance. By analyzing execution times, resource utilization, pricing models in different configurations, benchmarking helps developers to optimize their serverless applications. The performance evaluation of FaaS platforms has been the research goal of many works. Scheuner and Leitner in their literature review [9] containing 112 studies, found that AWS Lambda is the most frequently studied platform, followed by Azure Functions, Google Cloud Functions, and IBM Cloud Functions. The majority of research concentrated on micro-benchmarks measuring platform overhead and CPU speed. This review identified also that academic and industrial benchmarking approaches were using different configurations.

The majority of the studies focus on simple function executions. Research on complex workloads and function triggers, such as message queues, streams, and workflow integrations are rather limited.

Many benchmarking tools have been developed to optimize memory allocation and execution time for serverless computing engines e.g.: AWS Lambda Benchmark Tool [15], AWS Lambda Benchmark [16], ServerlessBench [17], FaaSdom [18], ServerlessBenchmark [19], AWS Lambda Power Tuning [20]. Each of these tools has some strengths and limitations but in most of them the statistical analysis is very limited (or nonexistent) and comparison of different processors architectures is not possible. At the Institute of Computer Science, Warsaw University of Technology a tool, called LambdaLab [4], was designed and implemented to specifically target those limitations. LambdaLab enables evaluations of configurations including all available memory sizes, both processor architectures, provides statistical analysis, including confidence intervals to improve result reliability. It also is able to visualize and show rankings of results, assisting users in selecting the optimal configuration. By incorporating these features, LambdaLab is more flexible than the other tools.

A. Benchmarking experiments

AWS Lambda was evaluated in several benchmarking experiments. In these experiments, the above-mentioned tools were often used. Scheuner and Leitner in their review [9] noted some common features of these experiments:

- Lack of source code and configurations.
- Reliance on mean values without confidence intervals or variance analysis.
- Potential deficiencies of experiments without independent verification.

Some researchers conducted experiments focusing on the performance and usability of serverless computing platforms, AWS Lambda including e.g. Martins, Araujo, and Da Cunha [21] and -Sadaqat, Sánchez-Gordón, and Colomo-Palacios [22]. They evaluated many parameters like execution time, start latency. memory. These experiments lacked detailed statistical analyses to measure performance variability and did not include the experimental configurations, which disables their reproducibility.

In AWS Lambda Benchmarking Study [23], Xebia was comparing the performance of Rust, Scala, Python, and TypeScript in AWS Lambda. The results showed that in terms of execution time and cost efficiency, Rust was first, Python second- and Scala was the worst. This study also did not provide detailed statistical analyses, such as confidence intervals or hypothesis testing.

Wen, Chen, Sarro, and Liu introduced SuperFlow [24], a performance testing framework for serverless computing. They evaluated AWS Lambda's performance under various workloads, they were examining execution time, resource utilization, and scalability. While the framework aimed to provide a systematic approach to performance testing, this study did not discuss the statistical methods used to analyze the results, leaving questions about the robustness of the findings.

AWS Lambda offers two processor architecture for running functions: x86_64 and arm64 (see section II), so some researchers were evaluating the performance and cost efficiency of these two options. AWS, in collaboration with its partner, Cascadeo, conducted experiments evaluating the performance and cost efficiency of AWS Lambda functions running on Arm-based Graviton2 processors versus traditional x86 processors. The results are presented in [5] In Fig. 2 the relative differences between arm64 and x86_64 architectures are shown, expressed in percentages, where positive values indicate higher values for arm64. The study evaluated five benchmarking scenarios, including:

• Processor-intensive: in single-threaded (ST) and multi-threaded (MT) workloads
• Memory-intensive: workloads in ST and MT setups
• Lightweight micro-benchmark

For each of the workloads, three metrics were measured using unspecified methodology:

1. Performance (higher is better, the clock symbol in Fig.2).
2. Cost (lower is better, the encircled $ symbol in the Fig. 2)
3. Work done per dollar (higher is better, the gearwheel symbol in the Fig. 2)

The results suggest that arm64 provides a significant performance boost—over 60% higher for CPU-heavy tasks—while memory-heavy and lightweight workloads show a slight performance reduction of around 6-7%. This led to a claimed cost reduction of over 50% for CPU-heavy workloads and 13-19% for other workloads, all favoring arm64.

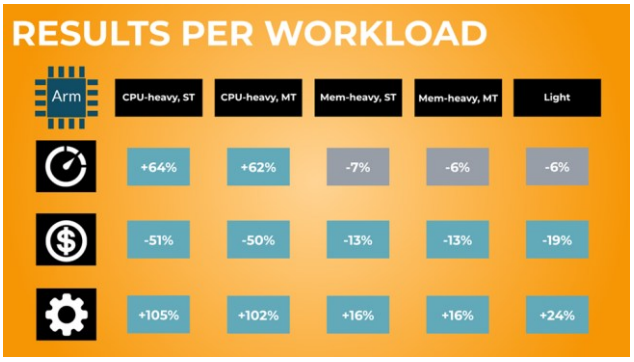

Fig. 2. Results per workload in AWS Lambda arm vs x86 performance and cost -source[5].

Chen conducted an independent experiment evaluating the performance of ARM64 architecture compared to x86 in serverless functions [25]. The research indicated that ARM64 could offer better cost efficiency for certain workloads because out of 18 functions tested, x86 was faster for only 7 functions. In this study mean values were reported and no rigorous statistical analyses were conducted to determine the significance of the performance differences.

In [26] an experiment with single-threaded CPU-intensive iterative workload (Fibonacci computation) was described. The results clearly demonstrated that x86_64 outperforms arm64 in both execution time and cost. This distinction is crucial because it shows that performance advantages are not inherent to an architecture but are highly dependent on workload characteristics.

IV. Experiment

In section II.A some results of experiment conducted by AWS and Cascadeo [5], showing superiority of arm64, were described. As a CPU-heavy workload in this experiment cryptographic hashing SHA-256 algorithm was used. In our experiment presented in [26], where we used Fibonacci numbers as the workload the x86 appeared faster and cheaper. Therefore, we decided to use the SHA-256 algorithm as the workload and to examine a variety of settings which were not covered in [5].

While the SHA-256 algorithm's implementation is provided by hashlib [27], our contribution lies in constructing the workload structure to systematically stress-test CPU performance. It is achieved by executing a configurable number of iterative hash computations on an input of variable size. Each iteration takes the previous hash as input, creating a chained hashing process where every computation depends on the result of the previous one. This ensures a controlled and repeatable execution pattern, making it an effective benchmark for evaluating processor efficiency. Similarly to [5] we examined single threaded (section A) and multi-threaded (section B) workload. We used our benchmarking tool LambdaLab (section III, [26],[4]) for run the workload and analyze the results.

A. Experiment 1- single threaded

The experiment consisted of executing the workload code to compute the SHA-256 hash on a fixed-length input of 1 KB, repeated for 5,000,000 iterations per invocation. This ensures that the workload remains CPU-bound. To obtain statistically meaningful results, the execution was repeated 10 times for each combination of memory size and processor architecture. The runtime environment used was Python 3.13. The experiment tested 23 different memory sizes, ranging from 128 MB to 3008 MB and was performed on both arm64 and x86_64 processor architectures. Therefore, in total, there were 56 unique configurations. Each workload invocation was repeated 10 times, resulting in 560 individual executions.

In Fig. 3 the relationship between execution time (in milliseconds) and memory size (in MB) for both processor architectures, arm64 and x86_64 is presented. Each data point represents the results of a single invocation, with green markers denoting the arm64 architecture, while blue markers represent x86_64. The results in Fig. 3 show that for smaller memory sizes, particularly in the range of 128 MB to 576 MB, arm64 has a slight performance advantage over x86_64, with lower execution times. For example, at 128 MB, the execution time for arm64 is approximately 10% lower than that of x86_64. For memory sizes beyond 576 MB, the performance of both architectures becomes nearly identical, with only marginal differences observed in execution times. This similarity indicates that for higher memory configurations, both architectures are able to fully utilize the allocated resources and achieve near-optimal performance.

In Fig. 4 the execution cost associated with the execution time are presented. Both time and cost dimensions are aggregated from the repeated runs for every memory configuration. The data is presented by points connected with solid lines, representing the average values, with shaded regions around each line indicating the 95% confidence intervals. Architectures are color-coded: x86_64 is represented in blue, while arm64 is depicted in green. Both time and cost dimensions are shown together, as a dual-axis representation, which allows for a direct comparison of execution time (on the left y-axis, milliseconds) and cost (on the right y-axis), across different configurations.

Furthermore, in Fig.4 it can be seen that the execution times for both architectures are nearly identical for memory sizes above 1280 MB. For smaller memory sizes below 1280 MB, x86_64 exhibits slightly higher execution times on average compared to arm64, with the performance gap diminishing as the memory size increases. This trend highlights the slight efficiency advantage of arm64 for smaller memory configurations, likely due to its optimized instruction set for this type of workload.

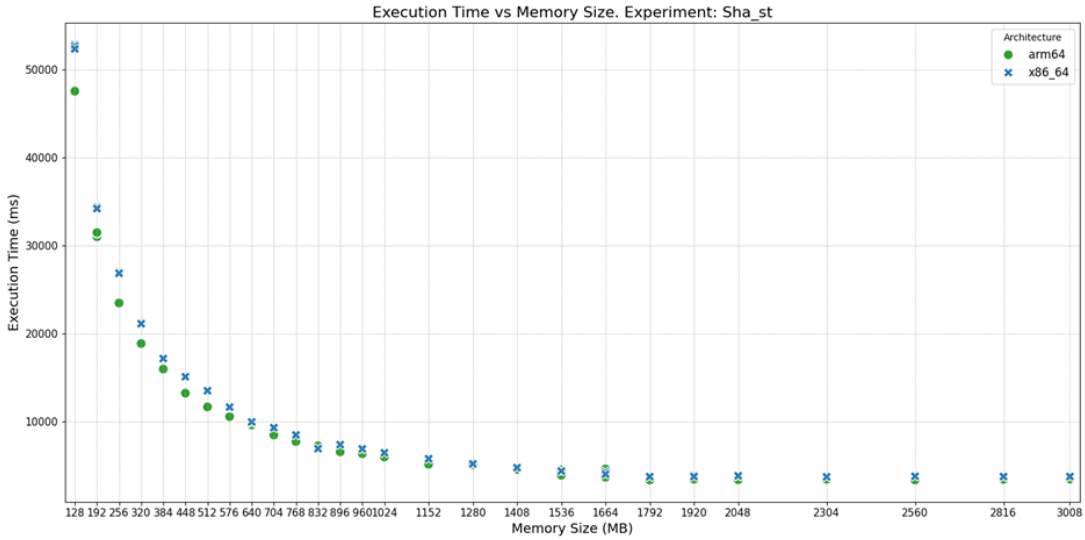

Fig. 3. Execution time vs memory size - single threaded.

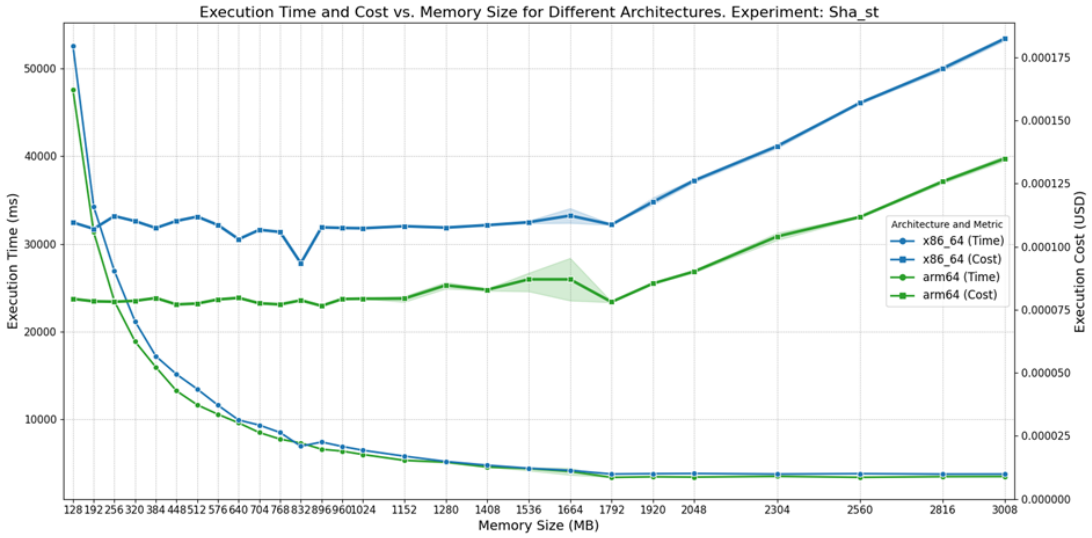

Fig. 4. Execution time and cost vs memory size - single threaded.

Interestingly, a performance anomaly is observed for x86_64 at 832 MB, where a sudden increase in performance occurs. This unexpected behavior may indicate runtime or hardware-level optimizations that are uniquely triggered at this memory size. Additionally, larger confidence intervals are observed for arm64 at 1536 MB and 1664 MB, and for x86_64 at 1664 MB, suggesting increased variability in execution times for these configurations. Outside of these memory sizes, the benchmark demonstrates exceptional consistency, which underscores its reliability and thoughtful design.

In terms of execution cost, arm64 consistently achieves lower costs than x86_64. Because the performance of both architectures is similar, this cost difference is attributed to the lower price per GB-second for arm64. This cost efficiency makes arm64 a more economical choice for this workload. A more detailed cost comparison is presented in Fig. 5.

Fig. 5 confirms that arm64 is consistently less expensive than x86_64, with the relative cost difference ranging from 18.7% to 44.4%, with most values around 40%. The largest cost disparity occurs at the memory size of 512 MB, highlighting that at smaller memory sizes arm64 achieves significant cost efficiency over x86_64. Conversely, the smallest cost difference is observed at 896 MB, indicating that the two architectures have relatively similar costs in this configuration. For smaller memory sizes (128 MB to 576 MB), the percentage cost difference is higher, as reflected by the orange line's peak at 512 MB. The cost difference stabilizes at higher memory sizes, maintaining a consistent gap of around 40%. Interestingly, while the execution time difference is larger at smaller memory ranges, the relative cost difference remains similar across both small and large memory sizes. This suggests that the pricing

model, rather than raw performance differences, plays the dominant role in cost disparity between the two architectures.

This variability underscores the importance of carefully selecting configurations to maximize cost efficiency. Even small changes in memory size can have a significant and sometimes unexpected impact on the relative cost difference. Understanding these trade-offs is crucial for optimizing workloads at scale, particularly when deploying cost-sensitive applications

B. Experiment 2 - multi threaded

The workload choice and explanation of the source code are described in Section IV.A. The workload code builds upon the single-threaded SHA-256 hashing experiment, extending it to utilize multiple threads in such a way that all of the available CPU resources are fully utilized. The experiment tested 23 different memory sizes, ranging from 128 MB to 3008 MB and was performed on both arm64 and x86_64 processor architectures.

In the Fig.6 the relationship between execution time (in milliseconds) and memory size (in MB) for both processor arm64 and x86_64 is shown. Each data point represents the average execution time of a single invocation, with green markers for arm64 and blue markers for x86_64. The results highlight a significant performance disparity between the two architectures, particularly for smaller memory sizes. At 128 MB, x86_64 demonstrates an execution time approximately 270% longer than arm64, underscoring the inefficiency of x86_64 in handling this workload at minimal memory configurations. As the memory size increases, the performance of both architectures improves exponentially, up to 1024 MB, eventually stabilizing around 1792 MB, reflecting full utilization of allocated CPU resources provided by AWS Lambda. At 1792 MB, arm64 achieves an average execution time of approximately 15.4 seconds compared to x86_64's 54.9 seconds. Despite the performance improvements, the relative difference, is the same as in the lower memory range, with x86_64 continuing to stay behind arm64 by approximately 270%, even at the higher memory sizes. These results confirm that arm64 hardware level optimization allows it to perform the hashing calculations much more efficiently than x86_64.

Continuing from the analysis of execution times in Fig. 6, in Fig. 7 the relationship between execution time (in milliseconds) and execution cost (in USD) as a function of memory size (in MB) for both arm64 and x86_64 architectures is presented. The dual-axis graph illustrates execution time on the left y-axis and execution cost on the right y-axis. Data points are connected by solid lines, with green representing arm64 and blue representing x86_64. Shaded regions around the lines indicate 95% confidence intervals, providing insights into the variability of the measurements.

The results in Fig. 7 confirm the trends observed in Fig.6, where arm64 consistently outperforms x86_64 in terms of execution time for all memory configurations. At smaller memory sizes, arm64 achieves significantly lower execution times, for instance the execution time for arm64 at 128 MB is approximately 204 seconds (3.4 minutes) compared to x86_64's 759 seconds (12.7 minutes), resulting in an approximately 270% longer time for x86_64, as previously noted. In terms of cost, arm64 consistently demonstrates superior cost-efficiency for all memory sizes. At 128 MB, the cost for arm64 is approximately $0.00034 per invocation, compared to x86_64's $0.00158, highlighting a cost difference of over 360%. This substantial disparity persists for all memory sizes, with x86_64's costs remaining approximately 350–370% higher than arm64's, even at the larger memory configurations where execution times stabilize. The cost curve for arm64 appears almost as a flat line, suggesting a near-linear relationship between memory allocation and performance improvement. Interestingly, the larger memory sizes, beyond 1024 MB, show a slight increase in cost per invocation, than the lower ones. This reflects the diminishing returns in performance improvement observed in Fig. 6, as execution time stabilizes while costs rise incrementally.

In contrast, the x86_64 cost curve is highly irregular, resembling a spiked line. The worst cost configuration for x86_64 is at 960 MB, with a cost of approximately $0.00166 per invocation, while the best configuration is at 3008 MB, with a cost of $0.00161 per invocation. The absolute difference may seem small, but given enough scaling, translates to a large money sum. This irregularity indicates a less predictable relationship between memory size and cost efficiency for x86_64 in executing this workload. The confidence intervals for both execution time and cost remain narrow for most configurations, indicating the reliability of the benchmark. Minor variability is observed at specific configurations, such as 1536 MB and 1664 MB for arm64, and 1664 MB for x86_64, which aligns with the variability noted in Fig. 6.

These findings further reinforce arm64's efficiency, not only in terms of execution time but also in cost, making it the more economical choice for this multi-threaded SHA-256 hashing workload. The detailed cost comparison is visualized in Fig. 7, where the blue bars represent the average cost for x86_64, while the green bars represent arm64. Above each bar, the orange line indicates the percentage cost difference between the architectures for each memory configuration.

The results in Fig. 8 confirm that x86_64 is consistently more expensive than arm64, with the relative cost difference ranging from approximately 335% to 380%. The largest cost disparity occurs at the smallest memory size of 128 MB, where x86_64 incurs costs 363.4% higher than arm64. Conversely, the smallest cost difference is observed at the largest memory size of 3008 MB, where the relative gap narrows slightly to 335.2%. The cost difference remains relatively stable in all memory configurations, decreasing slightly as memory size increases. Despite this narrowing, the cost difference remains substantial for all memory sizes, consistently favoring arm64 in terms of cost efficiency.

This persistent disparity underscores the critical importance of selecting arm64 for workloads that are optimized for this architecture, as even at its worst-performing configurations, arm64 maintains a significant cost advantage over x86_64. Such findings emphasize the necessity of optimizing configurations for specific workloads. Even seemingly small changes in architecture or memory allocation can lead to disproportionately large impacts on cost. Identifying optimal configurations tailored to workload-specific requirements is crucial for minimizing expenses, particularly in large-scale deployments.

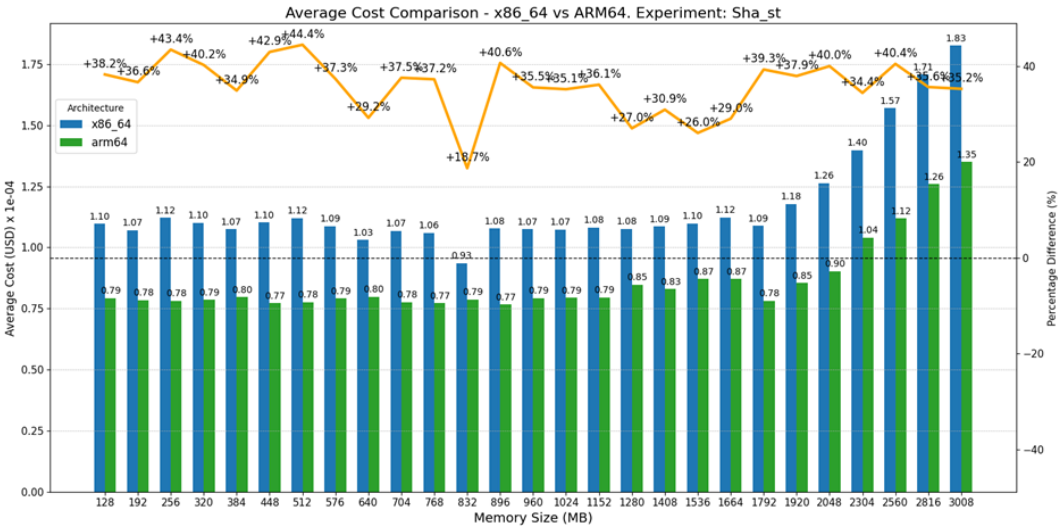

Fig. 5. Average Cost Comparison - single threaded.

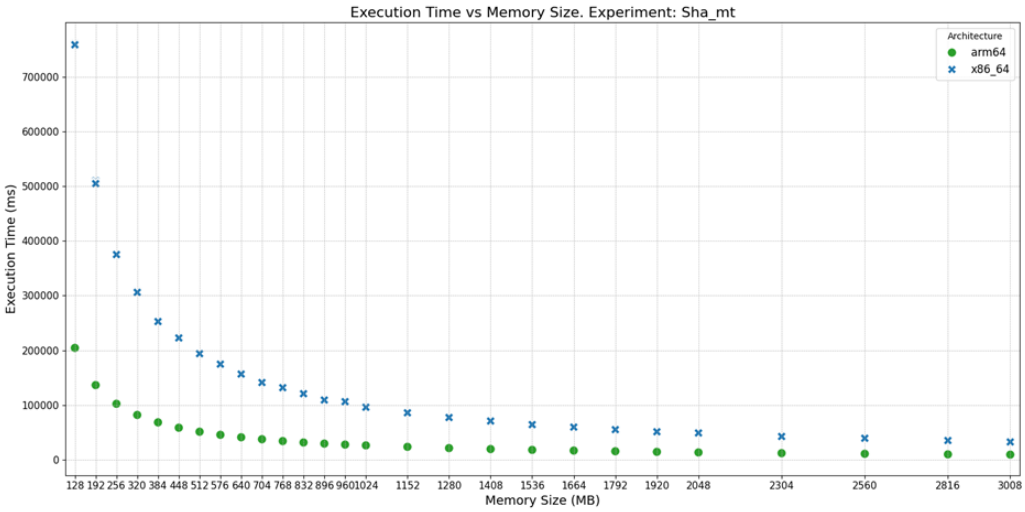

Fig. 6. Execution time vs memory size - multi-threaded.

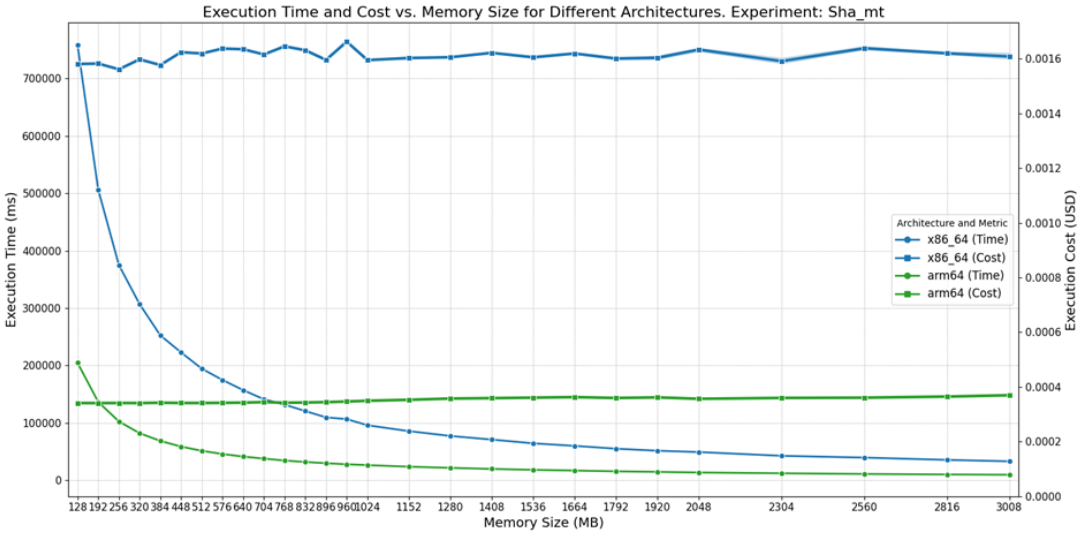

Fig. 7. Execution time and cost vs memory size - multi-threaded.

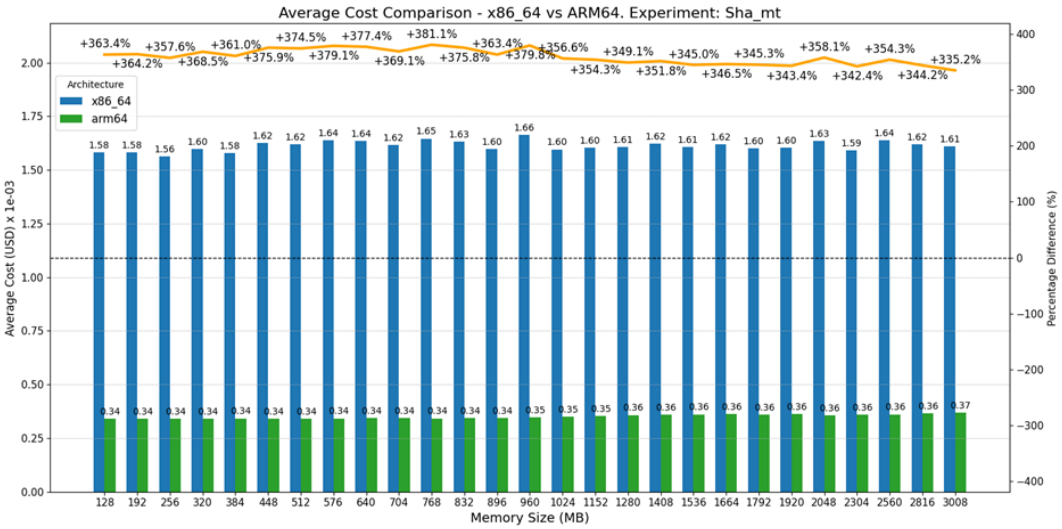

Fig. 8. Average Cost Comparison: multi-threaded

CONCLUSION

The goal of our research was to examine how processor architecture impacts execution time and cost efficiency for different workloads, using LambdaLab to conduct systematic experiments. In this paper we presented experiments with cryptographic hashing SHA-256 algorithm.

For the single-threaded SHA-256 hashing workload, arm64 demonstrated both faster execution times and dramatically lower costs, being over 300% cheaper than x86_64. This highlights arm64's hardware optimizations and its ability to handle cryptographic workloads more efficiently.

In the multi-threaded SHA-256 hashing workload, arm64 again provided faster execution times, however, the cost advantage of arm64 in this case was largely due to AWS's pricing model rather than a significant performance gap. Here, arm64 benefited from its lower price per GB-second, even as the performance of the two architectures converged at higher memory configurations.

In the experiments described in this paper arm64 appeared to be more cost effective but in other our experiment, with Fibonacci numbers workload [26], x86_64 performed better. These observations illustrate that the cost efficiency is highly workload-dependent so a tool like LambdaLab is necessary to benchmark real-world use cases, and identify the most cost-effective configurations for specific workloads.

REFERENCES

[1] P. Mell and T. Grance, "The NIST Defini tion of Cloud Computing," National Institute of Standards and Technology (NIST), NIST Special Publication 800-145, Sep. 2011. https://doi.org/10.6028/NIST.SP.800-145

[2] M. Armbrust et al., "A view of cloud computing," Commun. ACM, vol. 53, no. 4, pp. 50–58, Apr. 2010. https://doi.org/10.1145/1721654.1721672.

[3] "What is AWS Lambda? - AWS Lambda." Accessed: Jan., 2025. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[4] A. Zdanowski, "LambdaLab: a tool for performance evaluation and configuration optimization of AWS Lambda functions, Bachelor's Thesis, March 2025, Inst. of Computer Science Warsaw Univ. of Technology

[5] "Comparing AWS Lambda Arm vs. x86 Performance, Cost, and Analysis AWS Partner Network (APN) Blog." Jan., 2025. https://aws.amazon.com/blogs/apn/comparing-aws-lambda-arm-vs-x86-performance-cost-and-analysis-2

[6] A. Marinos and G. Briscoe, "Community Cloud Computing," vol. 5931, 2009. pp. 472–484. https://doi.org/10.1007/978-3-642-10665-1_43

[7] Z. Li et al., "The Serverless Computing Survey: A Technical Primer for Design Architecture," ACM Comput. Surv., vol. 54, no. 10s, pp. 1–34, Sep. 2022. https://doi.org/10.1145/3508360.

[8] E. Jonas et al., "Cloud Programming Simplified: A Berkeley View on Serverless Computing," Feb. 09, 2019, arXiv: arXiv:1902.03383. https://doi.org/10.48550/arXiv.1902.03383.

[9] J. Scheuner and P. Leitner, "Function-as-a-Service Performance Evaluation: A Multivocal Literature Review," J. Syst. Softw., vol. 170, p. 110708, Dec. 2020. https://doi.org/10.1016/j.jss.2020.110708.

[10] "Intel 8086," Wikipedia. Jan. 27, 2025. https://en.wikipedia.org/wiki/Intel_8086

[11] "ARM architecture family," Wikipedia. Feb. 01, 2025. https://en.wikipedia.org/wiki/ARM_architecture_family

[12] "ARM Processor - AWS Graviton Processor - AWS," Amazon Web Services, Inc. Feb. 03, 2025. https://aws.amazon.com/ec2/graviton/

[13] "Selecting and configuring an instruction set architecture for your Lambda function - AWS Lambda." Feb. 03, 2025. https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch.html

[14] "Serverless Computing – AWS Lambda Pricing – Amazon Web Services." Feb. 03, 2025. https://aws.amazon.com/lambda/pricing/.

[15] B. Ayala, Bryan-0/aws-lambda-benchmark-tool. (Jul. 25, 2024). Python. Accessed: Feb. 04, 2025. https://github.com/Bryan-0/aws-lambda-benchmark-tool

[16] theam/aws-lambda-benchmark. (Dec. 27, 2024). The Agile Monkeys. Accessed: Feb. 04, 2025. https://github.com/theam/aws-lambda-benchmark

[17] T. Yu et al., "Characterizing serverless platforms with serverless bench," in Proceedings of the 11th ACM Symposium on Cloud Computing, Virtual Event USA: ACM, Oct. 2020, pp. 30–44. https://doi.org/10.1145/3419111.3421280 .

[18] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "FaaSdom: a benchmark suite for serverless computing," in Proc. of the 14th ACM Int. Conf. on Distributed and Event-based Systems, Montreal Quebec Canada: ACM, Jul. 2020, pp. 73–84. https://doi.org/10.1145/3401025.3401738

[19] hjmart93, hjmart93/ServerlessBenchmark. (Nov. 20, 2023). Python. Feb. 04, 2025. https://github.com/hjmart93/ServerlessBenchmark

[20] A. Casalboni, alexcasalboni/aws-lambda-power-tuning. Feb. 04, 2025. JavaScript. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[21] H. Martins, F. Araujo, and P. R. Da Cunha, "Benchmarking Serverless Computing Platforms," J. Grid Computing, vol. 18, no. 4, pp. 691–709, Dec. 2020. https://doi.org/10.1007/s10723-020-09523-1

[22] M. Sadaqat, M. Sánchez-Gordón, and R. Colomo-Palacios, "Benchmarking Serverless Computing: Performance and Usability," J. Inf. Technol. Res. JITR, vol. 15, no. 1, pp. 1–17, 2022. . https://doi.org/10.4018/JITR.299374

[23] "AWS Lambda Benchmarking: Rust, Scala, Python, TypeScript - Xebia." Feb., 2025. https://xebia.com/blog/aws-lambda-benchmarking/

[24] J. Wen, Z. Chen, F. Sarro, and X. Liu, "SuperFlow: Performance Testing for Serverless Computing," Jun. 2023, arXiv:2306.01620. Oct. 2024. http://arxiv.org/abs/2306.01620

[25] 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 Computing, Bologna Italy: ACM, Dec. 2023, pp. 7–12. https://doi.org/10.1145/3631295.3631394

[26] I. Bluemke, A. Zdanowski "Experiment evaluating configurations of AWS Lambda functions" submitted to "Practical Aspects of Software Engineering", 51st Euromicro Conf. Series on Soft. Eng. and Advanced Applications, September 2025

[27] "hashlib — Secure hashes and message digests," Python documentation. Feb. 08, 2025. https://docs.python.org/3/library/hashlib.html