Transform-Load-Query Pipeline

Performance comparison: x86 versus Arm/Graviton

Nischal Khadka
MSCSS
University of Washington
Tacoma W.A U.S.
snow11@uw.edu

Sri Vibhu Paruchuri
MSCSS
University of Washington
Tacoma W.A U.S.
svp4@uw.edu

Dev Gandhi
MSCSS
University of Washington
Tacoma W.A U.S.
devg@uw.edu

ABSTRACT

AWS Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. Lambda can be triggered from over 200 AWS services and software as a service (SaaS) applications and you only pay for what you use.This benefits both cloud tenants and providers as this service lead to greater elasticity and better datacenter utilization. Also, tenants don't need to provision or scale their resources.

These benefits can be potentially augmented by choosing the appropriate processor architecture.AWS Lambda allows the user the option of configuring the Lambda function to either traditional x86 or the newer Arm/Graviton processor. However, there is a dearth of information on which architecture to run the Lambda function on to achieve better performance. This paper presents a series of experiments to alleviate this lack of information.

CCS CONCEPTS

• **Computer systems organization → Cloud computing.**

KEYWORDS

Serverless, Distributed Computing, Performance Evaluation

1 Introduction

Serverless computing is a new paradigm of cloud computing where instead of providing virtual machines (VMs), tenants/users register event handlers with the platform (AWS Lambda). When a specific event occurs, the platform invokes the handler. This simplifies the process and provides multiple benefits to both the cloud provider and the tenants.

The benefits to tenants can be boosted by choosing the appropriate architecture the function runs on. AWS Lambda provides the unique opportunity to choose Arm/Graviton processors in addition to the x86 processors that most other platforms provide.

AWS advertises a 19% boost up in the speed of Lambda functions if the user chooses Arm/Graviton processors instead of the x86 processors.

This paper aims to put these claims to test. With a series of experiments on a pipeline of three services. The pipeline consists of three services. The first service fetches data from the Amazon S3 bucket and applies certain transformations to this data. The second service loads the transformed data into a relational database (Amazon Aurora). The third service queries the data loaded onto the database.

The paper compares the performance of Lambda functions running on x86 processors versus the Lambda functions running on Arm/Graviton processors.

We investigate the following research questions:

1) How does the performance of each individual service and the complete pipeline vary based on the type of start: cold start versus warm start?

2) How is the data throughput of each individual service and the complete pipeline affected as the memory configuration of the Lambda functions are changed?

2 Case Study

The experiments were tested on a multi-stage TLQ pipeline that consists of a set of independent AWS Lambda functions. The data used for the application is sales data which is CSV format. The smallest dataset consists of 100 rows and the largest dataset consists of 1.5 million rows. Service 1 fetches the data from the Amazon S3 bucket and performs four transformations:

1. Add a column ‘Order Processing Time’ that represents the number of days between ‘Order Date’ and ‘Ship Date’.
2. Change the values in the ‘Order Priority’ column from ‘L’, ‘M’, ‘H’, ‘C’ to ‘Low’, ‘Medium’, ‘High’ and ‘Critical’ respectively.
3. Add a column ‘Gross Margin’ based on the formula ‘Total profit/ Total Revenue’.
4. Remove duplicate data.

After performing the above-mentioned transformations, Service 1 loads the data back to the Amazon S3 bucket.

Service 2 fetches the transformed data from the Amazon S3 bucket and loads it into a relational database, Amazon Aurora. Before loading the data, It first checks if the appropriate table exists. If not, then it creates the table. If the table exists, then it truncates the table.

Service 3 performs a SQL query based on the data loaded into the relational database based on two filter parameters and an aggregate parameter (groupby).

The application chosen as a TLQ pipeline is a good application of serverless platforms. In addition to this, various experiments can be performed to discern the difference between the performance of AWS Lambda on the x86 processors and Arm processors.

2.1 Design Tradeoffs

This paper attempts to compare the performance of AWS Lambda on an x86 processor versus that of AWS Lambda running on Arm/Graviton processors. Amazon claims there is significant performance and cost-benefit to choose Arm/Graviton configuration over the x86 option. The claim quantifies the benefit at 19%. We expected the performance to be more or less similar. However, the results showed clear benefits of choosing Arm configuration.

2.2 Serverless Application Implementation

All the services have been implemented in Python, so we used Boto3 which is the AWS SDK for Python.

Service 1 receives the Amazon S3 bucket name and the filename of the CSV file to be transformed. Then, the service connects to the bucket and fetches the data. This data is loaded onto a pandas data frame. Since pandas library is not supported on AWS Lambda by default, we downloaded the library files and uploaded them into Lambda layers so that they can be shared across multiple functions. Library files for x86 can be found online. However, the same cannot be said about Arm files. We overcame this problem by launching an Arm-based EC2 instance and downloading the appropriate files. Service 1 performs the transformation on the data frame and loads the data back into a new CSV file. The name of this CSV file is passed to the next service along with the bucket name.

Service 2 connects to the Aurora database using the Pymysql library connection. Again, this library is not supported. So, appropriate files were downloaded. The table name is extracted from the filename of the transformed CSV file. Once the connection is established, the data is loaded into the database once the appropriate checks are completed. Service 2 passes the name of the table to the next service along with the filter and aggregate parameters.

Service 3 performs the SQL query based on the parameters received from the previous service. For instance, if the filter parameters are ‘Country’ & ‘AVG(Unit_Price)’ and the aggregate parameter is ‘Country’, then the query executed would be ‘select Country, AVG(Unit_Price) from sales Group by Country;’

Two versions of each service were created. One on x86 architecture and the other on the Arm architecture. The runtime used across all the functions is Python3.8.

Service 2 has to be configured to the same VPC (Amazon Virtual Private Cloud) as the Aurora database. This leads to a problem as this service also has to connect to Amazon S3 to fetch data. However, it can’t reach the Amazon S3 as it is configured to a VPC. So, to overcome this problem a VPC endpoint was configured so that the Lambda function can connect to Amazon S3.

2.3 Experimental Approach

For Experiment 1 which compares the performance of each individual service and the complete pipeline of the different architectures, we chose a memory configuration of 4096 MB for all the functions. This experiment was repeated twice so that performance can be tested for warm and cold starts.

Experiment 2 compares the performance of each individual service and the pipeline as the memory of the functions is varied.The initial memory configuration is 128 MB and each iteration of the experiment doubles the memory. The largest memory used is 4096 MB. Again, this experiment was

conducted on x86 and Arm architecture so that the performance can be compared across architectures.

The timeout was adjusted to 10 minutes for all the functions.

As stated previously, Service 2 and Service 3 were configured to a VPC as they have to connect to the Amazon Aurora database.

Step Functions state machine was created which orchestrates the pipeline.It ensures that Service 1 and Service 2 and Service 3 are run in succession. The state machine is executed by passing a JSON object which contains the name of the S3 bucket and file name of the CSV file. Service 2 and Service 3 received their requested JSON objects from Service 1 and Service 2 respectively.

To ensure the correctness of the experiment setup, the SAAF framework was used. For instance, the 'newContainer' field of the response JSON was inspected to check the type of start for each of the services.

To pass attributes from one service to another, the SAAF framework was used. The inspector class has 'addAttribute' which can be used to add parameters that can be consumed by the next service.

3 Experimental Results

The results of Experiment 1 can be classified into the following categories:

1. Pipeline - Regardless of the type of start, the execution times on Arm architecture is clearly lower (better) than x86.
2. Service 1 - The results are similar to the pipeline results. The execution times are lower on the Arm architecture.
3. Service 2 - Though in most cases, the performance is better on the Arm architecture. There are some cases when the performance is better on x86.
4. Service 3 - Similar to Service 2, the results are mixed. However, there are some cases (warm start) when the performance on Arm architecture is significantly better.

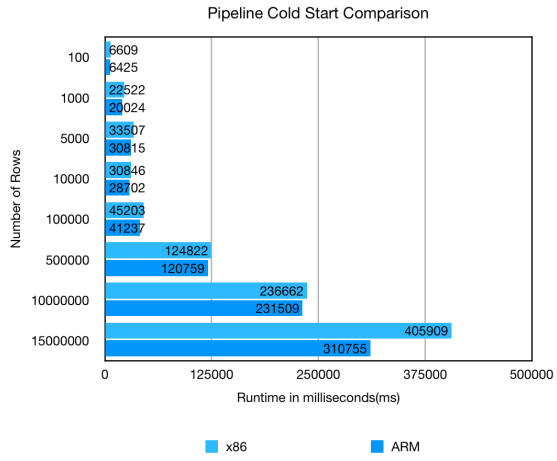

Figure 1: **Experiment 1 - Pipeline cold start performance comparison between x86 and Arm**

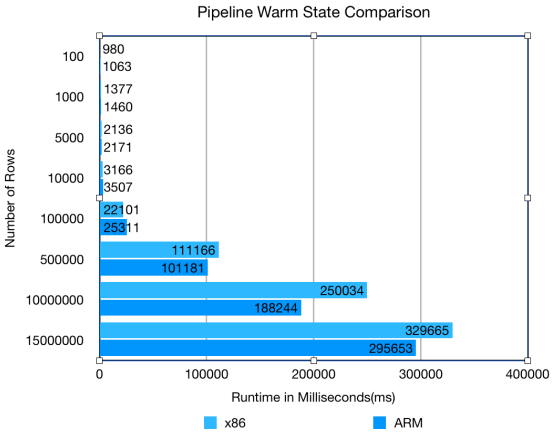

Figure 2: **Experiment 2 - Pipeline warm start performance comparison between x86 and Arm**

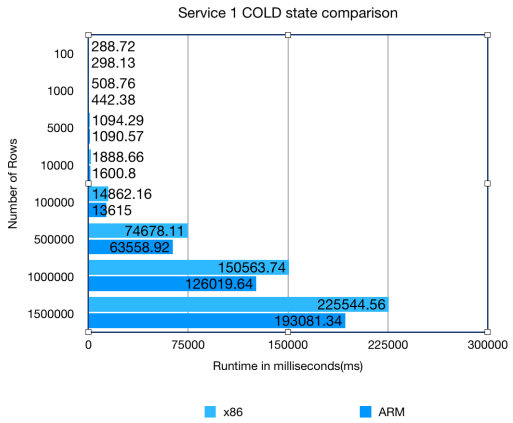

Figure 3: **Experiment 1: Service 1 cold start performance comparison between x86 and Arm**

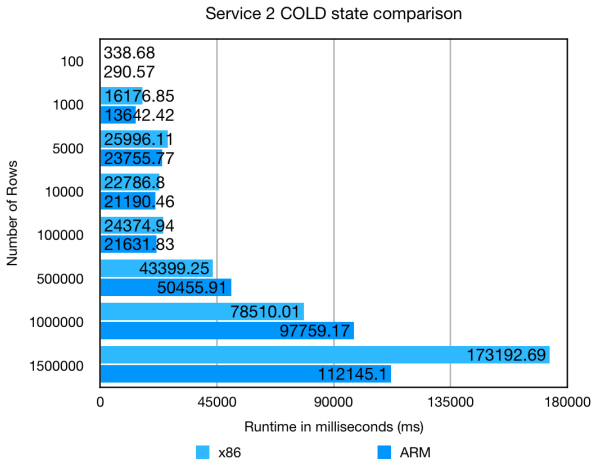

Figure 5: **Experiment 1: Service 2 cold start performance between x86 and Arm**

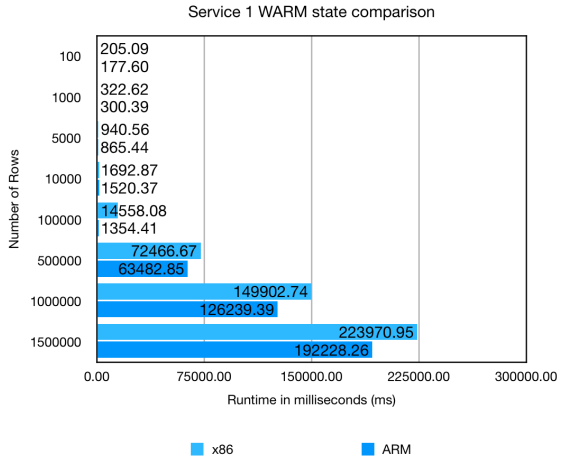

Figure 4: **Experiment 1: Service 1 warm start performance comparison between x86 and Arm**

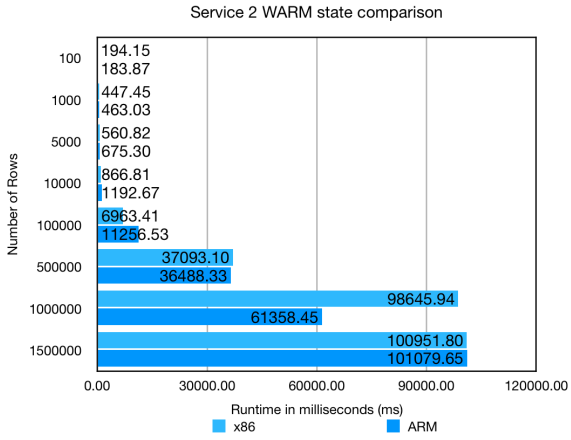

Figure 6: **Experiment 1: Service 2 warm start performance between x86 and Arm**

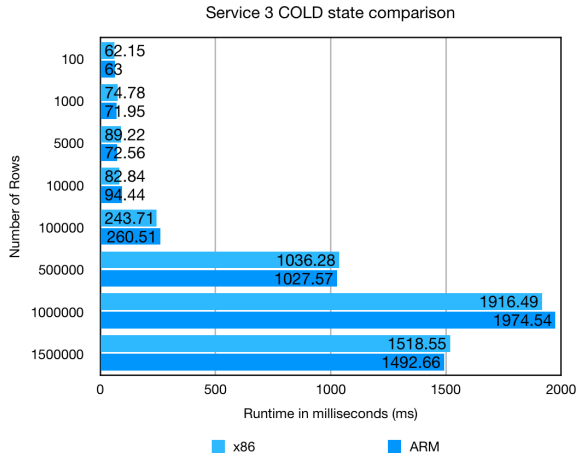

Figure 7: **Experiment 1: Service 3 cold start performance comparison between x86 and Arm**

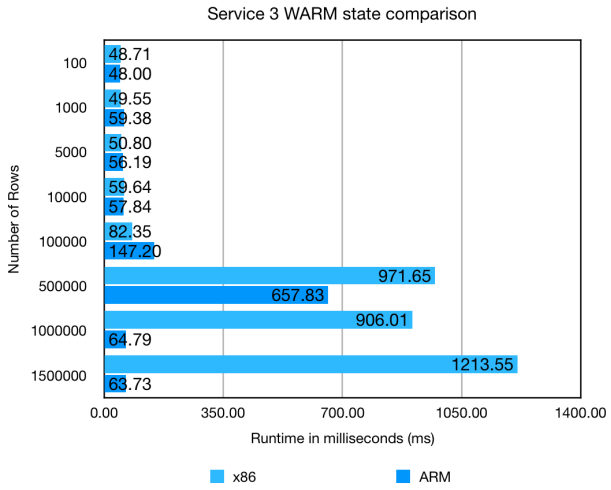

Figure 8: **Experiment 1: Service 3 warm start performance between x86 and Arm**

The results of Experiment 2 can be classified in the following categories:

1. Pipeline - The data throughput is higher (better) when the Lambda function is running on Arm except when the memory allocated is low.
2. Service 1 - The data throughput is higher on Arm in all cases.
3. Service 2 - In most cases, the data throughput is higher on the Arm architecture.
4. Service 3 - There is no clear discernible pattern. In some cases, the lambda function on Arm works better and in other cases, x86 is the clear winner. There are even cases when the performance is fairly equal on both the architectures.

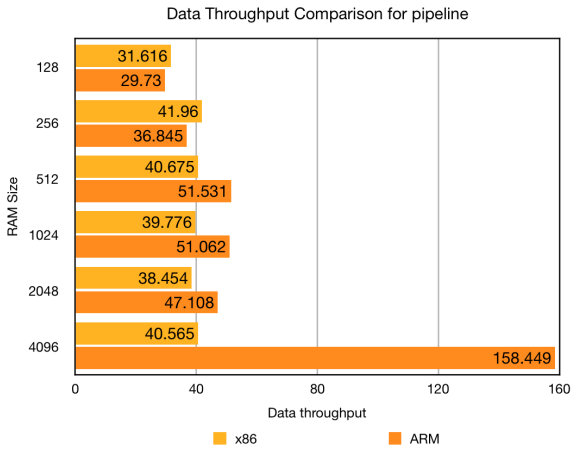

Figure 9: **Experiment 2: Data throughput comparison between x86 and Arm (Pipeline)**

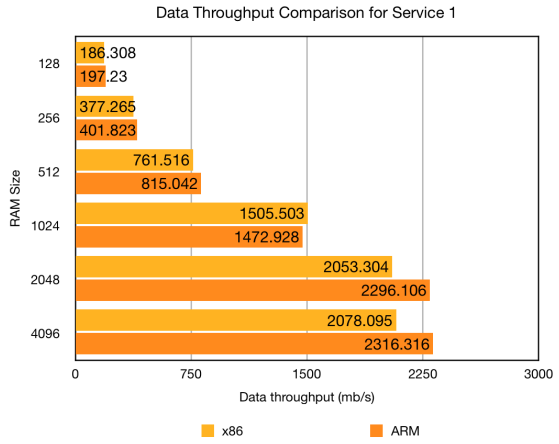

Figure 10: **Experiment 2: Data throughput comparison between x86 and Arm (Service 1)**

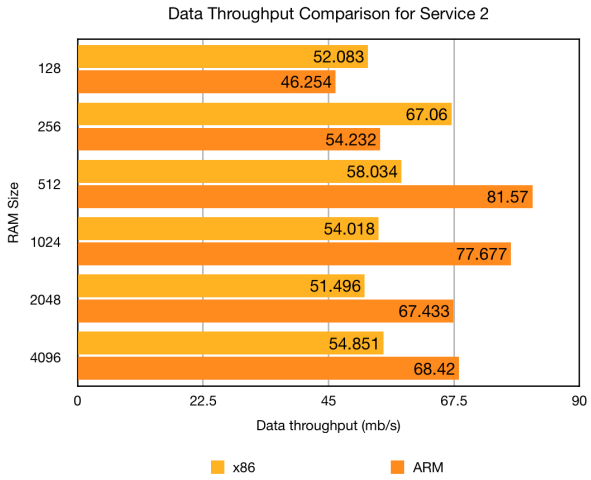

Figure 11: **Experiment 2: Data throughput comparison between x86 and Arm (Service 2)**

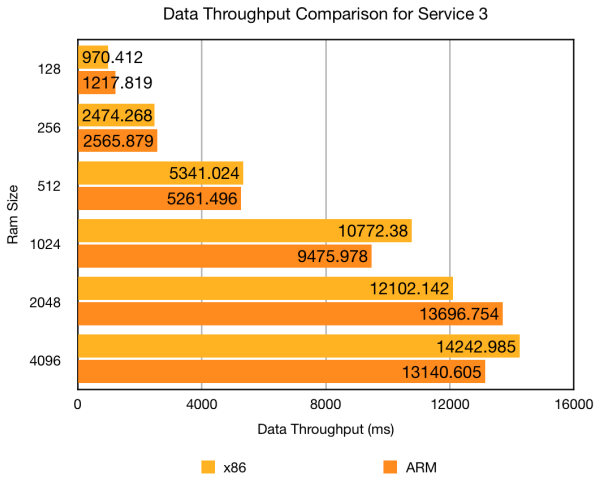

Figure 12: **Experiment 2: Data throughput comparison between x86 and Arm (Service 3)**

Based on the results of the two experiments, we can clearly see the benefits of configuring an AWS Lambda function to Arm/Graviton based processors. In most cases, the performance is significantly better on the Arm architecture. Also, this superiority seems to be increasing as the memory allocated to the function is increased.

4 Conclusions

Based on the results of the experiments, the performance of the Lambda function is better than that of the function on x86 architecture. Though, this is always not the case. Also, this increase in performance is not always pronounced. There are cases when the function works better when it is configured to x86 architecture.

REFERENCES

[1] Sterling Quinn, Robert Cordingly, Wes Lloyd. 2021. Implications of Alternative Serverless Application Control Flow Methods. In Proceedings of 7th International Workshop on Serverless Computing (WoSC7) 2021. ACM, Virtual Event, Canada, 6 pages.

[2] 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 Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '21). Association for Computing Machinery, New York, NY, USA, 161–164.

[3] Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, Wes Lloyd. 2020. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation of Serverless Software Designs In Proceedings of 6th International Workshop on Serverless Computing (WoSC6) 2020. ACM, TU Delft, The Netherlands, 6 pages.

[4] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, Gustavo Alonso. 2020. Photons: Lambdas on a diet. In ACM Symposium on Cloud Computing

(SoCC '20), October 19–21, 2020, Virtual Event, USA. ACM, New York, NY, USA, 15 pages

[5] ACM Reference Format: Yotam Harchol, Aisha Mushtaq, Vivian Fang, James McCauley, Aurojit Panda, and Scott Shenker. 2020. Making Edge-Computing Resilient. In ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA. ACM, New York, NY, USA, 14 pages