2020 **The Master of Science in Computer Science** **A. K. D. M. P. Munasinghe**

Optimizing costs of Serverless: Function profiling approach to optimize memory and running time of Serverless functions

A. K. D. M. P. Munasinghe

2020

Optimizing costs of Serverless: Function profiling approach to optimize memory and running time of Serverless functions

A dissertation submitted for the Degree of Master of Science

in Computer Science

A. K. D. M. P. Munasinghe

University of Colombo School of Computing

2020

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.
To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: A. K. D. M. P. Munasinghe
Registration Number: 2016/MCS/061
Index Number: 16440612

_____

Signature: Date:

This is to certify that this thesis is based on the work of
Mr./Ms.
under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by,
Supervisor Name:

_____

Signature: Date:

Abstract

Function profiling is a mechanism used by developers, performance analysts and architects to evaluate resource usages of a particular function. When coding was done locally and deployed in local or in-house servers profiling was used as a method to evaluate the performance of a program. With the emergence of Serverless the function deployment environment no longer reside on a local Server. The runtime is not accessible for the developer and is maintained by an external entity. With a minimal availability of system information developers are using brute force techniques to identify the required memory parameter for function deployment. The public unavailability of the underlying CPU allocation which correlates to the provided memory parameter intensifies the requirement for the use of brute force methods.

This study presents a Proof of concept and an experimental study to define how function profiling can be used to provide a combined memory parameter out of CPU, memory and running time produced by a profiler. The resulting output is then mapped against a memory configuration for function deployment in a Serverless environment. In the two fold research methodology selection of profilers and the platforms are carried out as proof of concept. Then the proof of concept is converted to an experimental study to deep dive into Serverless platform to observe the behavior of Serverless memory and CPU. Finally, by utilizing regression models a fine-tuned cost estimation function is defined.

The study successfully updated the knowledge over mapping local CPU time to Serverless CPU time based on the memory parameter with a linear regression model. This further optimizes the running time of a function which is expandable in a range of CPU parameters and Memory configurations.

With the understanding of the underlying environments, functions for estimating Serverless function. The profiler techniques, CPU time-sharing and Serverless cost estimation are the primary outputs of the research. Which is then backed with an algorithm definition to develop an analysis tool to achieve cost optimizations in a Serverless platform using function profiler "memory_profiler" and scalene for Python.

Acknowledgements

This thesis would have been a dream for me without support extended by the following noble people as it is because of them I completed this thesis with great confidence and passion.

I would like to convey my sincere gratitude to the supervisor of the research study, Dr. D. A. S. Atukorale, Deputy Director of University of Colombo School of Computing (UCSC), and all the staff members of the UCSC for supporting and encouraging me to become a mindful academic.

I also like to convey my special thanks to Dr. Lasanthi De Silva, who was the coordinator of masters research projects for the immense support given throughout the period of the research study.

I would like to extend my thanks to all other masters colleagues who willingly shared their beliefs, past experiences and insights about the research work I engaged in.

It is my great pleasure to acknowledge all the evaluators and examiners who share their views and valuable insights on my research from the very beginning.

My family members who were always wishing my well-being should be appraised and rewarded with many thanks for baring all the difficulties with me.

Table of Contents

List of Figures

List of Tables

Abbreviations

AWS Amazon Web Services

CPU Central Processing Unit

GCF Google Cloud Functions

IAAS Infrastructure As A Service

KVM Kernel-based Virtual Machine

mprof Memory_profiler

MSA Micro Services Architecture

RSS Residual Sum of Squares

SOA Service Oriented Architecture

Chapter 1

Introduction

1.1 Background

The evolution from Mainframe application development to Serverless application development (Mainframe → Client/Server → Service Oriented Architecture (SOA) → MicroServices Architecture (MSA) → Serverless) welcomed various programming and deployment models to computer science. Each of these paradigms introduced unique characteristics to computing and contributed towards reducing programming complexity, cost-effectiveness, resource management, and improved scalability. Improvements to Program Architecture, Application modeling / Design, and new technologies are part of the major value additions introduced by each of these paradigms. Monolithic and large application systems have been transformed into distributed, small-scaled, decoupled and individually functioning components. Application development and deployment have become a combination of development, DevOps, monitoring, and maintenance. Therefore, developers are concentrating on service distribution, integration, scalability, load balancing, and orchestration of application instead of worrying about the platform configurations.

Alongside the evolution of development practices, a paradigm shift towards Cloud-based IAAS platform have become the infrastructure for the new deployment models. Serverless deployments bridge the gap of requiring specialized experience for infrastructure knowledge on IAAS platforms with configuration less coding. Serverless Computing (or simply serverless) is emerging as a new and compelling paradigm for the deployment of cloud applications, largely due to the recent shift of enterprise application architectures to containers and microservices [1]. Serverless provides a simplified programming model while reducing the knowledge over infrastructure required by the application developer. Also, it provides a cost reduction for cloud

services while paying only for the resources for running time of a specific service. All these flexibilites are maintained at the platform provider end while application developer concerns about the development architecture of the application or service.

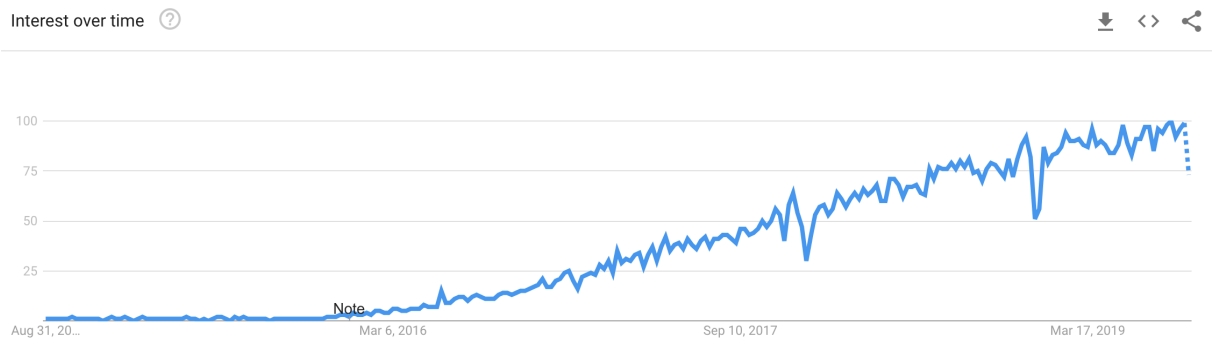


Figure 1.1: Interest Over Time for the term "Serverless" in Google Trends. [2]

Although Serverless has been a novel concept, within two years there has been an increase in Google search trends for Serverless (figure 1.1.) This implies the growing interest in the concept of Serverless. According to Baldini et al. [1] the primary value addition which brings in such attraction is the ability to scale from 0-n while developer not having to worry about the operational aspects such as deployment, maintenance of the code, fault-tolerance and auto-scaling. In particular, the code is scaled to zero where no servers are running when the user's function code is not used, and there is no cost to the user. The same advantage in return has introduced design complexities over managing costs when spikes in resource usage. This is due to the pricing on Serverless depends not on the infrastructure but on the resource usage over time and the number of requests served.

1.2 Problem Definition

As described in Costless by Elgamal et al. [3] the pricing for an AWS Serverless function depends on the following factors.

1. The number of times each function is executed per month (e.g., 1,000,000 executions/month).

2. The memory allocated to the function by application developers. The CPU resources allocated to the function represent an implicit parameter. This parameter value is proportional to the function's allocated memory (i.e., a 256 MB function is automatically allocated twice the CPU speed than a 128 MB function).

3. The duration how long the function runs.

4. The price per 1 GB of memory and 1 second of execution. For AWS Lambda, the price of 1 GB and 1 second is 0.00001667$/GB-s [4].

When generalized across the commercially available Serverless Platforms the cost function primarily depends on two key parameters.

1. Number of requests for each function

2. Amount of compute resources used per execution (Compute Charges).

$$Price_{TotalCost} = Price_{ComputeCharges} + Price_{NumberOfRequest} \tag{1.1}$$

Based on the primary costing factors the abstract function to describe the costs over serverless functions can be described as in equation 1.1.

Table 1.1: Cost Breakdown for Commercial Serverless platforms [4] [5] [6] [7]

Platform	Per request cost (Per M)	Compute cost (GB/s)
AWS Lambda	0.20	$1.67 * 10^{-5}$
IBM Cloud Functions	0	$1.7 * 10^{-5}$
Google Cloud Functions	0.40	$2.5 * 10^{-6}$
Azure Functions	0.20	$1.6 * 10^{-5}$

All the commercial function platforms identified in Table 1.1 use the same primary parameters to calculate the cost. However, for a more fine-grained cost calculation, we should consider all the factors used in equation 1.2 [3] which provides a precise calculation of costs.

$$Price_{TotalCost} = T(C_1 * S * \frac{M}{1024} + C_2) \tag{1.2}$$

T = Total Number of Requests for the period

S = Average compute seconds per execution

C_1 = Cost per GB / Second

M = Memory limit assigned for the function in Megabytes

C_2 = Cost Per Request

The equation 1.2 describes how the listed factors contribute towards the overall cost of a function execution for a given time period. The implication of this function is to manage costs born by serverless function. Therefore developers should be aware of all the above parameters. However, no of requests directed at a function is out of the control of the developer unless there are certain throttling policies involved to restrict the calls. Else the function should be available for any number of calls directed at it.

Considering CPU used for a function equation 1.2 doesn't include clear indications on how CPU is considered for either runtime or the Cost. The CPU allocation for each function is decided by the amount of memory configured for each function. Individually the Serverless platforms have their own method of correlating the memory configuration to underlying CPU allocated for a function.

In the case of AWS Lambda [8] the underlying CPU allocation for memory has a 1:1 linear mapping. However, the same text claims that it is not guaranteed to provide the exact mapping of the CPU to every function. With reference 1.2 how Google cloud functions allocate underlying CPU resources, it doesn't hold a linear distribution as well. Therefore, deciding the direct memory to CPU mapping for a function runtime is in question.

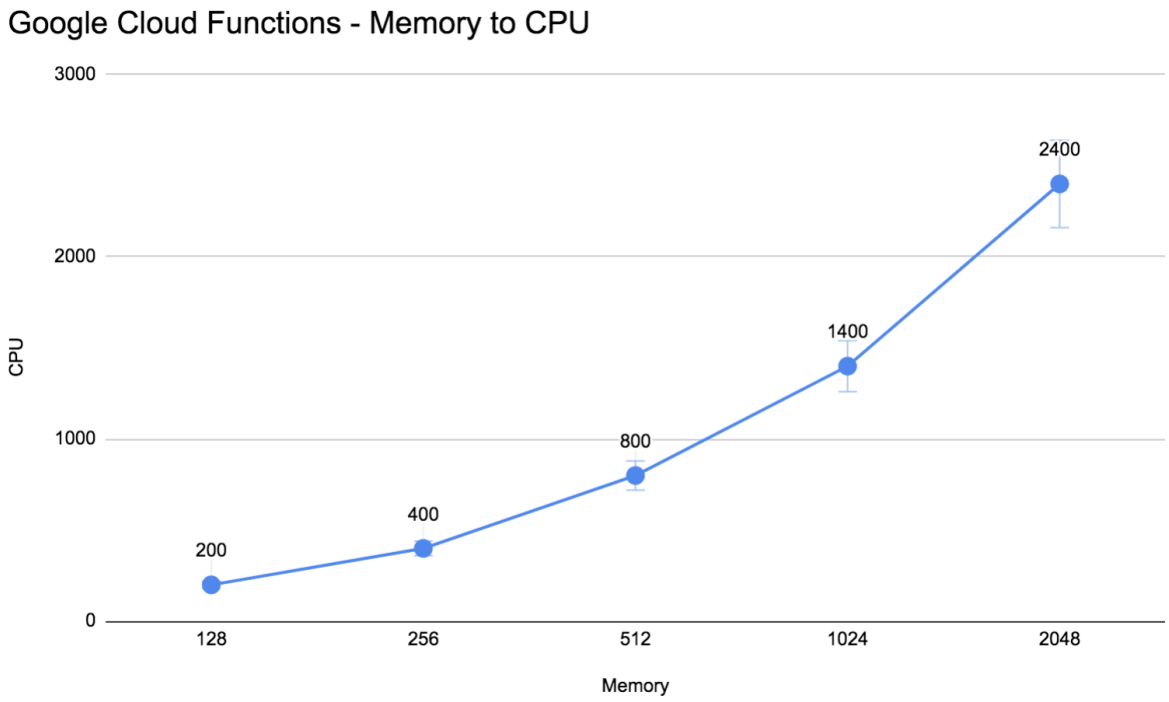


Figure 1.2: Google Cloud Functions Memory to CPU allocation

Eventually given that the CPU allocated is directly proportionate to the amount of memory used, this leaves the developers with making decisions over memory allocated and the running

time of a function. We cannot strictly define that with less amount of memory the function will provide a cost-effective running time. Also if we increase the memory for a function we would improve on the time but pay extra for the memory that is not utilized. Hence, developers require to find a balance between the allocated amount of memory and the optimal running time expected from a function.

This leads us to the problem of how can a developer determine the optimized memory parameter for a Serverless function before it is deployed in a Commercial Serverless platform.

1.3 Aims and Objectives

The research problem has sub-components that can be extracted to formulate the primary objectives of the research. The following can be identified as the primary objectives expected by this study.

1. Identify function profiling approaches to obtain optimized CPU and Memory parameters for Serverless functions.

2. Provide an algorithm to map memory and CPU parameters taken from profilers to vendor-specific memory configuration.

3. Reduce overall Serverless cost by optimized memory and running time parameters.

1.4 Scope of the Study

The intention of the study is to identify profilers which is capable of executing a Serverless function locally. Then the profiler outputs are used to evaluate the running time variables on a Serverless platform to provide an estimated output of the potential running time. Within the scope of this study the intention is to provide an estimation for the running time and the total cost.

1.4.1 Function Profilers

With the availability of multiple function profilers currently available the study is scoped into identifying a profiler out of the existing profilers. Rather than to re-invent a profiler that is capable of providing the CPU and Memory parameter is considered. Based on the need the changes were introduced to profilers such as function entry points to adapt to the nature of Serverless functions.

1.4.2 Runtime Language

With the nature of Serverless functions if a pre compilable language is used there will be an addition of compilation time for the container startup. This also adds memory overhead on the function execution which will create outliers on the results. Hence function runtime language is selected as Python. Since the Serverless platforms tend to provide the latest versions of runtimes. For Python the version selected is 3.8.

1.4.3 Experimental Setup

The experimental setup consists of two primary environments one of which is a local developer environment and the other an existing serverless platform. The local developer environment is used to estimate the parameters for the function from the developer perspective and then it is deployed on the Serverless environment with the values derived through the function 4.18. The Serverless platforms used to run the setup is AWS Lambda and GCF.

1.5 Structure of the dissertation

Following is the summary of how each of the chapters are organized in this study. In chapter 2 the background which relates to the current study is analyzed using already published literature by other researches. By providing a contrast between recent literature the possible research gaps are identified to create the research questions addressed in this study.

In chapter 3 the research methodology followed in this research is described. The usage of mixed-method research methodology is describe on how the POC and experimental study is designed. The background functions, experimental setup and selection of technologies is described further in this chapter. Finally the expected research outcomes are listed.

Chapter 4 includes the details on how the proposed solution is implemented with the options for multiple regression functions. The outline for how the research is analyzed step by step to achieve the final outputs are further explained in this chapter.

With the completion of proposing the solution with multiple options for evaluation the next step is to evaluate and obtain results for each. In chapter 5 the evaluation process is explained on how the selection of profilers and estimator functions are finalized. This includes the summary of the results obtained during the evaluation which are discussed in detail to provide required support for the claims made.

Finally in chapter 6 the research is concluded with a discussion on obtained results and the potential future extensions of the study.

Chapter 2

Literature Review

2.1 Serverless Architecture

From the transformation of just being a buzz word to a real-world concept, Serverless has transformed quite a lot from its inception. Therefore, a lot has changed in between and the research areas have been updated constantly. Understanding the current architectures and how Serverless is leveraged in the present context is critical. To understand the current state of the Serverless platforms we can refer the Thesis by Bolscher [9].

This study has extensively discovered the challenges faced in Serverless space concerning widely used cloud computing concepts. two out of five research questions addressed in the study concentrates on providing a solid background for what is Serverless. When considering the strong areas of Serverless discussed in the thesis it has covered

1. Granular Scaling of Serverless
2. Resource optimization
3. Reducing Operational costs
4. Novel Architectural opportunities
5. Improvements to current development processes

Also, as part of the advantages of Serverless the research also has covered the drawbacks and current challenges faced by Serverless.

1. Performance challenges
2. Development and Operational complexities

3. Function composition
4. Monitoring complexities
5. Migration efforts
6. Threats of vendor lock-in
7. Security concerns
8. Reducing operational costs

With in-depth explanations and citations to the above points, the researcher has comprehensively analyzed the improvements gained by developers. While shifting over to the Serverless platform the revelations on potential drawbacks help developers to avoid taking unaccounted decisions while adopting Serverless.

The research methodology applied in the study Design Science Research Methodology (DSRM) is considered a design science research methodology used in Information Systems (IS) researches [10]. The intention of this methodology is to provide a Serverless architecture design framework for Serverless architects based on the best practices identified through a thorough back ground study. The same methodology opens up space to test the framework and evaluate with domain expertise. Since, the evaluations are based on case studies rather than on empirical data or evidence the adopted approach is a viable approach for the study and the researcher has followed the defined methodology accordingly.

Rather than just being an Architecture Serverless had to evolve as a technology which incorporates to the current use cases of the consumers. The design and implementation of such a platform was considered a huge effort due to technological advancements required. The study by Scott et al. [11] discuss in-depth on how the datacenter evolution with virtualization has achieved the feat of successfully creating an opensource Lambda platform identified as OpenLambda. Within this research the evolution of the Serverless concept is identified and conveyed on how the evolution of the non shared machines converted to application-level lambda sharing environments. Also, the performance gains achieved by simplifying the function deployment is further discussed at the background study to start the research.

Later an in-depth analysis of the Serverless programming model, advantages of Lambda architecture and the design principles attached around the development of OpenLambda is discussed. The technological concerns such as execution engines, Interpreted Languages, Package support, Cookies and session management, Databases and data aggregators are

discussed in detail. For each of these components the benchmark values to be considered and the data models are defined so that a future adopter would be able to go through the fundamental areas of Serverless implementation technologies.

While the following quote "Achieving these cost reductions however is not as straightforward as it might appear, the potential savings of serverless computing 'heavily depend on the execution behavior and volumes of the application workloads'" [9] conveys the idea that Serverless still needs means of reducing costs over deployments. This is due to the volume of application workloads are out of the control of developers at the moment. Hence this particular study has concluded that the current solution of estimating costs for Serverless is to deploying the code and running it for the expected load for some time. Which is one of the conclusion I tend to disagree with since later efforts like Costless [3] [12] has thoroughly studied on estimating beforehand and reducing costs of Serverless functions.

Therefore, the current research will be used to strengthen the above claim that cost of Serverless can be estimated beforehand and provide a meaningful approach to estimate the volume of application workload beforehand.

2.2 Virtualization for Serverless platforms

In Serverless environments the execution environments are virtual machines which are running in a shared environment. The demand for cost-effective deployments is a concern of the platform provider. Adopting virtualization environments have become the best solution for reducing the cost of deployment, instance autoscaling and to achieve efficiency in hardware resources. The study by Alexandru et al. [13] describes how AWS Lambda [14] utilizes virtualization for the underlying runtime environment. Within the study the researches who are from Amazon Web Services team has explained how the traditional Virtualization techniques are optimized to be used with Serverless platforms.

The options of Container-based isolation, language-specific isolation and hardware-based virtualization are considered while building up Firecracker [15] as a KVM (Kernel-based Virtual Machine) for AWS Lambda. Achieving multitenancy through virtualization and the rationale behind the selection of Linux kernels is described as a decision made based on security, code compatibility and memory footprint of the kernel. On top of the selection of Linux kernels further evaluations are done based on isolation of hardware, overhead and density of running thousands of functions within a single virtual machine, performance compared to running a native function against a virtual container, compatibility against various Linux binaries without

modifications to the original binary, Fast switching between functions and Soft allocation of CPU and Memory resources. The evaluation performed to measure the properties described above to produce a comparison between Boot time, Memory overhead and IO performance. The best evaluation the study has received is the successful production deployment of AWS Lambda where it processes events in trillions every month. The findings made within the study by Alexandru et al. [13] has been used within AWS Lambda since the version 2 of Lambda platform.

Apart from using Linux kernels for Serverless environments the Microsoft also rethought their strategies on providing a Serverless platform on Microsoft Azure. The study by McGrath and Brenner [16] presents how a Windows container-based function deployment and execution platform is built. The objective of the study is focused on providing a performance-oriented Serverless computing platform on top of Windows servers. The implementation of the study utilizes .NET, Azure storage, Azure storage queues, Azure storage tables and Azure storage blobs which are bound to technologies provided by the Azure platform.

With compared to AWS Lambda the Azure functions [17] instead of running a KVM based virtual machines it utilizes Docker with Windows Nano Servers. The underlying docker images are built with the respective running time language to allow the execution of multiple function languages. The execution function is attached to a container using a read-only volume which is attached during the container startup.

With the implementation of the prototype platform the study by McGrath [16] has done an in-depth comparison against AWS Lambda, Apache OpenWhisk, GCF and Azure functions. Which improves the current performance of Azure functions. However, compared to other Serverless platform performance on concurrent requests and back off time both the prototype and the Azure functions are very low in performance.

Although both studies by Alexandru et al. [13] and McGrath [16] describes how respective Serverless platform architects its underlying virtualization environment both of the studies doesn't mention how the CPU allocation is performed. For AWS lambda the only information about the underlying CPU allocation is in the form of [18] where it refers to the CPU allocation as proportions to the memory. In the study by McGrath [16] and Azure documentation both confirms that the CPU allocation is decided proportionally based on the memory parameter. Which opens up a gap to be researched into.

2.3 Costs of Serverless vs Monolithic architectures

To bring out a solid argument on the savings achieved by a certain technology one should compare and contrast it to the other existing technologies. Monolithic and microservices architectures being the predecessors of Serverless architectures the ideal comparison would be to compare costs of Serverless with these architectures. The study by M. Villamizar. et al. [19] has addressed this concern. Using a case study approach the researchers have created environments for Monolithic, Cloud, Microservices and Serverless to execute the same functionality. This application has a simple API exposed which accesses a backend relational database to retrieve some information.

Application functionality has been maintained consistently while the underlying deployment architecture changes according to the deployment pattern. Based on the deployment architecture each of the selected platforms has introduced various points of costs due to the requirements for that specific architecture. This means for each deployment developers had to worry about different points where costs will be introduced and saved. For microservices, it was the additional number of nodes and methods of scaling. For Cloud deployments the costs for the vendor and gateways. For Serverless in this case AWS lambda function run time and memory parameters.

The external elements such as internet access from a local machine and the connectivity to the backend database is kept consistent in all the architectures which is a key point in the study. Also, when it comes to language selection the programmer has been stuck to MVC web framework architecture since the requirement to maintain a three-tier architecture was present. Hence, the application which is deployed in each of the platforms has used the Play web framework with Jax-RS. Given the requirement we can decide that the selection of languages in this context is properly validated. However, in order to avoid startup delays encountered by tomcat or other types of heavy weight Java web servers the researchers have opted to Netty and Jetty which are by design intended for faster startup times and consumes less resources. This particular choice of technology raises some doubts on how the developer has accounted for the same startup time delays in Serverless functions. This is due to the fact the selected serverless platform AWS lambda doesn't disclose the information on what type of runtime web servers used within the functions. Therefore, in relation to autoscaling provided by the Serverless platform the comparison between response time can be considered for questions. Also, this has quite a bit of impact for the memory used by the runtime since it should account for the memory utilized by the runtime web server as well. Hence, for the current research we have to consider a runtime which doesn't necessarily require a web server to run the code within environments.

The most highlighting fact about this research is the approach taken to test each of these environments and configurations. Although the same JMeter script is used for testing in order to leverage between two scenarios where the specific services work the 2 web applications with varying deployment parameters are exposed to the tests in a percentile manner (20/80, 50/50, 80/20). For each architecture respectively two services are exposed with the mentioned percentiles. Later with the results produced the architectures are compared for the costs incurred by each of the scenarios. In return comparison of the results between each architecture provides a strong basis to be compared based on how each platform would behave with respect to external stimulators as well as within non-stimulated environments.

With the conclusion, the study has revealed that by moving to cloud and Serverless the developers can reduce costs by a margin of 77.08 % and the higher the number of requests been made to the application higher the savings become. Which in return provides a solid background to the current research that moving to Serverless for the benefit of costs resides at the number of invocations done in a period.

With the future works of the same study the writers has highlighted that the need for evaluation of memory, I/O and network-intensive services used in these platforms. Within the scope of the study they have only evaluated the usage of single memory parameter and single platform (AWS Lambda). The usage of a memory parameter has clearly affected the study since it is a key cost reduction factor. Also, since these applications and the lambda functions are not memory, CPU intensive the parameters they have used in the configurations might have been reduced. Hence, resulting in more cost savings and further extending the outcomes of the research. This highlights the fact that the requirement for pre analyzing the memory parameters even for the contexts of research is still not touched and properly studied.

2.4 Costless

Consumers of Commercial Serverless platforms are constantly refining how they manage costs on Serverless deployments. These efforts mainly uses trial and error approaches and tend to be less publicized. Concerning scientific studies focused in this area, Costless [3] has done a thorough study on how should Serverless developers plan, manage and implement functions with the focus of managing costs. This study has individually considered Amazon Lambda and Lambda workflows to provides a thorough and holistic idea of where a chain of functions has its cost implications. Furthermore, this has supported the claims of reducing costs by organizing a sequence of functions considering the total cost.

Rather than purely highlighting the Serverless function the aspect of placing the function in a SaaS vendor is considered in this research. The function fusion approach discussed in this study analyses how complex functions can be split into multiple functions considering the cost factors involved. However, the fusion has been done with the use of AWS workflows and Lambda state transitions. The split functions are then evaluated with several function placements on edge devices, VMs and Lambda to further contrast the cost gains by the use of Serverless.

The identification of the factors affecting Serverless pricing and formal definitions of the pricing formulas are another set of key information that is reusable for further studies. Highlighting the factors affecting the cost and evaluating the impact of changing these parameters is done throughout the whole research. The usage of cost graphs to illustrate the flow of costs can be considered a valuable approach when used in cost-related research.

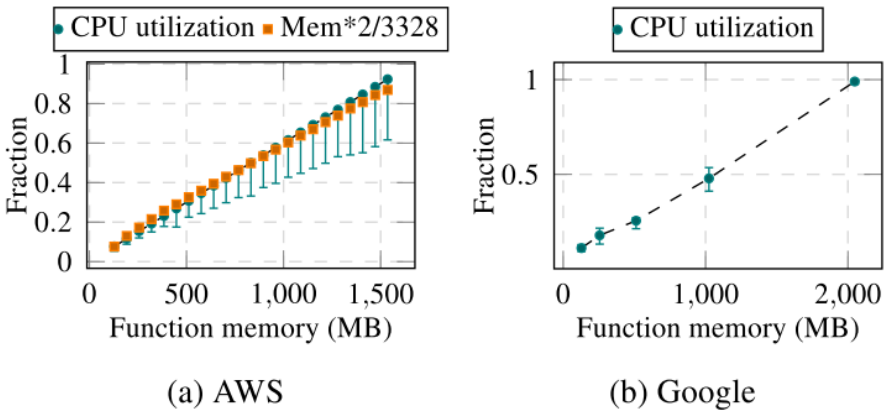
With regards to single function performance and memory parameters, the study has considered several memory parameters. As stated in the publication, the approach toward deciding memory parameters for the function is by using Function Profiling. However, they haven't mentioned or not provided any solid background on this decision and nor they have provided the method or techniques used for function profiling.

As a function, Costless [3] has used an image processing workflow for face detection and creating a thumbnail with an index. This is a solid reference to begin the current study.

Since the current research approach is on how to use function fusion (Collection of functions) and function placement (Function deployed location AWS Edge device or Lambda) the study had to tweak memory parameters based on the outputs of each fusion. Hence the consideration over fine-tuning memory has been considered partially. Therefore, the same research brings out the necessity of a focused study on how to optimize Serverless memory parameters for better results.

2.5 Cross platform resource characteristics

Understanding runtime configurations of a Serverless platform is critical for the current research as it relies mainly on the underlying CPU and memory relationship of each platform. The requirement to understand how each of the platforms deal with underlying resources for each configuration should be well understood. In [20] the researches have deep-dived into each of the following Serverless platforms AWS Lambda, Google functions and Azure functions. By doing so they have managed to demystify the correlation between memory to CPU allocation for each of the platforms.



(a) AWS (b) Google

Figure 2.1: The median instance CPU utilization rates with min-max error bars in AWS and Google as function memory increases, averaged across 1,000 instances for a given memory size. [20]

Using python-based applications to access the host containers of each of the function runtime the study has read the information within. This approach has greatly contributed for the report as it is able to directly read the information. Also, this has revealed that for each of the platforms for the same memory configuration there could be several types of CPU available hence the runtime for each function should may differ. Using the above results they have strengthened the claims done by Chapin in his blog publication [21] where he was able to observer inconsistent behavior of running for a Fibonacci function for various memory parameters. The figure 1.2 illustrates the relationship between CPU for each of the memory parameters.

The behavior of how the CPU allocation changes according to study has its unique characteristics for each of the platforms. Numerically AWS lambda has an increase of CPU utilization rate from 7.7% to 92.3% (SD 0.7% and 8.7%)as memory increased from 128 MB to 1536 MB. With related to Google cloud functions the increase has ranged from 11% to 100%. The ability to co-reside functions in the same container has also been evaluated within the same function and has met with similar results since each of the platforms are managing the concurrent function requests to be able to use the ample amount of resources from the system.

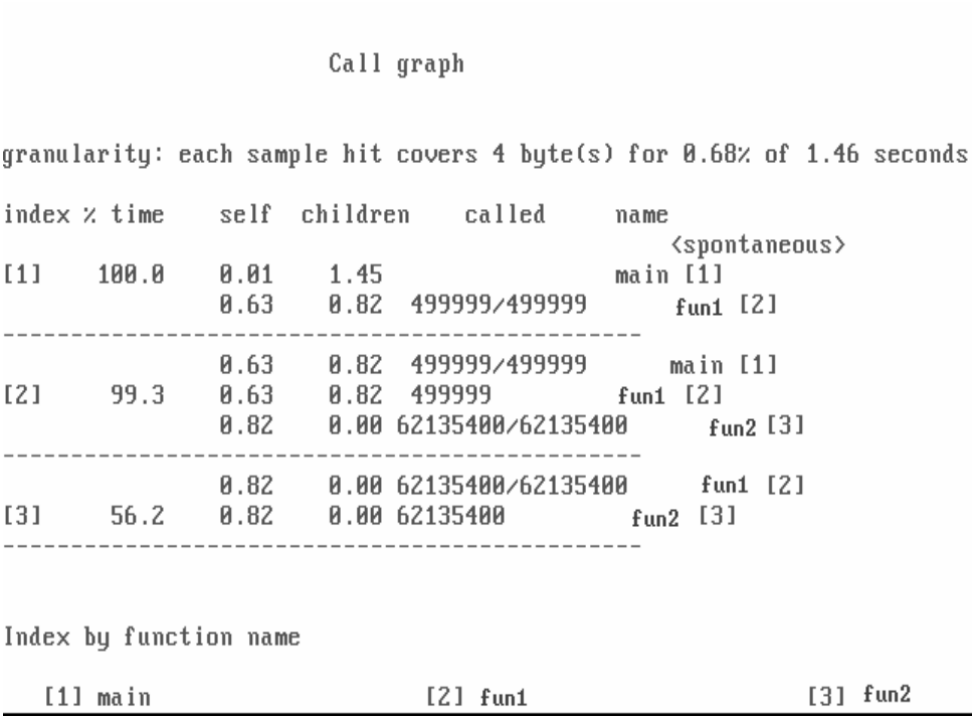
The availability of the source code by the researcher as an Open Source repository allows to update the same values to the date since the study was publicize in February 2018. Also, the study strongly backups on the fact that there is a requirement for an analysis on how developers should consider memory parameters with caution given that the underlying CPU resources tend to vary for each invocation. Hence, the impact on running time for various CPU configurations need to be analyzed before hand.

2.6 Function Profiling

Evaluating a function before deployment may seem impossible, given the nature of function behavior. However, in cases of performance optimization developers and performance engineers use profiling techniques. To gain an understanding of the current profiling techniques [22] in their study to Linux profilers has done a thorough explanation of what are the existing profiling techniques which are used,

1. Instrumentation Techniques
2. Sampling Techniques
3. Line-by-line

These techniques are used in a variety of domains for performance analysis and this particular study they have explained on how Linux profilers are used for Kernel and Code run within.



```
                        Call graph

granularity: each sample hit covers 4 byte(s) for 0.68% of 1.46 seconds

index % time    self  children    called     name
                                                <spontaneous>
[1]    100.0    0.01    1.45                 main [1]
                0.63    0.82  499999/499999      fun1 [2]
-----------------------------------------------
                0.63    0.82  499999/499999      main [1]
[2]     99.3    0.63    0.82  499999         fun1 [2]
                0.82    0.00 62135400/62135400      fun2 [3]
-----------------------------------------------
                0.82    0.00 62135400/62135400     fun1 [2]
[3]     56.2    0.82    0.00 62135400          fun2 [3]
-----------------------------------------------


Index by function name

   [1] main                    [2] fun1                    [3] fun2
```

Figure 2.2: Gprof call graph output [22]

With the output results from the Gprof tool Figure 2.2 it is evident how an Instrumentation approach could be used to get the base idea on how sample invocations are profiled for memory usage for an average period. This provides a base approach towards the current study to provide the feasibility of adopting the existing profilers to get the initial parameters for a valid memory configuration.

2.7 Research gaps

Based on the literature discussed above the following are identified as the research gaps to be filled within this study.

1. Adapting serverless functions for the volumes of load with cost-effectiveness.
2. Multiple studies depending on guessed memory parameters for functions emphasizes the need to pre-define the memory parameters.
3. A deep dive and fine-tuning how CPU allocation works within each Serverless platform to provide a quantifiable memory to CPU mapping.

2.8 Research Questions

Based on the identified research gaps the research questions are derived.

2.8.1 Research Question 1

The analysis of Elgamal et al. [3] shows how they have involved function profiling to provide a ballpark figure of memory parameters. However, there is a requirement to do an in-depth analysis of the use of function profiling to pre-determine the memory parameters of a Serverless function. Therefore the first research question is derived based on the above argument,

What are the suitable function profiling techniques to define memory parameters for a Serverless function?

2.8.2 Research Question 2

As Serverless platforms have their correlation between memory and CPU the memory parameters provided by the function profiler alone cannot be directly used to define the Serverless memory parameters. Hence the requirement to define a formula to convert the profiler output parameters should be identified. Which results in the next research question,

What is the formula that is capable of converting profiler CPU and Memory parameters into the Vendor-specific Serverless function memory parameter?

2.8.3 Research Question 3

With the memory parameters defined on a Serverless Function when reducing costs, the memory parameters should be capable of optimizing the running time of the function as well. In return, the function would provide a balance between the memory and running time to manage/reduce costs spent on a function.

Can the memory parameter and execution time provided by function profiling be used to reduce costs of Serverless by defining the optimal memory configuration?

Chapter 3

Methodology

3.1 Research Methodology

Based on the hypothesis "Function profiling can be used to define memory parameters of a Serverless function to reduce costs by optimizing the running time" the research methodology is formulated as follow. The study uses mixed methodology research which consists of a proof of concept combined with an experimental research figure 3.1. The rationale behind the selection of mixed methodology is to address multiple axis of the research. The proof of concept is used to initially validate the theory that function profiling can be used to provide the deployment memory parameter (CPU and memory usage) for a Serverless function before it is deployed in a platform. Followed by the proof of concept the hypothesis was tested using the experimental method. This is to identify the optimization of costs on a particular Serverless function based on the resulting deployment memory parameter. An evaluation condition that has to be met is to achieve cost-effective memory allocation and running time of the function without brute-force trial on the deployment platform.

The study started with an extensive background study surrounding the current status of Microservices and Serverless. Since Microservices have been a primary enabler for the Serverless eco-system it was considered in the background study to understand the supporting technologies. The background study also required to understand CPU utilization models and memory allocation for functions this provides context to real-world profiling. The literature study was conducted with references to recent studies which covered the current status of Serverless, Costs of Serverless and traditional architectures, CPU and Memory utilization techniques, and existing function profiling techniques. Based on the problem, background and the existing literature research gaps are identified.

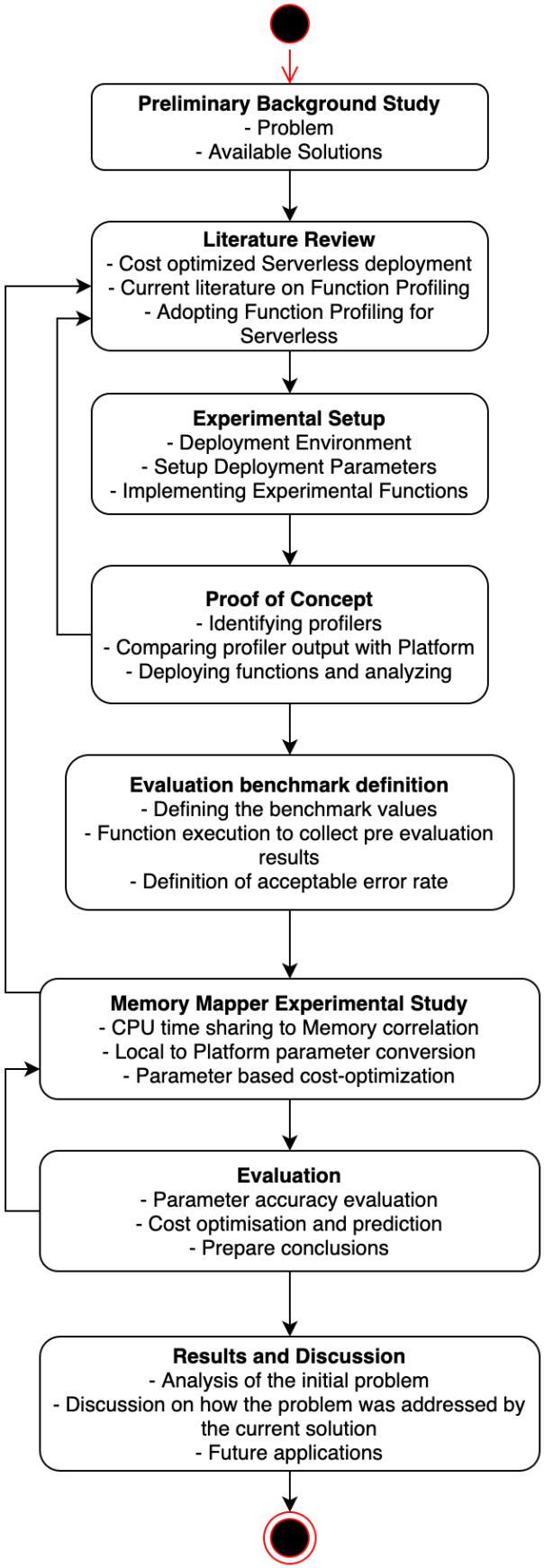


Figure 3.1: Research Design

3.2 Research Design

After the definition of suitable research methodology the research design is carried out to define how each research questions 2.8 are addressed using the particular methodology. The figure 3.1 shows the flow of research on how each step is carried out.

3.2.1 Environments and parameters

The base of the study required to consider the fact of the model's compatibility between developer environment and a Serverless platform. Therefore, maintaining the consistency between these environments was considered from the beginning. Hence, the following control parameters needed to be implemented at the beginning of the research.

1. Function runtime language
2. Function execution and evaluation Serverless platform.
3. Variant memory utilization function based on input parameters.
4. Variant CPU utilization function based on input parameters.
5. Mixed CPU and memory function based on input parameters.
6. Local function environment.
7. Control function execution environment.

Function runtime language

The above elements are utilized with Proof of Concept, Memory mapper experimental study and in the evaluation. Maintaining the consistency between the environments as well as the input parameters were key hence the functions are written to accept input parameters which will determine the resource usage by the function.

Runtime language selection was to be performed on the availability of the language on AWS Lambda and GCF, the dynamic or compiled language in order to avoid latencies added by the cold-start performance issues, high costs and compilation during startup, availability of standalone profilers.

Table 3.1: Language selection matrix [23] [24]

Language	AWS	GCF	Dynamic language	Standalone profiler
Node Js	✓	✓	✓	X
Python	✓	✓	✓	✓
Java	✓	X	X	✓
C#	✓	X	X	✓
GO	✓	✓	X	✓

In addition to the above characteristics Python has been also used in several of the recent studies [20], [25]. Since, the most recent version of Python supported by each of the platforms are 3.7 and 3.8. Giving the priority to the latest version for this study the Lambda functions are written in Python 3.8.

Function definition

The requirement to evaluate the platform CPU and Memory utilization the requirement for variable functions based on input parameters is required. The functions should be able to generate load on CPU and Memory individually. Although total independence between CPU and Memory cannot be achieved the functions required to be focused on each parameter individually while keeping the other low. For this requirement following key characteristics are considered.

- CPU function should utilize 100% of CPU for the total period of execution.
- Memory function should be able to utilize various memory values based on an input variable.
- The ability to calculate the function running time.
- A combination function for both CPU and Memory utilization.

Listing 3.1: Variable memory function

```
def memory_func(value):
d = {}
i = 0;
for i in range(0, value):
    d[i] = 'A' * 1024
    if i % 1000000 == 0:
        print(i)

if __name__ == '__main__':
    memory_func(2000000)
```

Based on the above concerns there were two initial functions defined. A memory function 3.1 which creates in-memory arrays for the number of values provided as input. The number of arrays created changes based on the input variable and the memory utilization varies accordingly. The correlation between the input integer to the highest memory used by function is linear. When mapping with the lambda or GCF environments simply the code is changed at the point of access which does not affect the algorithm and the code lines within the function.

The next function 3.2 is a Fibonacci sequence which uses CPU to the ≈100% to calculate the Fibonacci values for the defined range. Based on the amount of CPU allocated for this particular function while utilizing CPU to ≈100% the function running time depends mostly on the available memory to utilize the function. When CPU time-sharing is involved the same function uses the maximum CPU share to ≈100% for the sequence calculation. The key advantage of using this particular function is, based on the CPU power allowed to the function the running time of the function varies. Therefore, this function is used to correlate running time taken between local, control and function platform environments 3.2.2.

Listing 3.2: CPU Fibonacci function

```
def fibFunct():
    nterms = int(input("How many terms? "))
    # first two terms
    n1, n2 = 0, 1
    count = 0
    # check if the number of terms is valid
    print("Fibonacci sequence:")
    while count < nterms:
        nth = n1 + n2
        # update values
        n1 = n2
        n2 = nth
        count += 1

if __name__ == '__main__':
    fibFunct();
```

The Memory CPU function 3.3 is a combination of both the Memory function and CPU function to generate loads on both Memory and CPU for the evaluation purposes. This function is especially employed to evaluate how the black box CPU time-sharing is converted through the experimental analysis 5.1.

The underlying CPU time sharing of function platform was analyzed with a specialized function which generates maximum CPU load for a period of seconds. The fundamental for this function was extracted from [20] where the Python time() function is called for a consistent period of time. This generates a maximum capacity within each CPU the function runs in. The consistent CPU utilization for a defined time period generated the required information to examine the CPU time sharing percentage from the underlying host machine.

Listing 3.3: Memory CPU function

```
def fibMemCPU(value):
    nterms = value
    # first two terms
    n1, n2 = 0, 1
    count = 0

    d = {}
    while count < nterms:
        nth = n1 + n2
        # update values
        n1 = n2
        n2 = nth
        d[count] = 'A' * 1024
        if count % 100000 == 0:
            print(count)
        count += 1

if __name__ == '__main__':
    fibMemCPU(2000000);
```

3.2.2 Experimental Setup

Due to the nature of the problem the key concern is developers been able to decide the memory parameter of a Serverless function. This should be achieved without involving brute force techniques against a Serverless platform. Therefore, the function should be evaluated in a developer environment prior to be deployed in a Serverless platform. The experimental setup is required to replicate this behavior which will validate the realtime application of the solution. It was initially decided to create two environments Local developer environment where the function will be profiled for deployment parameters, the Serverless environment where the deployment parameters will be tested. Later, it was decided to add a third environment which is a control environment to revalidate the local parameters and to avoid overfitting to the local environment.

Two out of three environments are under the control of the researcher while the third environment which is the Serverless environment is only manageable by the parameters allowed

to public developers. From the two local environments one is the developer environment which is a personal computer and the other been a container running on the same machine. The local developer environment is Mac OS 2.9GHz QUAD Core - Intel Core i7 CPU with 16 GB of memory. The control docker environment is a docker image built on top of python:latest Ubuntu 20.04. The docker file 3.4 is the sample image build file for the virtual environment. Since this control environment is running in a virtualized environment it is also guaranteed that it doesn't get affected with the processes running in the local developer environment. Another key advantage on the control environment is the ability to adjust the resource allocation during the startup to provide varying resources to evaluate the results generated.

Listing 3.4: Docker file

```
FROM python:latest

# install mprof profiler
RUN pip install -U memory_profiler

# copying the function code to the container
COPY MemoryFunction.py /
```

The execution function code is copied to the root (/) of the container for execution. The container is started with the following command **docker run -it –cpus=".1" –memory="1g" mpmunasinghe/research-doc:latest bash**. This will spawn a container with 0.1 percent of total CPUs in the host machine and allocating 1GB of memory. With the bash parameter the bash terminal within the container opens up allowing the execution of the code.

Within this environment when the code is executed with the command **mprof run -M python MemoryFunction.py** the function is profiled and the data file for the profiling output is generated. The resulting output carries the maximum memory parameter reached by the function and the total running time Appendix B.

In relation to the function execution environment two platforms are mainly considered AWS Lambda and Google Cloud Functions. Both of these are selected based on the following capabilities.

1. Available public data on CPU allocation.
2. Consistent allocation of CPU resources.
3. Ability to access /proc/cpuinfo within the runtime.

4. Inbuilt capability to expose functions as HTTP endpoints.

5. Availability of downloadable logs for function executions.

The output parameter from the local environment are used to deploy functions to the two of the secondary platforms. Initially the memory parameters defined by the local function profiler was deployed to each platform which was then evaluated to consider the running time of the function and the capacity of the function execution. Then based on the initial outputs the correctional values are identified for each of the platforms to really match the development environment values to the exact running time values.

Listing 3.5: Variable memory function - Google Cloud functions

```
from flask import escape
def memory_function(request):


request_json = request.get_json()
value = request_json['value']
d = {}
i = 0;
for i in range(0, value):
    d[i] = 'A' * 1024
    if i % 1000000 == 0:
        print(i)
return value
```

With consideration of the changes to the running codes both of the functions discussed in section 3.2 are instrumented to be compatible with the Serverless platform. For both the Google Cloud functions Listing 3.5 and AWS Lambda Listing 3.6 the method accessors and the parameters are the only points that were needed to be adjusted.

Listing 3.6: Variable memory function - AWS Lambda

```
def lambda_handler(event, context):
value = event['value']


d = {}
i = 0;
for i in range(0, value):
    d[i] = 'A' * 1024
    if i % 1000000 == 0:
        print(i)


return {
    "value": value
}
```

Maintaining consistent CPU power was not considered a key parameter, the rationale was to allow any developer environment to be capable of executing the formulas without the consideration of the performance of their developer machine. Therefore, the resultant function should be self-sufficient to calculate the output values based on the provided CPU parameters by the developer as inputs. Although, the CPU power should not matter during the developer executions for the purpose of consistency and normalization the results are obtained maintaining specific CPU performance in both developer (2.9MHz) and control (2.5MHz) environment.

3.2.3 Proof of Concept

Profiler selection

The proof of concept initialized with providing a comparison and a contrast on existing profilers while minimizing the changes to the profiler code. This requirement to use an existing profiler for faster adoption by the users is the basis for this decision. Hence during the POC the profiler capability to provide the maximum memory and CPU usage for a function is considered as a viable profiler. The following Python profilers were considered for the selection mprof (memory_profiler), IOPipe, cProfile, Pympler, Psrecord and bash script based profiling.

IOPipe is a Serverless framework centered profiling tool which is capable of profiling both the memory and CPU of a function. This is very efficient in providing runtime profiling for functions. The major drawback in adopting this solution for the study is that IOPipe profiler is a realtime evaluator which requires profiling within the Serverless platform. Since the requirement to profile the function independent of the platform is in place IOPipe was considered as a rejected candidate.

Pympler is a python package which is an inline profiler. Pypmpler is capable of performing targeted memory analysis to profile memory leaks. This profiler requires package imports to the function and line by line instrumentation where the profiling is required. The nature of manual instrumentation of code requires developers to take an additional effort to update the code lines for profiling. Also, due to the selected profiling within the code lines the overall time for function execution is not captured. The added packages will also affect the overall performance of the function. Therefore, Pympler is also considered a rejected candidate.

Psrecord stood out as a good candidate for profiling both the memory and CPU percentage of the python function. It is simple to use and requires no instrumentation and capable of externally profiling the function during runtime with **psrecord "python CPU-FibonacciFunction.py" –log activity2.txt –plot plot2.png**. Although, this stays as a candidate and is used to leverage the functionality of the chosen profiler due to lack of ongoing development (no code fixes or updates since 6 Sept 2018) and to an unanswered bug on the reliability of the CPU profiling accumulation of CPU exceeding 100% at given instance the tool is to be considered as not a rejected candidate but a suitable candidate as a validator for memory output.

cProfile is a CPU profiler which identifies the CPU usage as an external profiler. The analysis is however not based on the overall performance of the function instead it looks upon each segment of the function. Given the overall output is based on the external monitoring of the function the tool is selected as a candidate for CPU profiling of a function.

mprof memory_profiler is an external profiler to monitor functions memory during the running time. This requires no internal code change and can evaluate the function within the tool itself. Provides a graphical output on the profiled values within the tool. This is selected as the default candidate for the profiler with the exception of having a second profiler to provide CPU usage for the function to further validate the scenario.

scalene is a high-performance CPU and memory profiler which is capable of profiling both CPU and memory during the function execution. The ability to externally execute the code for profiling is a key factor for this profiler. Although it doesn't provide a graphical output of the result the profiler provides a summary result at the end of execution. The CPU profiling

through scalene is capable of providing the average CPU usage of the function for the overall function execution time. The execution time doesn't get affected with the profiler during the CPU profiling however when used for memory profiling the execution time increases significantly. Therefore, scalene is introduced as the CPU profiler tool for the study.

bash scripting is another option for profiling a function. The scripting has to depend on the Top command output or an existing metadata file within a developer environment for the result. A CPU analysis script is used as a supporting CPU analyzer for mprof however it is used only as to provide validation for the claims that a certain function tends to utilize CPU at what percentage in overall.

The comparison on the requirements and features provided by each profiler following profilers are selected as suitable candidates for the research to be carried out. mprof (memory_profiler) and cProfile as the primary profilers and Psrecord and Bash script as the secondary profilers.

Profiler to Serverless adoptation

Next step of the POC is to identify whether the profiler results are able to produce a tangible result to be based at a serverless environment is considered. This is to provide an answer to the Research Question 1 2.8.1. Profiler evaluation is based on two particular parameters generated during the profiling output the maximum memory used and the complete execution time. Based on these outputs a range of Serverless memory parameters +/- 1 configuration levels are tested. Eg:- If the profiler provides a maximum memory result of 182MB the close highest memory configuration 192MB will be applied and will be tested also for 128MB and 256MB. Although, it is obvious that the higher memory configuration 256MB should work, in order to provide a correlation between execution time the analysis on the upper bound is included.

Within the phase of POC it is not required to provide an exact match between the profiler output and the platform configuration. The intention is to prove that the selected profilers are capable of providing a result which can be closely matched (+ or - 1 level of configuration). Which will be then a strong background to enhance the study towards fine-tuning the output parameters using the profiler result to platform time estimator function.

The POC itself is used for the selection of a profiler to investigate the underlying Memory to CPU mapping of the platform. Hence the requirement is to observe how well the profiler provides the memory output irrespective of the CPU utilized. However, to normalize the results the usage of variable memory function and high CPU utilization function is used in the Analysis. The analysis and evaluation on the POC phase is extended to memory mapper experimental study.

3.2.4 Memory mapper

With the results obtained at the POC the foundation for the initial profiler output and Serverless platform parameters the next phase for CPU and memory variant profiling has to be analyzed. The definition of memory mapper is carried out in an experimental methodology where the functions will be defined based on a couple of phases of experimental rounds with profiler outputs and platform executions. This phase consists of the steps as illustrated in flow chart 3.2.

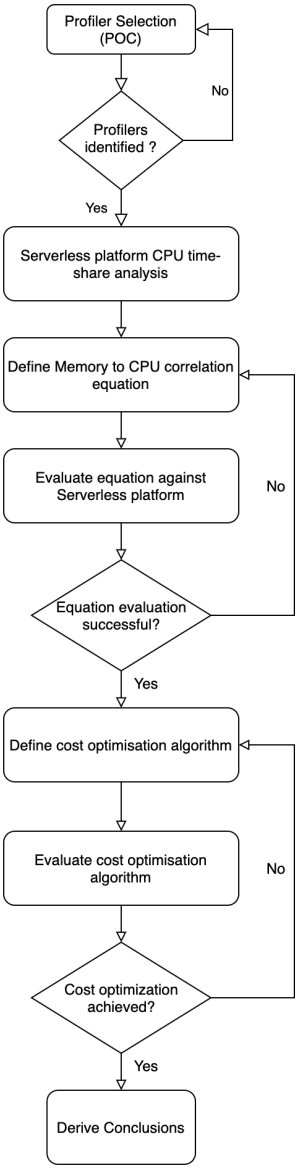


Figure 3.2: Memory Mapper Flow

When reading through [20] the study has already done an evaluation of the CPU fraction allocated by AWS Lambda and GCF. However, due to the recent updates done by AWS and GCF in between the time of the initial research and now it was decided to reevaluate the CPU time shares used by each of the platforms before building up CPU to Memory mapping and

Cost optimizations. Although, the publication done by Elagamal Et al [20] hasn't provided the functions used for this particular evaluation they have provided the outline of the function used. The basis of the evaluation is depended on a Python-based function which generates maximum CPU (≈100%) for 10000 milliseconds. In order to produce the maximum amount of CPU utilization the python time.time() function is called within a while loop until the timeout is met Listing 3.7. The particular function is wrapped with CPU proc stat information to extract CPU usage before and after the function execution. These values are extracted as logs within the platform Serverless function and then later processed to identify the CPU percentages used.

Listing 3.7: CPU time sharing analysis function

```
def lambda_handler(event, context):
    import os
    import time
    from subprocess import Popen, PIPE, STDOUT

    # Capturing process id of the current instance
    pid = str(os.getpid())

    # /proc/process_ID/stat info before the CPU function execution
    processStart = str(os.popen('''cat /proc/''' +
                    pid + '''/stat |
                    awk '{print $14+$15+$16+$17 }' ''')
                    .readline().strip())
    uptimeStart = str(os.popen('''cat /proc/uptime |
                    awk '{print $1}' ''').readline().strip())

    # Reading input value
    value = event['value']

    # printing CPU infromation
    p = Popen("cat /proc/cpuinfo", shell=True, stdin=PIPE,
            stdout=PIPE, stderr=STDOUT, close_fds=True)
    print(p.stdout.read())
```

```
# CPU utilization function
t_end = time.time() + 1
while time.time() < t_end:
    time.time()


# /proc/process_ID/stat info after the CPU function execution
processEnd = str(os.popen('''cat /proc/''' +
             pid + '''/stat |
             awk '{print $14+$15+$16+$17 }' ''')
             .readline().strip())
uptimeEnd = str(os.popen('''cat /proc/uptime |
             awk '{print $1}' ''')
             .readline().strip())

# Clock Tick value of the process
hertz = os.popen('''getconf CLK_TCK''').readlines()

# Printing before and after information in to the log
print("Process_Result:startCPU," + processStart
        + ",endCPU," + processEnd +
        ",startTime," + uptimeStart
        + ",endTime," + uptimeEnd + ",Hertz,"
        + hertz[0])

return {
    "value": value
}
```

The function 3.7 is executed for all the memory configurations provided on the platform for 1000 executions per each configuration. The resulting percentages are then plot against the memory to CPU percentage on a scatter plot. Based on the previous evidence the distribution should be a linear relationship between CPU and Memory. Therefore, a linear regression analysis will be performed on the results to define the relationship linear function. Based on the

executions the scatter plot was generated using the Median and 95th percentile for each of the memory configurations. Therefore, There will be two linear functions to be evaluated against the local profiler outputs.

The evaluation stage 1 is then performed based on the derived linear functions against the primary functions defined for variable memory and high CPU running time functions both locally and on the Serverless platform. For the purpose of variability the function was executed with a range of parameter input both under locally through mprof and on Lambda. The resulting values are compared against the predicted running time of the function against the actual running time in seconds. The suitable function candidate is decided based on actual running time difference to the estimated time. The calculation of the residual sums of square (RSS) is the evaluation metric for the equations. The lowest RSS for multiple CPU values will be evaluated against estimation time and actual Lambda running time. If the RSS is higher the benchmark the function will be reevaluated within the range of memory configurations.

Followed by the linear function evaluation and finalization it is evident that the direct profiler result matters accounts to the fact that memory to memory mapping is predicting the timing very accurately. However, due to the CPU time sharing when CPU is considered for the function execution time the running time is growing rapidly. Therefore, the element of CPU usage analysis and calculation of complete CPU utilization locally has to be factored. Based on this factoring the next phase of optimizing the cost of overall function parameters is considered.

Based on the 2nd output of the study the Memory to CPU correlation function and the total cost function 1.2 identification of the best optimal memory parameter which leverages both CPU and Memory is defined. This is an intersection between the cost rise for the number of expected executions to cost gains achieved by the profiler result analysis. The observation and the requirement raised for this particular extension of the study is due to the CPU time sharing nature of the functions although the function didn't utilize maximum memory the CPU fraction CPU output increased the running time of the function. This eventually resulted longer function running times which will either exceed the threshold running time allowed by the platform or amount in higher costs. Also, with HTTP based calls timeout for request exceeding 30s for most of the browser and tools. Therefore, the ability to either optimize the running time for better cost gains and to avoid timeouts the cost optimization algorithm has to be introduced.

The Total cost function will be further extended with the profiler to memory mapper function first to define the values required to achieve a certain cost objective. Then the memory to time consumption is optimized based on the total cost introduced. Although allowing maximum CPU share will minimize the cost using maximum CPU will require an allocation of a higher

memory which will in return increase the cost exponentially. Therefore, it is critical to identify the sweet spot of memory to CPU ratio for the function deployment to be cost-effective.

The evaluation of the second stage is based on the algorithm's ability to match the pricing with the actual costs calculated by the Serverless platform. This excludes all the external costs on the pricing on the API exposure and will be narrowed down only to the cost of a Serverless execution for a particular number of calls with a certain memory parameter.

3.3 Research Contributions

With the research design and the final outcome following key items are produced as the key contributions for the research community.

- A function profiler suitable for externally profiling memory for Serverless platforms.
- Equation to correlate profiler CPU time sharing against Memory allocated on AWS Lambda.
- Serverless Cost optimization algorithm for variable CPU and Memory configurations.
- Research paper publications on the outputs of the study.
- Dissertation to discuss the process and the outcomes of the study.

Chapter 4

Proposed Solution

4.1 Serverless Time Estimator

4.1.1 CPU time sharing in Serverless runtimes

The application layer of Serverless platforms which are running on top of virtual machines rely on time-sharing for CPU allocation. In order to extract the time-sharing information from the underlying virtual machines the research design has defined the function 3.7. The execution of CPU time sharing function 3.7 for a single execution produced log section similar to Appendix A. The second log line consists of the CPU information of the current virtual machine the function is executed. The amount of full CPU power assigned to the machine and the number of cores assigned for the function is extracted from this particular log line. Out of the produced logs the log line which contains information on the CPU resources utilized by the process are represented in 4.1. This consists of the start and end resource values for the function execution.

Listing 4.1: Lambda Process result log

```
Process_Result: startCPU,9,endCPU,17,startTime,1595.76,
endTime,1596.94,Hertz,100
```

The output from 4.1 result respresents the following variables.

- startCPU (C_s) - Cummulative CPU time at the begining of execution of user code (ucode), kernel code (stime), waited for children user code(cutime) waited for children kernel code (cstime).
- endCPU (C_e) - Cummulative CPU time at the end of execution of user code (ucode), kernel code (stime), waited for children user code(cutime) waited for children kernel code (cstime).

Table 4.1: Lambda Memory to CPU time sharing percentage

Memory Config	128	256	384	512	640	768	896	1024	1152	1280	1408	1536	1664	1792	1920	2048
Median	6.78	14.286	21.296	28.846	35.922	43.137	50.495	57.843	65.347	72.277	79.412	86.139	90.196	99.02	100	100
95th Percentile	7.759	14.815	22.642	29.808	37.255	44.66	51.961	59.406	66.667	74.257	81.188	88.119	94.059	100.99	100.99	100.991
Highest	8.621	16.364	23.585	30.769	38.462	46.078	52.941	60.396	68	75.248	83	89.109	96.04	103	103	103
Lowest	5.738	12.727	19.444	26.923	33.962	40.385	48.077	54.902	62.745	69.608	77.228	79.412	87.129	92.079	94.059	96.04
Standard Deviation	0.587	0.622	0.674	0.738	0.831	0.854	0.837	0.876	0.946	0.912	0.93	1.251	1.67	1.479	1.217	1.012
Proportion	7.143	14.286	21.429	28.571	35.714	42.857	50	57.143	64.286	71.429	78.571	85.571	92.857	100	100	100
Proportion Difference	0.363	0	0.133	-0.275	-0.208	-0.28	-0.495	-0.7	-1.061	-0.848	-0.841	-0.425	2.661	0.98	0	0

- startTime (T_s) - Machines running time since the last boot up from /proc/uptime to the start of function execution.
- endTime (T_e) - Machines running time since the last boot up from /proc/uptime to the end of function execution.
- Hertz (Hz) - The number of clock ticks per second of your machine.

The CPU time share is calculated as a fraction of the overall CPU resources available for the duration 4.1 of function execution.

$$CPU_{usage} = \frac{C_e - C_s}{Hz\,(T_e - T_s)} \tag{4.1}$$

Followed by the function definition the functions are invoked in batches of 1000 executions per each memory configuration for both Lambda and GCF platforms. The execution logs are then extracted for processing and the calculations are made to identify both the median and 95th percentile CPU usage amounts.

The results generated in 4.1 are plotted on a scatter plot to understand the distribution of how the median and 95th Percentile CPU time-sharing is defined on top of AWS lambda servers. The graphical illustration of the data further proved the claim that Lambda provides CPU shares proportionate to the memory allocated to a function. Also, further it proved that the current updated values does have changed from the study performed by Wang [20]. With relevance to the data generated in this study instead of the proportional CPU allocation factor of (2/3328) * Memory identified by [20] has been changed to changed to (1/1792) * Memory. This change is due to the underlying changes from Lambda services has moved CPU allocation to reach 100% at the 1792 MB instead of the previously calculated 1664 MB memory configuration. Hence the original CPU to time share AWS proportional relationship can be described as 4.2. The proportional relationship is constantly increasing until it reaches a maximum of 100 until the configuration parameter reaches 1792 MB and stays consistent until the final memory parameter of 3308MB.

$$CPUFraction = \frac{M}{1792} \tag{4.2}$$

However, the results obtained by the realtime CPU time sharing to Memory allocation provided within the allocation analysis has deferred to the original input values. There were significant changes for the Highest value and 95th percentile distribution of proportional CPU allocation values. In the case of the highest CPU allocation which achieves the best CPU utilization of the deviation is up to 4.29%. With comparison to the average case of CPU allocation it was evident that most of the CPU allocations were below the expected CPU allocation at the middle range of memory configurations from 512MB to 1536MB where most of the servers were under-provisioned with CPU at certain ranges and some ranges being over-provisioned. With this deviation it is evident that there will be significant deviations when it comes to higher amounts of function invocations with higher memory configurations. While low memory functions might perform more than expected.

Therefore, the requirement to further create a more fine-tuned relationship with memory to CPU time sharing with the obtained results for median and 95th percentile are evaluated with linear regression and quadratic regression to create a trendline relationship between memory configuration to CPU time sharing correlation.

Linear regression

The linear regression is performed using the linear regression calculation based on the function 4.3 and values denoted by 4.4

Where linear function is,

$$y = a + bx \tag{4.3}$$

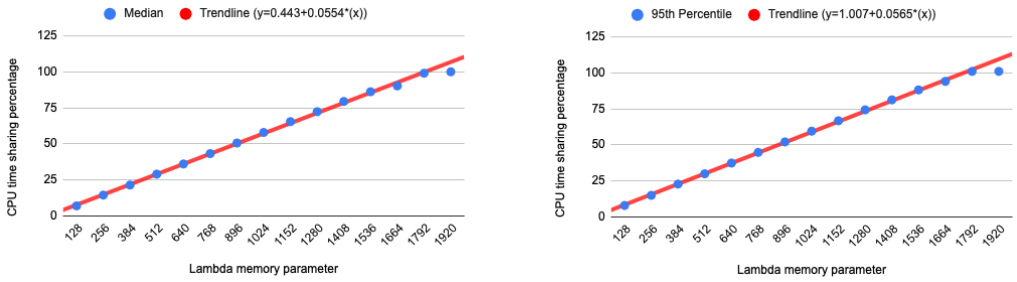
Where a and b denoted by,

$$b = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}$$
$$a = \bar{y} - b\bar{x} \tag{4.4}$$

The resulting regression produced two functions trendlines for Median 4.5 (r=0.9982389493) and 95th percentile 4.6 (r=0.99785488008). Scatter plot and the trendline for each of these functions 4.2 illustrates the near linear relationship as well as the correlation coefficients provide further evidence towards the relationship.

$$y = 0.4431 + 0.0554x \tag{4.5}$$

$$y = 1.007 + 0.0565x \tag{4.6}$$



(a) CPU time sharing median distribution 4.5 (b) CPU time sharing 95th percentile 4.6

Figure 4.1: CPU Time sharing distributions and linear trendlines

One of the key observation is that the CPU time-share fraction increases linearly until the memory configuration of 1792MB. For all the memory configurations above 1792MB the CPU time share is at the 100% maximum hence adding those values to the analysis distorts the relationship. Therefore, regression is performed for the memory configuration range 128MB - 1792MB.

Quadtratic regression

The quadratic regression is performed on the dataset median distribution with reference to 4.7 and 4.8

Where quadratic function is,

$$y = A + Bx + Cx^2 \tag{4.7}$$

Where A, B and C denoted by,

$$B = \frac{\sum (x_i - \bar{x})^2 \sum \left(x_i^2 - \bar{x^2}\right)^2 - \sum \left(x_i^2 - \bar{x^2}\right)(y_i - \bar{y}) \sum (x_i - \bar{x}) \left(x_i^2 - \bar{x^2}\right)}{\sum (x_i - \bar{x})^2 \sum \left(x_i^2 - \bar{x^2}\right)^2 - \left(\sum (x_i - \bar{x}) \left(x_i^2 - \bar{x^2}\right)\right)^2}$$

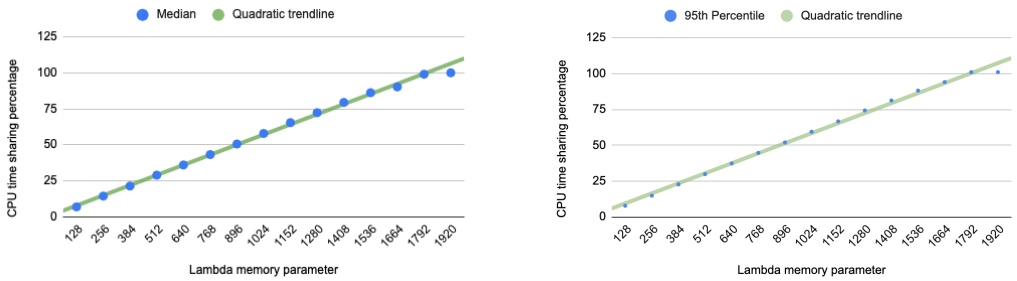
$$C = \frac{\sum \left(x_i^2 - \bar{x^2}\right)(y_i - \bar{y}) \sum (x_i - \bar{x})^2 - \sum (x_i - \bar{x})^2 \sum (x_i - \bar{x}) \left(x_i^2 - \bar{x^2}\right)}{\sum (x_i - \bar{x})^2 \sum \left(x_i^2 - \bar{x^2}\right)^2 - \left(\sum (x_i - \bar{x}) \left(x_i^2 - \bar{x^2}\right)\right)^2} \tag{4.8}$$

$$A = \bar{y} - B\bar{x} - C\bar{x^2}$$

The resulting regression was used to create two quadratic trendlines based on the median 4.9 the 95th percentile 4.10 for the CPU utilization calculation.

$$y = -2.5973 * 10^{-6}x^2 + 6.0402 * 10^{-2}x - 1.2590 \tag{4.9}$$

$$y = -2.1966 * 10^{-6}x^2 + 6.0727 * 10^{-2}x - 0.4325 \tag{4.10}$$



(a) CPU time sharing median quadratic trendline 4.9

(b) CPU time sharing 95th percentile quadratic trendline 4.10

Figure 4.2: CPU Time sharing distributions and quadratic trendlines

Both of the Median quadratic function and the Percentile quadratic function represented following Coefficient of determination R^2 and Residual sums of squares Median function 4.9 R^2 = $9.994804625 \cdot 10^{-1}$, rss = 5.952719313 and Percentile function 4.10 R^2 = $9.998646404 \cdot 10^{-1}$, rss = 1.611871653. In which the highest rss gain in the middle range of memory parameters.

The four regression functions obtained in this section are later used for profiler results to memory parameter conversion and running time predictions. The base function for the actual

Table 4.2: Lambda, GCF CPU values

Memory config	**128**	**256**	**512**	**1024**	**2048**
Lambda CPU	2500.012	2500.012	2500.012	2500.01	2500.008
GCF CPU	2700.253	2700.233	2700.236	2700.294	2700.272

proportion definition is compared to further evaluate and strengthen the claims made by this particular study. With the evaluation phase these functions are later reduced into one function with the most suitable function for a prediction on both running time and the cost for the most efficient optimization of Serverless costs.

4.1.2 Profiler to Serverless deployment

Given the CPU time-sharing is linear to the memory configuration the next step is to transform locally profiled memory and running time to Serverless memory and predict the running time of the function. Since the mprof profiler is capable of providing a direct mapping to the function memory parameter the analysis required is to calculate total CPU time required for a function and convert the local CPU time to Serverless CPU time based on the Memory based time sharing functions identified in section 4.1.1.

Prior to mapping local CPU shares to Serverless shares the requirement to understand the Maximum CPU power provided at Serverless platforms are required. With all the function executions done to calculate the CPU time share the log line with CPU info revealed that both AWS lambda and GCF is using consistent CPU powers underlying for each of the functions. While lambda utilizes 2.5Ghz with 2 cores, GCF uses 2.7Ghz with 2 cores. Which converts to 2500 MHz of Clock rate in Lambda vs 2700 MHz clock rate in GCF Table 4.2.

With the assumption that a local function executes at full capacity for the duration of the function a the product of Clock rate into the total time of execution provides the total CPU time utilized. The same CPU capacity is expected to be utilized at the Serverless platform as well. Therefore the base function 4.11 is defined with CR being the clock rate of the environment and TS the Time sharing percentage for Serverless platform.

In case of a local function is not executed with 100% CPU utilization locally. The scalene profiler output for the average CPU utilization of the function will be used to obtain the total fraction of CPU used locally by getting the percentage of $\mathrm{CR_{local}}.\mathrm{T_{local}}$.

$$T_{serverless} = \frac{CR_{local}.T_{local}}{CR_{serverless}.TS} \tag{4.11}$$

The function is further extended to two functions by replacing the Time Sharing percentage with Memory as the X value to form the four initial Time Prediction functions median time predictor linear function 4.12, percentile time predictor linear function 4.13, median time predictor quadratic function 4.14 and percentile time predictor quadratic function 4.15.

$$T_{serverless} = \frac{100.CR_{local}.T_{local}}{CR_{serverless}.(0.4431 + 0.0554M)} \tag{4.12}$$

$$T_{serverless} = \frac{100.CR_{local}.T_{local}}{CR_{serverless}.(1.007 + 0.0565M)} \tag{4.13}$$

$$T_{serverless} = \frac{100.CR_{local}.T_{local}}{CR_{serverless}.(-2.5973 * 10^{-6}M^2 + 6.0402 * 10^{-2}M - 1.27590)} \tag{4.14}$$

$$T_{serverless} = \frac{100.CR_{local}.T_{local}}{CR_{serverless}.(-2.1966 * 10^{-6}M^2 + 6.0727 * 10^{-2}M - 0.4325)} \tag{4.15}$$

In the evaluation phase the four of these functions are then considered as base functions to predict the time of a Serverless function. Memory parameters produced by the profilers are provided as input with the respective environment values with local Clock rates and Serverless clock rates respective for each of the platforms.

Followed by the evaluation round 1 it was identified that there is a considerable deviation for the predicted time based on the profiler output. It was nearly half of the time than predicted with a 100% error rate for prediction. Further analysis revealed that the CPU fraction calculation is based on one CPU shares inside a host VM Serverless runtime and in the actual environment the VM share resides two instances. Hence, the formula had to be updated to account for the number of co-residing instances inside the allocation 4.16 represented with CC (coresident count) with each of the time share function replacing TimeShareFunction.

$$T_{serverless} = \frac{100.CR_{local}.T_{local}}{CR_{serverless}.CC.(TimeShareFunction)} \tag{4.16}$$

This adjustment doubled the current resource usage fraction eventually doubling the utilization percentage. Followed by this adjustment the second evaluation round is performed.

With the running time estimation and the profiler output on the memory parameter the initial requirements for the deployment of the Serverless function is achieved. From the memory parameter from the profiler the function is capable of deployment in Serverless. By the evaluation based on the Time predictor the memory parameter can be further adjusted for further improvements in the running time of the function.

Table 4.3: Lambda Cost prediction vs Actual Cost

Memory Config	**128**	**192**	**256**	**512**	**1024**	**1536**
Lambda Cost Estimation	257.0146339	257.0146339	257.0146339	257.0146339	257.0146339	257.0146339
Actual Execution Cost	269.3221053	267.4817582	268.0414117	265.5348897	262.3249979	263.2158427

4.2 Cost Estimator

If the proportional CPU allocation for Serverless platforms stand as it is according to Total Cost calculation equation 1.2 the time of the execution is going to be proportionately increasing. In that case for a function which gets invoked for 10000000 (1M) times for 1 Second for various memory configuration would cost the user the same amount for each memory configuration.

According to Table 4.3 there is a significant difference for the costs between the predicted and the actual. This difference is eventually increased as the number of requests increase. Also, the observations made on the actual proportions generated by AWS lambda it was noted that there is a tendency to provide low running times with 512MB 1024MB configuration range due to the behavior of the quadratic nature of the distribution. Due to these two concerns the requirement to analyze the cost deviations for CPU intensive functions has to be analyzed and the final formula for cost estimation had to be defined.

With the initial phase of the study we were left with four functions to be evaluated for the execution time of the Serverless function. Cost optimization has been the goal of the study hence profiler outputs should be mapped with the cost function for the cost optimization. Since the running time of the function and memory configuration are the variables that can be controlled with configurations. The Cost optimizer function needs to take these as parameters. While memory is the only configurable parameter and running time is derived through local CPU time to Serverless CPU time calculation the cost function has to depend on the defined memory parameter.The cost estimation function is then updated to replace function execution time S to $T_{serverless}$ in function 4.17 obtained replacing the TimeShareFunction 4.18.

$$Price_{TotalCost} = T(C_1 * T_{serverless} * \frac{M}{1024} + C_2) \tag{4.17}$$

$$Price_{TotalCost} = T(C_1 * \frac{100.CR_{local}.T_{local}}{CR_{serverless}.CC.(TimeShareFunction)} * \frac{M}{1024} + C_2) \tag{4.18}$$

The function 4.17 is tested for multiple local memory_profiler and scalene executions based predictions with realtime batch run results on a serverless platform. The key optimizations

would be to achieve minimal costs within benchmark execution time (<300s for long running and <30s for short running) for a function. The dependent TimeShareFunction in 4.18 is finally replaced with the relevant time predictor function in the results and discussion based on the selected Time Predictor Function to provide the best-estimated profiler running time and cost.

Chapter 5

Evaluation and Results

5.1 Evaluation

The research problem, background study, and literature review was key inputs for the research methodology and design. The research methodology and design is organized to achieve the three research questions identified in section 2.8. Within the chapter 4 the details of how each of these questions are addressed are discussed while identifying multiple options for each of the questions. Here in the evaluation chapter the discussion is built around how each option is evaluated to decide the most suitable approach to address the research questions.

The first research question on whether there is a suitable profiling technique to define memory parameters is evaluated based on the following parameters.

- Modifications to be done on the function to be profiled.
- Ability to execute independent of the platform on a local developer machine.
- Output should include the highest memory required by the function and benchmark for a running time of the function.
- Profiler should be able to provide the CPU utilization for the duration of the function.
- The function deployed to the platform with the resultant parameters should work without out of memory errors.

The second question is to derive a formula to convert profiler results to the Serverless platform will be tested with deployment parameters figure 5.1. Based on the profiler outputs Memory, CPU utilization and running time the Time estimator function is expected to produce the running time of the function.

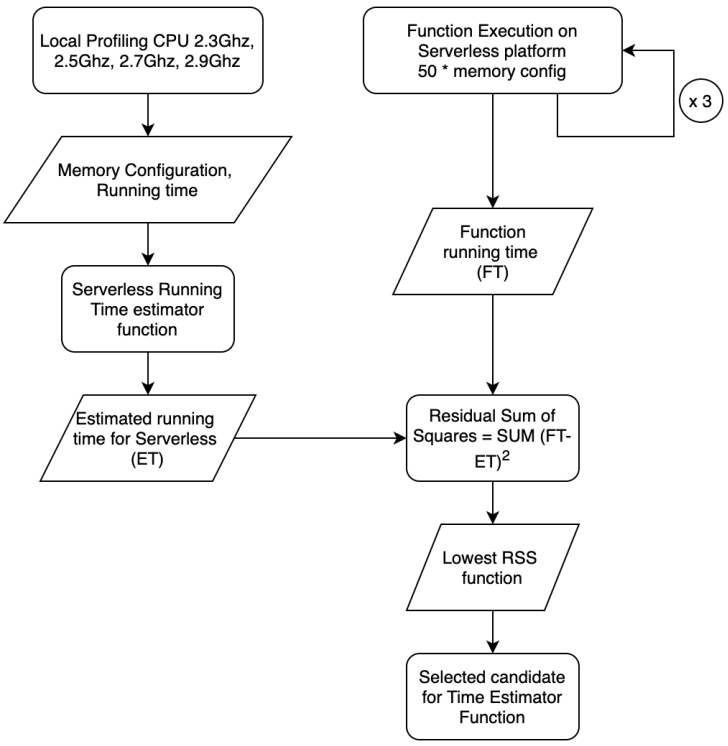


Figure 5.1: Time Estimator function evaluation

Since both estimated memory parameter and CPU usage accounts for the running time the extracted running time doesn't necessarily have to be exact as the local running time. Therefore, the output is compared with realtime Serverless running time for a closer match. The evaluation is based on three key factors to leverage between different local environments as well as the Serverless environment.

1. Multiple CPU powers on local developer environment.
2. CPU time share proportions based on memory configuration.
3. Realtime execution time for Serverless platform.

Based on the above factors three datasets are generated for the evaluation of the estimator functions.

1. Profiler output (memory_profiler, scalene) for processing input for CPU powers with 2.3GHz, 2.5GHz, 2.7GHz and 2.9GHz
2. Profiler output for various inputs to High CPU function, input execution readings from 500000 ,1000000, 1500000
3. 3 x 50 executions for each of the memory configurations from 128, 256, 512, 1024, 1536, 2048

These data is then processed through each estimation function to analyze the success against the actual running time obtained for each memory configuration. The resulting running time is expected to be minimal since an increase of a second than the estimated adds up to the overall cost. This affects the total cost by the total number of invocations which leads up to millions of calls per month, therefore the expected running time deviation is to be kept within +/- 5 seconds for each of the memory configurations. With the calculation of the residual sum of squares the lowest RSS value is selected as the suitable function for the running time estimation.

Final question is to address the cost optimization factors with the profiler outputs gained. The cost optimization function with the provided regression functions are behaving in two different patterns. If the linear regression function is the best fit for time prediction the cost increase will become linear for each memory configuration with minimal differences between the actual cost and the estimated cost. In case of the quadratic function is selected the cost of functions will have a difference which will be more cost-effective towards the mid-range of memory configurations. However, this observation had to be evaluated further to discover the effectiveness of the cost estimation. Based on the time estimation function selected from the previous Time estimator evaluation the total cost estimation is calculated for the functions. The effective cost for each of the actual Serverless executions are then compared against the cost estimation to understand the least error generated memory configurations. In case of a function execution dependency on maintaining fast executions is key and therefore the estimation evaluation also analyzes the speed of execution in the lower range which means higher CPU capacities but with the maximum cost-effectiveness. Later based on the selected function the cost estimation python tool is evaluated against Lambda another round of function execution on the Serverless platform to evaluate the total costs gained or loss due to the function placement based on the cost estimator.

5.2 Results and Discussion

5.2.1 Profiler evaluation

Out of the three research questions the first research question is to find a suitable function profiling technique to define memory parameters for the Serverless functions. For the selected runtime language Python memory_profile package stood as the strongest candidate based on the section profiler selection 3.2.3. In order to address the initial research question the memory_profile has been executed on the function 3.1 with variable array sizes from 1000000 to 6000000 within incremental gaps of 500000. For each of the profiled parameters generated by the profiler the

memory parameters are first recorded. These recorded memory parameters are then matched with available memory configurations for the nearest high configuration for both of the Serverless platforms AWS lambda and Google cloud functions.

Based on the output memory configuration functions are executed in the environment for 5 executions for each function configurations. Resultant configurations from the platforms which performed 100% success rate compared to the profiler outputs are in table 5.1

Table 5.1: Profiler output to platform configuration

Array Size	Profile Output	AWS config	GCF config
1000000	63.44	128	128
1500000	120.398	256	256
2000000	219.137	256	256
2500000	219	256	256
3000000	393	448	256
3500000	418	448	512
4000000	445	448	512
4500000	462	448	512
5000000	477.68	448	512
5500000	480.33	448	512
6000000	779	768	1024

Figure 5.2: Profiler output to platform configuration

The visualization on figure 5.2 shows the clear disparity between the profiler config and the

platforms. The gradual performance increase on the platforms compared to profiler output is considered as a result of increasing CPU resource allocations. Since the local profiler is used to profile on static CPU resources the config increase follows a curvature. In order to understand this further the actual memory utilization in each platform is extracted for the 98th percentile of function executions Table 5.2.

Table 5.2: Profiler output to execution memory for 98th percentile

Array Size	Profile Output	AWS 98%	GCF 98%
1000000	63.44	128	146
1500000	120.398	256	196
2000000	219.137	256	238.3
2500000	219	256	250
3000000	393	422	428
3500000	418	422	408
4000000	445	422	428
4500000	462	423	408
5000000	477.68	423	387
5500000	480.33	426	428
6000000	779	750	731

Figure 5.3: Profiler output to execution memory for 98th percentile

The percentile values show the gain over profiler output in Figure 5.3 due to the aforementioned variation of CPU values. These data reveals that as the next phase the

leveraging of the profiler should be evaluated with the CPU to consider the variant results of the output is due to the CPU performance. The CPU performance changes were analyzed in the design phase of the study under section 4.1.1 this revealed that Lambda relies on proportional CPU allocation. Therefore, it was evident that the running time has to be deviated between the local profiler running time-based on the CPU resources used for the particular duration.

This was the key point of the research to involve a CPU profiler for local profiling instead of relying on a memory profiler. Hence, the introduction of scalene as the CPU profiler for the study. This tool provides a better understanding on the amount of CPU resources required for the function execution.

5.2.2 Time Estimator Selection

The defined regression functions for the CPU fraction calculation are the functions which were subjected for evaluation. Additionally the proportional function estimation is included in the analysis to provide a contrast between the actual mapping against the result output.

In the execution of function profiler approach initially the local profiler values are generated for the defined CPU capacities using the docker container. Where the container is run on top of the 2.9GHz host machine with CPU shares corresponding to 2.3GHz (CPU .7931), 2.5GHz (CPU .8621), 2.7GHz (CPU .9310) and 2.9GHz (CPU 1). In order to normalize the data extracted 100 executions locally for each CPU configuration is profiled and the median selected as the local running time for the particular CPU power. The resulting local profiler running times are,

- For 2.3GHz input 50000 = 5.068s, 100000 = 22.143, 150000 = 44.2292s
- For 2.5GHz input 50000 = 4.946s, 100000 = 20.084, 150000 = 39.5013s
- For 2.7GHz input 50000 = 4.135s, 100000 = 18.434, 150000 = 36.3695s
- For 2.9GHz input 50000 = 4.035s, 100000 = 16.999, 150000 = 35.043s

Then from the three rounds of platform execution the extracted running times were compared for the residual sums of squares for the error rates and averaged for each local CPU power used for profiling. The summary of the results are represented in the table 5.3.

The results suggest that the use of Median Linear function is the function which has a consistent error rate for all three variations of CPU time used. For shorter execution times such as when the input for CPU function is 50000 both quadratic functions performed well for all the

Table 5.3: Time Estimator RSS for profiler CPU

CPU (GHz)	**2.3**	**2.5**	**2.7**	**2.9**
Input = 50000				
Median Linear	44.03898182	20.35692034	22.48768294	8.253268728
95th Percentile Linear	37.70319718	15.32568062	59.75212221	35.90480806
Median Quadratic	7.448440791	28.38961409	2.50010134	8.317139639
95th Percentile Quadratic	6.16933472	0.1698427501	18.09345393	5.375047578
Proportional	1.417489145	2.523542598	8.91239479	1.060728123
Input = 100000				
Median Linear	116.6238024	70.01302425	48.11584507	29.97157302
95th Percentile Linear	42.95843516	62.73264662	80.38219441	104.2680548
Median Quadratic	1112.374918	931.3553036	829.2207438	725.6539977
95th Percentile Quadratic	152.783511	95.76657127	67.66538199	42.92325073
Proportional	328.2695443	238.3682535	190.649746	145.0343491
Input = 150000				
Median Linear	615.8458092	425.4899488	415.1542664	605.5025967
95th Percentile Linear	411.0181311	758.6806978	845.3076761	419.744228
Median Quadratic	5070.307447	3769.763657	3558.56086	5020.733256
95th Percentile Quadratic	1143.540879	857.1866468	829.454089	1129.809522
Proportional	1515.716711	970.8997759	896.5331568	1492.832524

Table 5.4: RSS value distribution for Median Linear Estimation

Memory Configuration	**128**	**192**	**256**	**512**	**1024**	**1536**	**2048**	**Total RSS**
round 1	8.820536275	18.74896214	14.39212752	7.403916737	2.845852389	1.252343569	0.05625351285	53.51999214
round 2	8.448200124	15.55546868	12.34444164	7.182206597	2.655809439	1.206497002	0.04773055228	47.44035404
round 3	6.248843884	12.72346885	13.29620204	7.631262143	2.157683185	1.19594253	0.1337863982	43.38718903

Table 5.5: RSS value distribution for 95th Percentile Linear Estimation

Memory Configuration	**128**	**192**	**256**	**512**	**1024**	**1536**	**2048**	**Total RSS**
round 1	69.40241716	2.384198129	0.005479176132	2.016314455	1.408933131	0.6729850726	0.2007786146	76.09110574
round 2	70.46211206	3.725092555	0.04252196311	1.90142003	1.276187953	0.6394907798	0.1843656966	78.23119103
round 3	77.4572721	5.322686804	0.005369744829	2.135777289	0.9388154443	0.6318130295	0.3325520574	86.82428647

CPU powers however with the increase of execution time has increased the error produced by these functions. When the Lambda proportional allocation function is used for the estimation for long-running functions the error rate is high even though for the short running function it was the lowest error function. Based on the results obtained in this particular analysis Median Linear function, 95th percentile Linear function and Lambda Proportional functions are taken forward.

Further analyzing the data it was a concern to understand why each of these functions were resulting in higher amounts of RSS values even the best performing functions were off with higher margins at the long-running function range. A more granular observation on the results revealed that both of the functions tend to have a larger variation of estimated time and the actual time at the memory configuration of 128MB on the Serverless platform. The table 5.4, 5.5 and 5.6 describes the effect of the memory configuration on Total RSS for each round. The reason for this considerable deviation is due to the fact that with lower memory configurations the execution time is higher which tends to subject for noise created on the CPU resources. Therefore, the estimation values for lower memory configuration has a higher difference with compared to Serverless platform execution.

Table 5.6: RSS value distribution for Lambda proportion Estimation

Memory Configuration	**128**	**192**	**256**	**512**	**1024**	**1536**	**2048**	**Total RSS**
round 1	104.254882	54.69803514	29.29590318	9.042091098	2.876614435	1.199198016	0.07102131725	201.4377452
round 2	102.9650184	49.13795496	26.34090361	8.796900926	2.685529421	1.154343955	0.06140149475	191.1420528
round 3	94.87457397	43.99406237	27.72305764	9.29315105	2.184479566	1.144020629	0.1560947253	179.36944

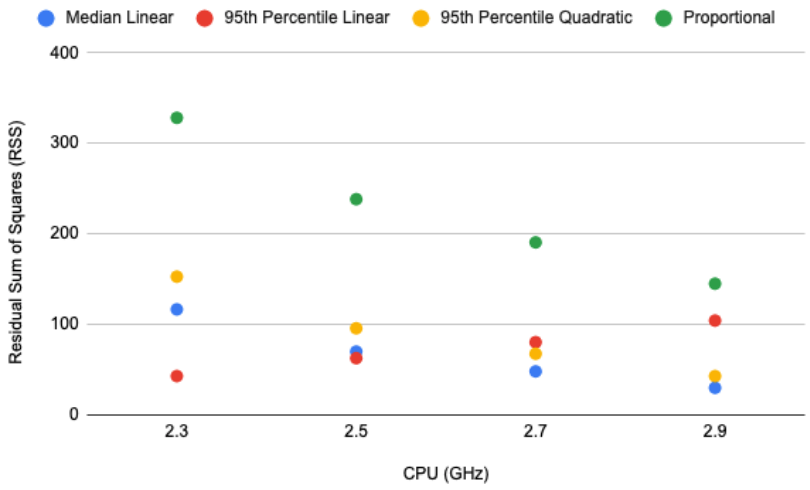


Figure 5.4: Local CPU to Time Estimator RSS values

Figure 5.4 illustrates the relationship of the RSS for each estimator function for local profiler CPU powers. Which represents the data in table 5.3 for input 100000. With relation to the data for both cross CPU power estimation and per memory estimation accuracy it is evident that the Median linear function produces respectively lower error rate. Hence the conclusion to use Median Linear Time Estimator is decided 4.12.

5.2.3 Cost Estimation analysis

With the previous section resulting with the conclusion to use equation 4.12 as the time estimator function the final cost estimation function is defined as 5.1

$$Price_{TotalCost} = T(C_1 * \frac{100.CR_{local}.T_{local}}{CR_{serverless}.CC.(0.4431 + 0.0554M)} * \frac{M}{1024} + C_2) \tag{5.1}$$

The cost estimations are generated based on this function and then compared against the averaged execution times for each round of function executions. Since the functions has a linear relationship with the number of request the cost estimation for a single call is calculated for the result discussion. The overall cost for each round of executions in Lamdba is calculated using the standard cost calculator function. Then the total sum for the three rounds of execution is collected to get the cost for 1500 function executions. For each local CPU power the resulting profiler output estimation is then estimated for 150 executions and compared with the actual cost table 5.7.

The resulting output reveals that the actual total costs are always below than the estimation with minor margins with a difference of 1 x 10^{-4} differences of Dollar differences for 150 executions. Even with a higher number of request the resulting costs are not going to carry a

Table 5.7: Cost Estimation difference for 150 executions

Local CPU	Local Memory	Local execution time	128	192	256	512	1024	1536	2048
2.9	80	16.9995	-0.00418443	-0.00213841	-0.0019473	-0.00307453	-0.00361658	-0.00770856	-0.00302271
2.7	80	18.4343	-0.00457768	-0.00253953	-0.00235246	-0.00348593	-0.00403116	-0.00812422	-0.00343891
2.5	80	20.0842	-0.00494093	-0.00291004	-0.00272672	-0.00386594	-0.00441412	-0.00850817	-0.00382336
2.3	80	22.143	-0.00553687	-0.0035179	-0.00334071	-0.00448938	-0.00504239	-0.00913807	-0.00445407

significant difference on the cost. The most significant observation from these results is that for the CPU function used for evaluation the best cost estimation which correlates to the same running time as local machine is below the 100% CPU allocation. The configuration 1024MB where proportional CPU fraction co-relates to 59% of CPU shares is where the actual invocation records the lowest total cost. Therefore, the cost predictions are reliable with the Final Cost estimation function.

5.3 Function Profiling for Serverless Cost optimization

With the results obtained and the observations made it is evident that function profiling done on a local machine can be successfully used for Serverless cost estimations. In order to achieve the particular output following steps has to be carried out as a workflow of execution.

1. Execute function locally using memory_profiler to obtain memory and running time of the function.

2. Execute the function locally with scalene for CPU profiling to identify average CPU utilization and running time of the function.

3. Using Time Estimator function calculate the optimal memory configuration for the expected running time on Serverless.

4. Estimate the cost of function execution for the particular memory configuration to obtain the ball park cost of function execution.

5. Deploy the function in the Serverless platform with the respective memory configuration.

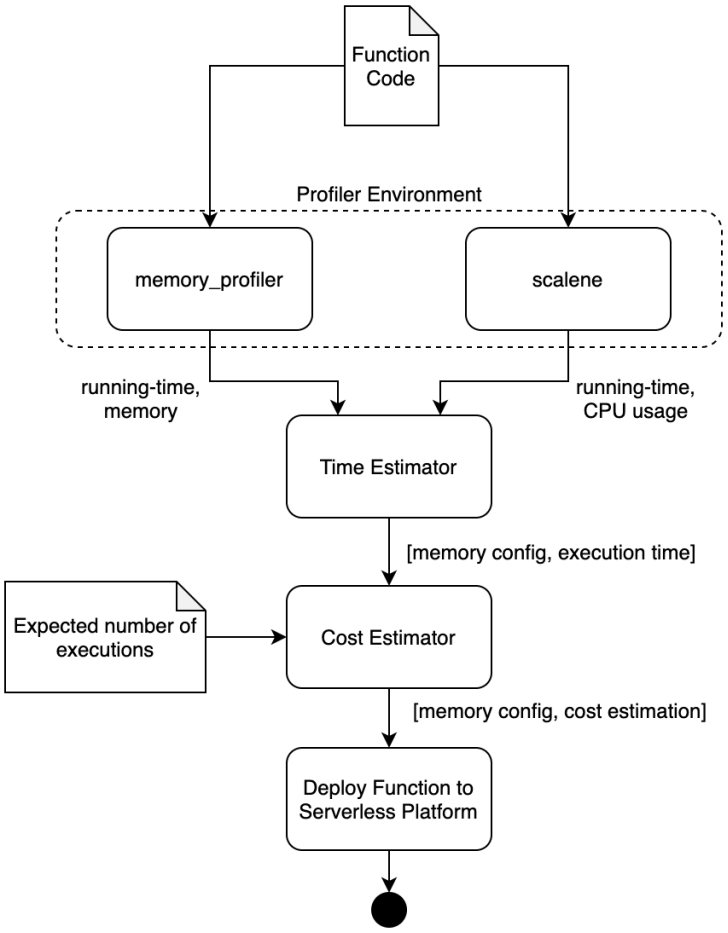


Figure 5.5: Local profiler to Serverless memory configuration workflow

Following the steps in Figure 5.5 we are able to transfer the control for the developer to successfully estimate the running time on the Serverless platform and expected cost for the particular configuration. This is a key result output which finally addresses the third research question on how to obtain cost optimizations based on the output memory parameter. Rather than relying on the straightforward proportional CPU allocation for profiler to running time estimation. The linear regression approach discussed in this research has proven that better cost optimizations can be achieved without the utilization of brute force techniques for Serverless platforms.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

At the Begining of the research the main problem to address was how the developers would be able to successfully evaluate the performance of their function before deploying them to a Serverless environment. Since the Serverless platforms are primarily allowing memory as a configuration parameter the developers were on the blue when deciding the suitable memory configuration for their functionality with CPU time taken in for consideration. The only information available with regards to CPU is that each of the platforms AWS Lambda and GCF are providing proportional CPU in relationship with memory. This resulted in developers to be executing the functions in a brute force method to understand the suitable memory configuration with the optimal running time requirement.

By using the memory_profiler for python and scalene profilers this study has successfully profiled functions for memory and CPU usage which are then converted to Serverless memory configuration. Which has addressed the first research question 1 2.8.1 What are the suitable function profiling techniques to define memory parameters for a Serverless function? With memory_profiler (mprof) as a memory profiling tool and scalene as a CPU profiler we are able to calculate memory parameter for a Serverless platform.

Rather than just providing a memory output through profiling the regression analysis on time-sharing on Serverless platforms has further allowed to convert the output CPU and Memory from local profiling to Serverless running time with the appropriate Serverless memory parameter. The regression model was evaluated against the newly updated proportional CPU allocation. The evaluation proved that the constant time-sharing provided by proportional allocation changes hence the regression model is more suitable for calculating the underlying CPU. Which in return

addressed the research question 2 2.8.2 What is the formula that is capable of converting profiler CPU and Memory parameters into the Vendor-specific Serverless function memory parameter? by introducing Time estimation function.

With the ability to obtain the running times of a function based on the memory parameter cost estimator function is finally defined and the steps to obtain the optimal cost is defined as algorithm steps. These steps can be utilized with local profiler outputs to adjust the memory parameter on the Serverless platform to obtain optimal running time while having a look at the potential cost of execution for a given number of executions. Which in return answers the research question 3 2.8.3 Can the memory parameter and execution time provided by function profiling be used to reduce costs of Serverless by defining the optimal memory configuration? with a solid yes.

Although there is error involved in all of the models and doesn't estimate either the time and or the cost exactly the given that these errors are in the range of fractions we are satisfied with the results obtained. The study has successfully created a mechanism for developers to profile their functions and provide an estimation of time and costs incurred on their functions before they are deployed into a Serverless platform. This leaves us with a conclusion to the study where function profiling can be used for function evaluation even though it is deployed into a Serverless platform.

6.2 Future Work

Within this research the Serverless platforms considered are only AWS Lambda and GCF. However, there are other Serverless platforms which needs the same evaluation to be performed on eg:- Azure functions and Openwhisk as tasks for the future understanding the CPU time sharing of these platforms has to be considered.

The primary target of this study was to produce a suitable technique to profile functions beforehand of deploying to a Serverless platform. Therefore, the evaluation focused on the capability of converting the profiler outputs CPU, memory and running time for a Serverless memory configuration. Therefore, the functions used are to be expected to use maximum memory and maximum CPU for the total period of execution. Although, it was intended to study the varying resource functions within the same study given the execution rounds of evaluation it is kept aside. Therefore, a future study would require to analyze the feasibility of the same solution for varying resource usages.

References

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *Research Advances in Cloud Computing*, pp. 1–20, 2017.

[2] Google, "Serverless - explore - google trends," 2019. [Online]. Available: https://trends.google.com/trends/explore?date=today%205y&q=Serverless

[3] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Optimizing cost of serverless computing through function fusion and placement," *Proceedings - 2018 3rd ACM/IEEE Symposium on Edge Computing, SEC 2018*, pp. 300–312, 2018.

[4] AWS, "AWS Lambda Pricing." [Online]. Available: https://aws.amazon.com/lambda/pricing/

[5] IBM, "IBM Cloud Functions - Pricing." [Online]. Available: https://cloud.ibm.com/functions/learn/pricing

[6] Google, "Pricing | Cloud Functions Documentation | Google Cloud." [Online]. Available: https://cloud.google.com/functions/pricing

[7] Microsoft, "Pricing - Functions | Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/

[8] Amazon.com Inc., "AWS Lambda Limits," 2019. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/limits.html

[9] R. Bolscher, "Leveraging serverless cloud computing architectures : developing a serverless architecture design framework based on best practices utilizing the potential benefits of serverless computing." August 2019. [Online]. Available: http://essay.utwente.nl/79476/

[10] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[11] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Serverless Computation with OpenLambda 1 Introduction 2 Lambda Background 3 Lambda Workloads," *USENIX Workshop on Hot Topics in Cloud Computing*, 2016. [Online]. Available: https://www2.cs.uic.edu/ brents/cs494-cdcs/papers/openlambda.pdf https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf

[12] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for SLO and cost aware resource procurement in public cloud," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2019-July, pp. 199–208, 2019.

[13] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[14] AWS, "AWS Lambda." [Online]. Available: https://aws.amazon.com/lambda/

[15] S. Engdahl "Firecracker – lightweight virtualization for serverless computing," 2008. [Online]. Available: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/

[16] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, pp. 405–410, 2017.

[17] Azure. [Online]. Available: https://azure.microsoft.com/en-in/services/functions/

[18] A. Lambda, "Aws lambda faqs," 2020. [Online]. Available: https://aws.amazon.com/lambda/faqs/

[19] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Cost comparison of running web

applications in the cloud using monolithic, microservice, and AWS Lambda architectures," *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 233–247, 2017.

[20] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 133–146, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[21] J. Chapin, "The Occasional Chaos of AWS Lambda Runtime Performance," 2017. [Online]. Available: https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e

[22] C. Ghoroghi and T. Alinaghi, "An introduction to profiling mechanisms and Linux profilers."

[23] Y. Cui, "AWS Lambda Language Comparison: Pros and Cons," 2018. [Online]. Available: https://epsagon.com/blog/aws-lambda-programming-language-comparison/

[24] T. N. Bui, "Benchmarking AWS Lambda runtimes in 2019," 2019. [Online]. Available: https://medium.com/the-theam-journey/benchmarking-aws-lambda-runtimes-in-2019-part-i-b1ee459a293d

[25] D. Bardsley, L. Ryan, and J. Howard, "Serverless performance and optimization strategies," *Proceedings - 3rd IEEE International Conference on Smart Cloud, SmartCloud 2018*, pp. 19–26, 2018.

Appendix A

Lambda CPU time sharing log

2020-04-18T08:51:46.959Z START RequestId: 25bb1091-0470-4927-b091-d4acdea7e45f Version: $LATEST

2020-04-18T08:51:47.208Z b'processor: 0 vendor_id: GenuineIntel cpu family: 6 model: 62 model name: Intel(R) Xeon(R) Processor @ 2.50GHz stepping: 4 microcode: 0x1 cpu MHz: 2500.010 cache size: 33792 KB physical id: 0 siblings: 2 core id: 0 cpu cores: 2 apicid: 0 initial apicid: 0 fpu: yes fpu_exception: yes cpuid level: 13 wp: yes flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm cpuid_fault pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust smep erms smap xsaveopt arat md_clear arch_capabilities bugs: cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs bogomips: 5000.02 clflush size: 64 cache_alignment: 64 address sizes: 46 bits physical, 48 bits virtual power management: processor: 1 vendor_id: GenuineIntel cpu family: 6 model: 62 model name: Intel(R) Xeon(R) Processor @ 2.50GHz stepping: 4 microcode: 0x1 cpu MHz: 2500.010 cache size: 33792 KB physical id: 0 siblings: 2 core id: 1 cpu cores: 2 apicid: 1 initial apicid: 1 fpu: yes fpu_exception: yes cpuid level: 13 wp: yes flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm cpuid_fault pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust smep erms smap xsaveopt arat md_clear arch_capabilities bugs: cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds

swapgs bogomips: 5000.02 clflush size: 64 cache_alignment: 64 address sizes: 46 bits physical, 48 bits virtual power management: '

2020-04-18T08:51:48.429Z
Process_Result:startCPU,9,endCPU,17,startTime,1595.76,endTime,1596.94,Hertz,100

2020-04-18T08:51:48.430Z END RequestId: 25bb1091-0470-4927-b091-d4acdea7e45f

2020-04-18T08:51:48.430Z REPORT RequestId: 25bb1091-0470-4927-b091-d4acdea7e45f Duration: 1471.48 ms Billed Duration: 1500 ms Memory Size: 128 MB Max Memory Used: 52 MB Init Duration: 119.92 ms

Appendix B

Profiler memprof result - sample

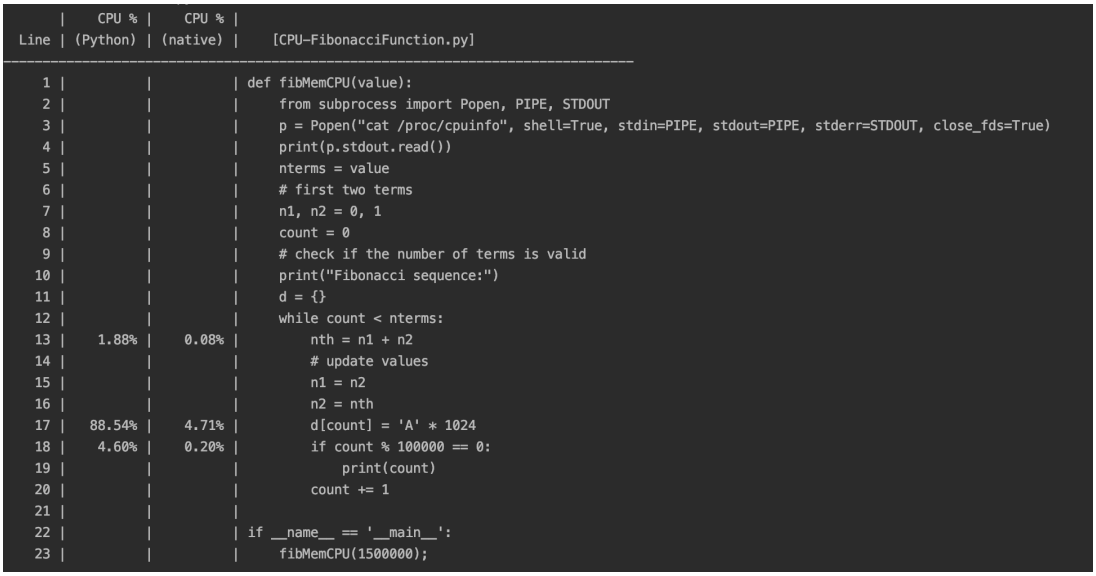
CMDLINE /usr/local/opt/python@2/bin/python2.7 MemoryFunction.py

MEM 0.179688 1576746200.7199 MEM 9.808594 1576746200.8203 MEM 95.640625 1576746200.9205 MEM 238.847656 1576746201.0209 MEM 347.882812 1576746201.1239 MEM 439.031250 1576746201.2242 MEM 531.613281 1576746201.3278 MEM 635.953125 1576746201.4290 MEM 729.585938 1576746201.5303 MEM 833.160156 1576746201.6312 MEM 938.054688 1576746201.7318 MEM 1043.734375 1576746201.8322 MEM 1152.546875 1576746201.9327 MEM 1252.378906 1576746202.0331 MEM 1377.351562 1576746202.1335 MEM 1502.718750 1576746202.2354 MEM 1539.710938 1576746202.3358 MEM 1544.863281 1576746202.4375 MEM 1577.675781 1576746202.5398 MEM 1625.628906 1576746202.6408 MEM 1717.414062 1576746202.7412 MEM 1782.792969 1576746202.8417 MEM 1835.824219 1576746202.9420 MEM 1883.707031 1576746203.0423 MEM 1902.468750 1576746203.1452 MEM 1942.542969 1576746203.2454 MEM 2028.250000 1576746203.3459 MEM 2107.429688 1576746203.4463 MEM 2196.339844 1576746203.5468 MEM 2266.648438 1576746203.6501 MEM 2350.207031 1576746203.7504 MEM 2395.203125 1576746203.8542 MEM 2408.667969 1576746203.9547 MEM 2535.148438 1576746204.0566 MEM 2643.027344 1576746204.1571 MEM 2720.632812 1576746204.2574 MEM 2819.976562 1576746204.3626 MEM 2862.839844 1576746204.4629 MEM 2936.273438 1576746204.5637 MEM 3021.621094 1576746204.6644 MEM 3101.578125 1576746204.7649 MEM 3152.273438 1576746204.8653 MEM 3201.859375 1576746204.9656 MEM 3238.980469 1576746205.0662

Appendix C

Profiler scalene result - sample

CPU-FibonacciFunction.py: % of CPU time = 100.00% out of 26.19s.



```
      |    CPU % |    CPU % |
 Line | (Python) | (native) |    [CPU-FibonacciFunction.py]
--------------------------------------------------------------------------------
    1 |          |          | def fibMemCPU(value):
    2 |          |          |     from subprocess import Popen, PIPE, STDOUT
    3 |          |          |     p = Popen("cat /proc/cpuinfo", shell=True, stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
    4 |          |          |     print(p.stdout.read())
    5 |          |          |     nterms = value
    6 |          |          |     # first two terms
    7 |          |          |     n1, n2 = 0, 1
    8 |          |          |     count = 0
    9 |          |          |     # check if the number of terms is valid
   10 |          |          |     print("Fibonacci sequence:")
   11 |          |          |     d = {}
   12 |          |          |     while count < nterms:
   13 |    1.88% |    0.08% |         nth = n1 + n2
   14 |          |          |         # update values
   15 |          |          |         n1 = n2
   16 |          |          |         n2 = nth
   17 |   88.54% |    4.71% |         d[count] = 'A' * 1024
   18 |    4.60% |    0.20% |         if count % 100000 == 0:
   19 |          |          |             print(count)
   20 |          |          |         count += 1
   21 |          |          |
   22 |          |          | if __name__ == '__main__':
   23 |          |          |     fibMemCPU(1500000);
```

Figure C.1: Scalene Sample output