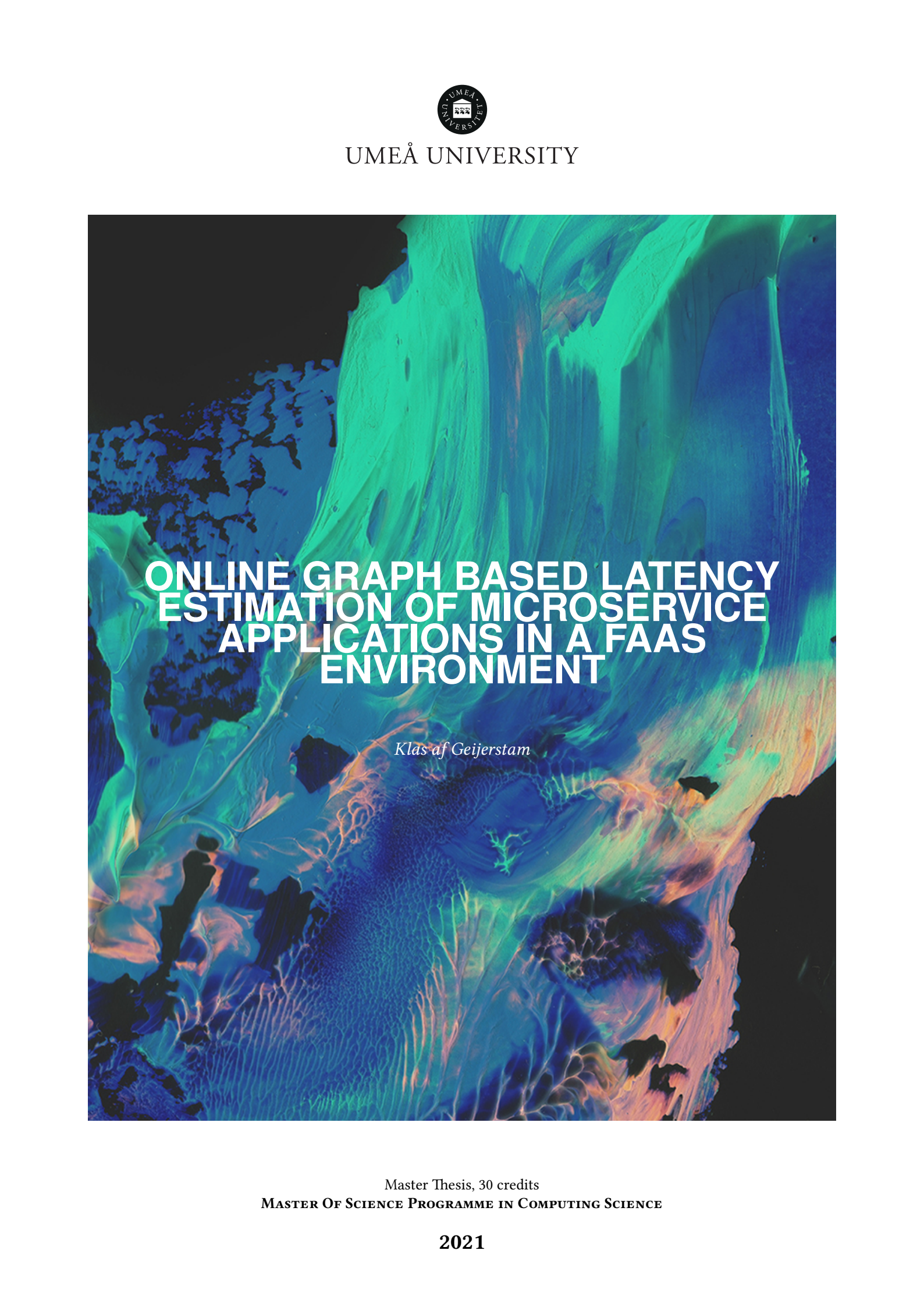

Master Thesis, 30 credits

Master Of Science Programme in Computing Science

2021

Abstract

Function-as-a-Service (FaaS) is an increasingly common platform for many kinds of applications and services, replacing the need to maintain and setup hardware or virtual machines to host functionality in the cloud. The billing model for FaaS is commonly based on actual usage, which makes the ability to estimate the performance and latency of an application before invoking it valuable. This thesis evaluates if previously defined algorithms for offline latency estimation, can be adapted to work with online data. Performing online estimation of latency potentially enables cheaper estimations, as no extra executions are neccessary, and latency estimation of applications and functions that can not be executed spuriously. The experiments show that for a set of test applications, the previously defined algorithms can achieve greater than 95% accuracy, and that a non-graph based estimation using exponential moving average can achieve greater than 98% accuracy.

Acknowledgements

I want to thank Jakub for being a great friend and an invaluable collaborator on the theoretical parts of this thesis. I also want to thank Desireé for her never ending patience with me, and for all the support she has given me. Finally, I want to thank Jerry Eriksson for being a constant force for good during my time at Umeå University.

Contents

1 Introduction

Function-as-a-Service or FaaS is a relatively new technology and platform that introduces a novel model to develop and create everything from event-triggered tasks to complex microservice applications. The technology allows the creation of horizontally scalable data pipelines with little to no effort in setting up an execution environment and other resources.

The billing model for public FaaS providers such as Amazon AWS and Google Cloud Functions are primarily based on usage, compared to traditional service rental which is typically a subscription model with a time based rate. This makes the ability to estimate the runtime performance and latency of functions and applications a useful tool in determining the cost of running an application in a FaaS environment.

1.1 Background

Previous work has defined algorithms to estimate cost, execution time and resource usage of microservice applications running in a FaaS environment with a high accuracy. The algorithms primarily rely on pre-recorded data of individual functions of an application as a base to calculate the estimated metrics for the application as a whole. Latency estimation can, and has, played an important role in the scheduling of FaaS functions and applications, primarily with a focus on minimizing cost and execution time (latency). Latency estimation has also played an important role in the definition of Service-Level-Agreements (SLA) for FaaS, where there is a need to be able to estimate execution time beforehand to meet the set SLA.

1.2 Purpose

Pre-recording data can be both time-consuming and costly, as the application and its functions have to be executed repeatedly to gather data, and the user is typically billed for computation/execution time. Spuriously executing an application or function to gather performance metrics can also be very difficult due to functions requiring input that cannot easily be predicted or generated, or the fact that functions and applications can rely on, and update, an external state.

1.3 Research Questions

This study aims to answer three research questions. How is the accuracy of latency estimation of the previously defined algorithm by Lin et al.[5] when fed with online data, compared to the accuracies originally achieved with offline data? How does the achieved accuracy compare to an estimation based on pure Exponential-Moving-Average (EMA)? After how many executions can a dynamically generated graph model be accurate enough to allow for a 95% accuracy in latency estimation?

The hypothesis is that pure-EMA should perform better than the graph model, as the graph model is still dependent on EMA for model-generation, and the added complexity of the graph model is not expected to result in better accuracy.

1.4 Method used

The algorithm by Lin et al. was adapted to work with online data. OpenWhisk was used to create a FaaS environment in which FaaS-applications could be executed and instrumented, and the applications defined in previous work were re-implemented to work with OpenWhisk.

1.5 Delimitations

This study uses applications previously defined by Lin et al.[5] to get comparable results. These applications consist of artificially created workloads of different kinds, like memory or network intensive operations. The functions of the applications do not transfer data between invocations in the form of parameters, a simplification made for the sake of testing, but it reduces how well the applications can model real-world applications and workloads.

2 Related work

Previous studies have looked at estimating cost and latency of microservice applications created in a FaaS environment, typically with a focus on one specific type of application. K. Ram Srivatsa et al.[4] looked at measuring runtime performance of machine learning applications, and using these metrics to improve scheduling and resource utilization in a FaaS environment. Ram Srivatsa et al. found that homogeneity in different applications could be used to improve the throughput of a FaaS system, as functions with similar workloads could be scheduled on nodes optimal for their task.

Spillner et al.[9] evaluated how well a FaaS platform could be utilized for computationally intensive tasks. Many of the computationally intensive tasks, such as calculating digits of π or numbers of the fibonacci sequence that were part of the tests performed by Spillner et al. are present in the test applications of this study.

Cordingly et al.[1] investigated how performance and cost of FaaS applications could be predicted with purpose-built machine learning algorithms. The models were trained on performance metrics gathered from applications running in AWS Lambda and Google Cloud functions. Cordinly et al. found that the accuracy of their model differed depending on many aspects, such as memory allocation and workload. The mean maximum error achieved was a 3.49% error in performance and cost estimation.

In "Modeling and Optimization of Performance and Cost of Serverless Applications"[5], C.Lin and H.Khazaei introduced a series of algorithms that procecesses a graph representation of a microservice into a form that simplifies estimation of the execution time (latency) of microservice applications. The graph model represent functions or services as nodes, and functions invoking other functions as edges. The weight of the edge is a number between 0 and 1, which is the probability that a function calls the target function of the edge. The application graph is allowed to contain branches, parallel sections and cycles/self-cycles. The algorithms proposed by Lin et al. process such a probability-graph into a directed acyclic graph with a single simple path between the start node and the end node, from which the estimated end-to-end latency of invoking the application can be calculated. The algorithm created by Lin et al. achieved a 98% accuracy in estimating the performance of an application.

The graph-model used by the Lin et al. is pre-generated by augmenting a perfect representation of the application with function latencies recorded by executing each function of the application multiple times. This gives the algorithm a near perfect model of the application, as well as a very good estimate of the expected latency of each function in the application.

3 Theoretical background

The following section describes and explains key concepts important to this thesis, and the related work.

3.1 Function as a service (FaaS)

Traditional services of cloud providers include server rental, VM-rental and managed services, such as database and webserver hosting. During the recent years a new service and architecture has emerged, namely Funcion-as-a-Service (FaaS). When developing an application to run in a FaaS environment, the user creates functions conforming to a runtime and format specific to the used FaaS platform[9]. The functions are most commonly invoked by triggers or events, and can be chained to create more complex applications.

One of the main advantages of FaaS over traditional services is that the application becomes inherently scalable, as the provider can always deploy the function to another container or virtual machine when needed. The billing model is primarily based on the execution time and memory allocation of the funcion, compared to that of traditional services like VM-rental where the user is charged at a fixed rate during the subscribed time.

3.2 Apache OpenWhisk

OpenWhisk is an open-source FaaS platform that can be deployed on self-hosted infrastructure[6], and it is also the main driver for IBM cloud functions[3]. As OpenWhisk can be deployed on owned infrastructure it is an ideal environment for experiments and modifications, allowing control over hardware and resource allocation much more granulary than what is possible with public FaaS providers such as AWS Lambda or Google Cloud Functions. It has successfully been used in previous work, both in performance evaluations[8] and as a modifiable FaaS platform[10].

OpenWhisk utilizes Docker to create isolated execution environments, called invokers, to execute functions. The invokers are specialized containers initialized with an environment specific for the function that is to be invoked, often paired with a language-specific file that lists dependencies of the functions, like a requirements file for Python or a TOML-file for a Rust function.

3.3 Containers

A container is a lightweight isolated execution environment not unlike a virtual machine. A container runtime does not have to virtualize the entire machine, as it is instead defined by using isolation support from the host kernel. This results in much better performance[7]

compared to pure virtual machines, as the host machine does not have to virtualize hardware, at a cost of reduced isolation. Most if not all FaaS frameworks build upon containerization, and in the end virtualization[9].

A common platform for containerization is Docker[2] which also is the host environment used for invokers in OpenWhisk. Docker provides a standardised environment to build, create and maintain containers and is the de-facto standard for containerization today.

3.4 Microservices

A microservice architecture is an application architecture where different parts of an application or system, that would traditionally be kept in a single service, is split into many smaller logical services. This allows for better scalability as each service can be scaled horizontally and replaced with little theoretical impact on the other services of the application. A microservice architecture also simplifies development as the services can be developed in parallel, and existing services can be replaced by new services implementing the same API as an existing service.

3.5 Probabilistic graph representation of applications

An application can be represented as a graph, where the nodes of the graph represent the functions of the application, and edges represent a function calling another. An application can contain loops, branches and parallel sections, which can very naturally be represented by edges in a directed graph.

Weighting the edges of the graph with a probability allows the graph to represent applications with branches and loops, where branches are formed with edges with a probability of less than 1.0, and loops as cycles. The sum of the outgoing edges for a node with a branch should have a summed probability of 1, and nodes that start a parallel section a summed edge weight greater than 1. An example of an application containing a branch and a cycle can be seen in Figure 1, where the cycle consist of nodes $f1$, $f2$, $f4$ and node $f4$ contains a branch between end_point and $f1$.

3.6 Graph transformation

Lin et al. developed a graph-transformation algorithm to transform a probabilistic graph representation of an application into a single-path non-probabilistic graph. The algorithm removes cycles, self-cycles, branches and cycles, subseqently transforming the graph into the aforementioned single-path format, examples of which can be seen in Figure 2 . The end to end latency of the application can easily be extracted from the single-path graph by summing the latency of the path between the start node and the end node in the graph.

Self-cycles

The algorithm removes self-cycles (somtimes called loops) by removing edges that introduce self-cycles in two steps. The first step increases the estimated latency of the function containing the self-cycle with the estimated amount of iterations of the self-cycle (given by Equa-

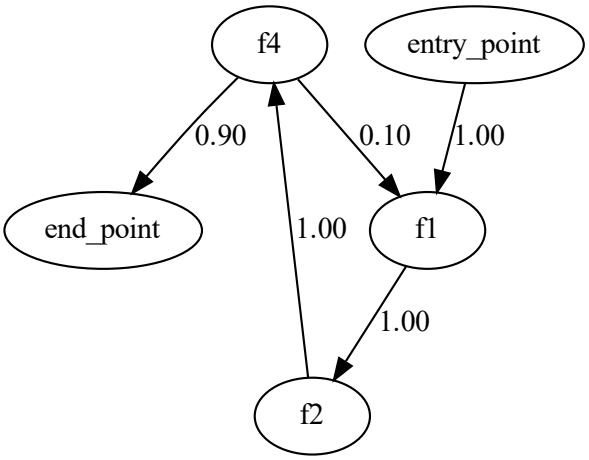

Figure 1: Graph representing an application with 3 steps. The edges are weighted by the probability of the function invoking another function. The edge between f4 and f1 indicates a 10% probability of f4 invoking f1.

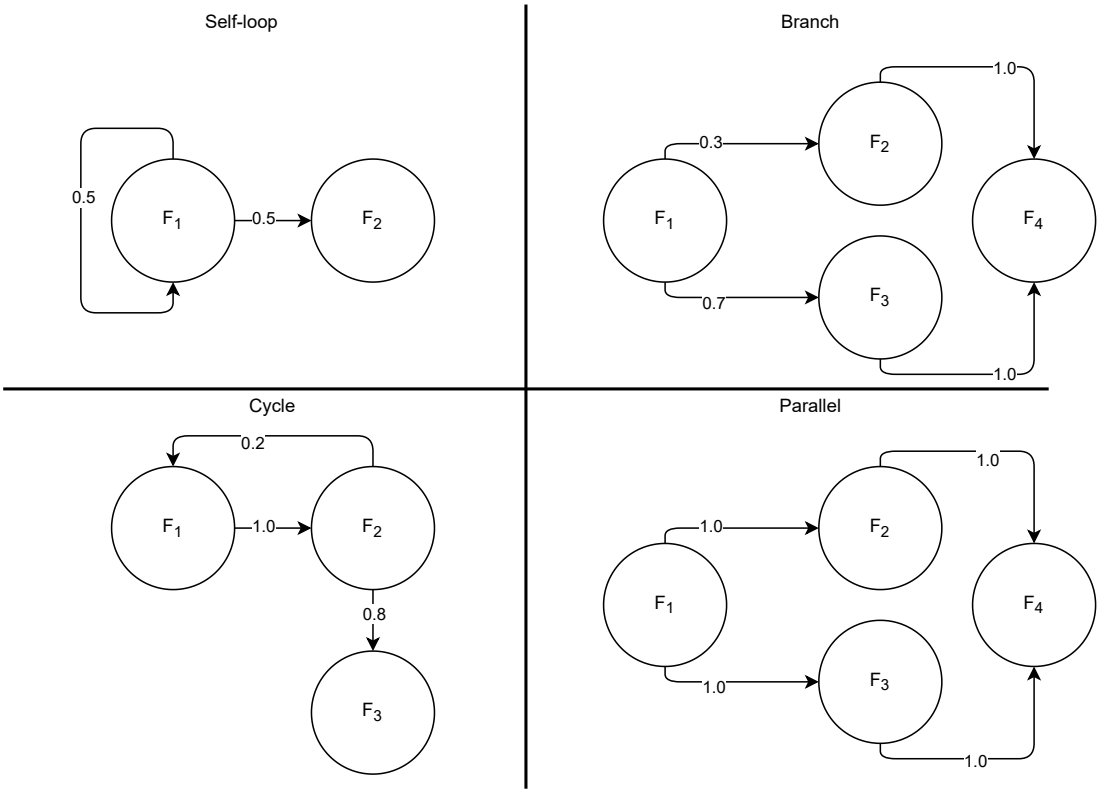

Figure 2: The four structures that are removed by the graph processing algorithm.

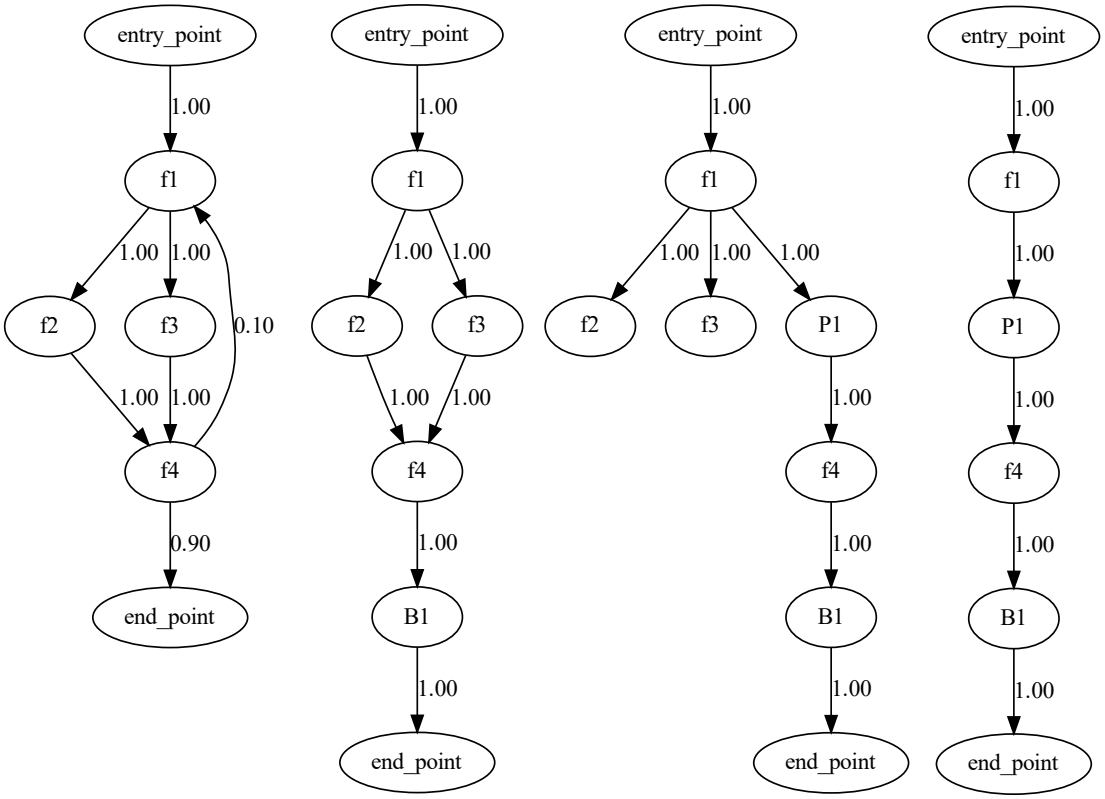

Figure 3: Transforming a graph. The transformation is based on the leftmost graph. In the first step the branch is removed, in the second the parallel section, and lastly any un-connected nodes are removed.

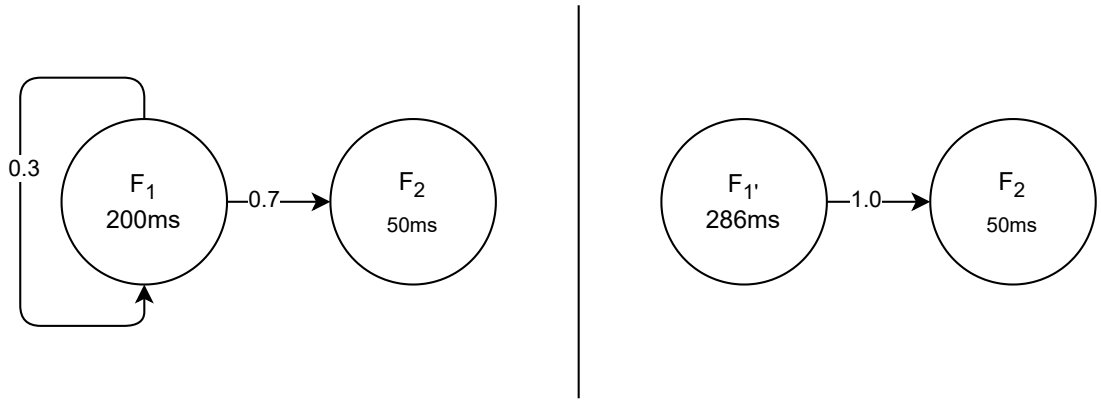

Figure 4: Example of removing a self-cycle/loop. The estimated cycle-count of the self cycle is calculated using Equation 1. The latency of F_1 is increased by $Latency(F_1) * Cycles$. The edge that introduces the self-cycle is removed.

tion 1), multiplied with the latency of the function. The second step distributes the probability of the edge that introduces the self-cycle over any other outgoing edges from the node according to Equation 2.

In Equation 1, $P(u, v)$ is the probability of a transition from function u to function v. The weight of the edge that introduces the self-cycle is distributed according to Equation 2. The edge is subseqently removed. An example of this operation can be seen in Figure 4, where the self-cycle from $f1$ to $f1$ is removed.

$$\frac{P(u,v)}{1 - P(u,v)} \tag{1}$$

Cycles

Any cycles in the application graph are removed by calculating the estimated cycle-count of the cycle (by Equation 1), and then multiplying with the total latency of the cycle. This latency is added to the latency of the source node of the cycle, and the edge that introduces the cycle is removed. The probability of the removed edge is distributed over the remaining edges proportionally to the probability of the edge by using the formula in Equation 2, where $P(u, v)$ is the probability of a transition from function u to function v, and R is the probability of the removed edge. An example of a removal of a cycle, and the redistribution of the edge probabilities can be seen in Figure 5, where the edge that introduces a cycle is removed ($f2$,$f1$) and the latency of $f1$ is updated with the total latency of the cycle ($200ms + 150ms$).

$$P'(u,v) = \frac{P(u,v)}{1 - R} \tag{2}$$

Branches

Branches in the application graph are removed in three steps. The first step calculates the estimated latency for each branch path. The second step creates a new “B” node that stores

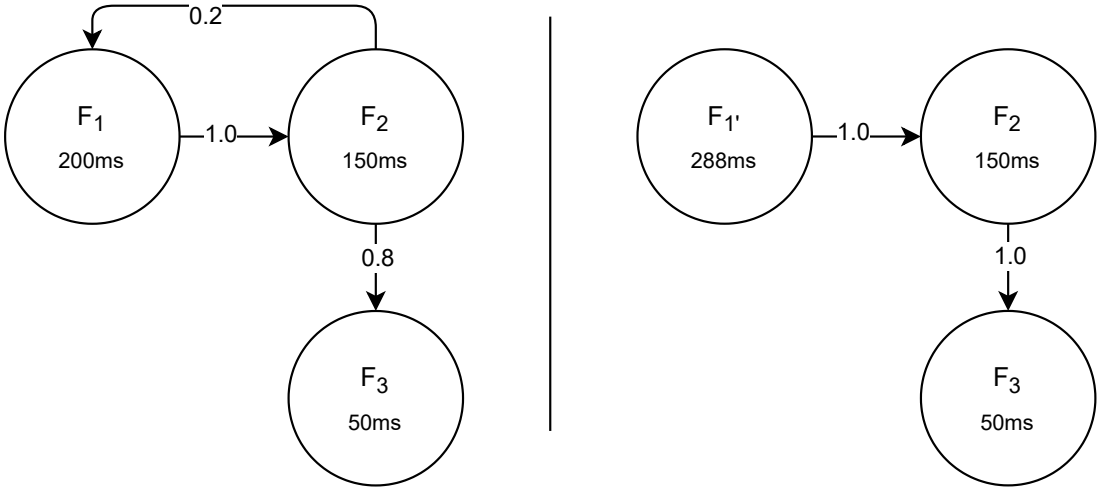

Figure 5: Removal of a cycle. The cost of the cycle ($200ms + 150ms$) is multiplied with the estimated cycle count. The latency of $F1$ is thus adjusted according to Equation 1 and the cycle latency of 350ms.

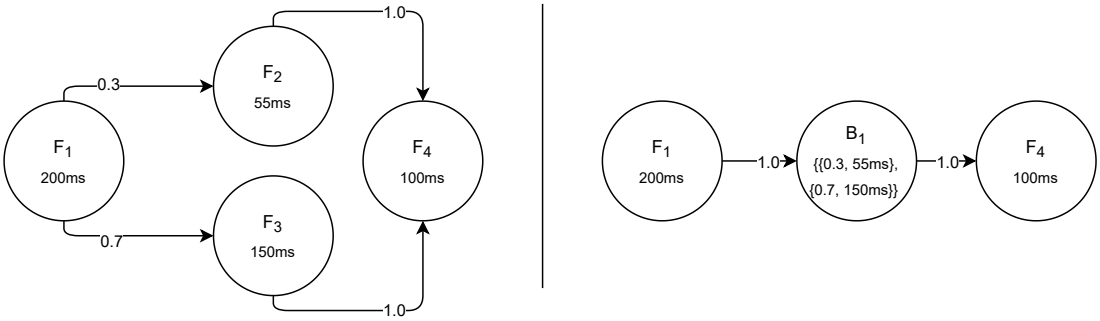

Figure 6: Figure shows the removal of a branch structure and how a B node is introduced to represent the branches.

the latency and probability of each path, to which the source node of the branch is then connected. The last step removes the edges that creates the branch. The difference between a B node and a normal node is that a B node stores multiple latencies, one for each branch path, whereas a normal node only stores one. The latencies are stored as tuples, the latency and the probability.

The latency of a B node is calculated as the sum of each path multiplied with the probability of the path. A B node storing the paths $(100ms, 0.3), (50ms, 0.7)$ would have the estimated latency of $100 * 0.3 + 50 * 0.7 = 65ms$.

Parallel structures

The algorithm processes parallel structures in two steps. In the first step, the algorithm finds the parallel path with the highest latency. As parallel sections, by definition, are executed simultaneously, only the latency-wise longest section affect the end to end latency of the application. The second step introduces a new "P" node, of which latency is set to the total latency of the parallel path with the highest total latency. The parallel paths are then removed, and the newly created P node is connected to the origin node of the parallel section, and the

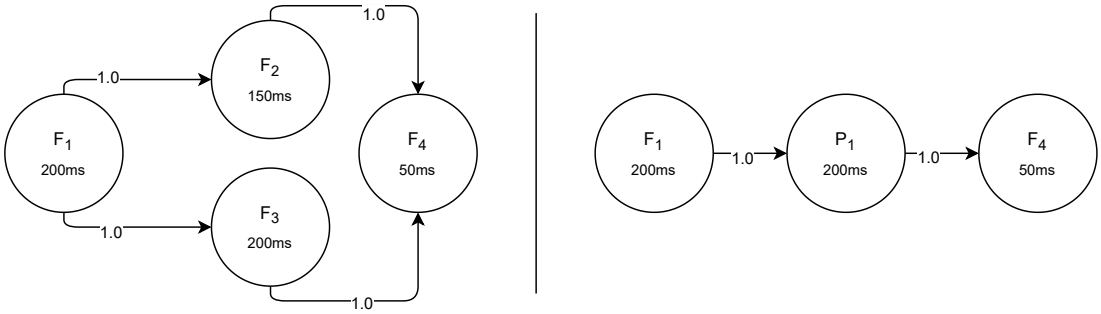

Figure 7: The figure shows how a parallel section is removed. The parallel section is replaced with a P node, with the weight (latency) of the total latency of the longest parallel path.

end node of the parallel section, an example of this operation can be seen in Figure 7.

The purpose of changing the node to a P node is purely to visualize that it represents a parallel section in the processed graph, as it stores latency just as a normal node does.

3.7 Exponential moving average

Exponential moving average is a weighed average, with exponentially reduced weighting of older values. In its pure form it never removes a value, albeit infinitively old values have an infinitively low weight. Since the exponential moving average is performed in epochs for this study, there is however no such values present, as only values up to the epoch-size are considered.

4 Method

The method was designed with comparability to the work of Lin et al. in mind, to ensure that the research questions could be answered with as much confidence as possible. Two experiments were devised to answer the three research questions. One experiment was designed to answer two of the research questions both relating to accuracy in latency estimation, and one experiment to answer the representational accuracy of the model used for the graph-based estimations.

4.1 Test environment

Functions were defined as OpenWhisk Python functions, and deployed as such. The applications were created by defining relationships between functions in a JSON format specifically created for the experiments, as this functionality was missing from OpenWhisk. To facilitate the creation of applications in OpenWhisk, an application management function was created that allows OpenWhisk functions to invoke other OpenWhisk functions via the OpenWhisk REST-ful endpoints. Launching an application thus consist of uploading the functions of the application to OpenWhisk, and then submitting the JSON file specifying the application to the application management function.

The graph processing algorithm were reimplemented from the original `Python` implementations created by Lin et al. in `Rust`. A REST service, from here on called "logger", was created with the `Actix-web` framework to allow interaction between external services, like OpenWhisk and the graph-based estimation algorithm. OpenWhisk was modified to send latency metrics to the logger via the REST-ful API of the logger. The graph-based estimation algorithm was not implemented in OpenWhisk as to introduce as little change as possible in OpenWhisk to limit the possibility of the estimation algorithms to affect the running and scheduling of invokers.

The logger builds a graph representation of the nodes dynamically as calls and latencies are submitted to the service by OpenWhisk. Graph-node latencies are created by running EMA on all recorded latencies for each node and edges and their respective probabilities derived from the number of times a function is invoked, and how many times it invokes another function.

4.2 Choice of Method

The graph processing algorithm was largely based on the previous work of Lin et al., but modified to work in an online setting. One modification were performed, related to robustness, as online generation of a graph model does not guarantee perfect data. The issue with dynamically generating a graph model is that the probabilities of edges part of branches and cycles can be less or greater than 1.0, which the algorithm can not handle as the experiments performed by Lin et al. works on a complete model with perfect probabilities. To alleviate this a smoothing function was introduced that identifies sections of the graph with imperfect probabilities, and smoothes these to give a sum of 1.0, thus allowing the algorithm to properly process the graph.

Lin et al. used AWS Lambda as the platform for executing functions and applications. The modifications needed for the experiments in this study was deemed difficult to implement in a satisfactory way in AWS. This resulted in OpenWhisk being selected as the FaaS platform, as it satisfied the requirements and had been successfully used in previous work[10].

EMA was chosen as the averaging method as it reduces the time extreme values affect the estimated latency of a function. EMA should also perform better in a scenario where the usage pattern of a function or application changes over time, as the more recent records are given an exponentially greater weight.

4.3 Data Collection

Initial tests showed that the total execution time of an application, from start to finish, consisted of a substantial amount of waiting due to scheduling in OpenWhisk. Therefore, only the pure execution times of functions were recorded, to minimize the risk of any inefficiencies in scheduling in OpenWhisk to impact the result. This was possible due to the usage of the REST service, which allowed a custom instrumentation of the functions in OpenWhisk.

4.4 Experiments

The experiments were designed to give results that could answer the three research questions. One experiment was designed to allow both estimation methods to be compared in terms of accuracy, over a multitude of different alpha-values for EMA. The second experiment was created to visualize how the graph representation of the system changes over time, to give insight into how the graph model changes over time.

Experiment 1: Graph representation accuracy

Experiment 1 uses a subset of the applications used by Lin et al. The purpose of the experiment is to determine how the graph representation of the application changes over time. As the graph based estimation of the runtime of an application is entirely dependent on the graph model used, it should be of vital importance that the model used for estimation is accurate. The application was run 100 times, and plots of the graph representation extracted and generated every 10 runs.

Experiment 2: Accuracy of latency estimation

Experiment 2 uses the same application as Experiment 1. The purpose of the experiment is to compare the accuracy of global EMA and the graph based latency estimation with the actual latency. An epoch length of ten is used for both the EMA estimation, as well as for the graph based estimation. Both the graph estimation as well as the EMA estimation was run using multiple alpha-values as to ensure that a sub-optimal alpha-value does not impact the accuracy of either approaches. The result of the experiment should give insight into how the accuracy of the graph-based estimation is affected by using online data instead of offline data, and how the accuracy of the graph-based estimation compares to the pure EMA estimation.

5 Results

The results from the two experiments were collected and then aggregated into three figures. The result of the first experiment showcases how the perception of the running application is changed over time in the logger. The result of the second experiment is collected and presented to make the accuracy comparison of the tested approaches easily comparable.

5.1 Experiment 1

The result of experiment 1 shows that the graph representation of the application is fairly accurate after only 10 runs. The number of invocations of a function was recorded alongside the latencies and probabilities of edges, from which a graphical snapshots was generated, as can be seen in Figure 8. The latency accuracies derived from the graph models are included in Experiment 2.

Multiple applications were tested in the same way with only minor differences in representational accuracy over the recorded runs. For brevity, only one plot is included in the result.

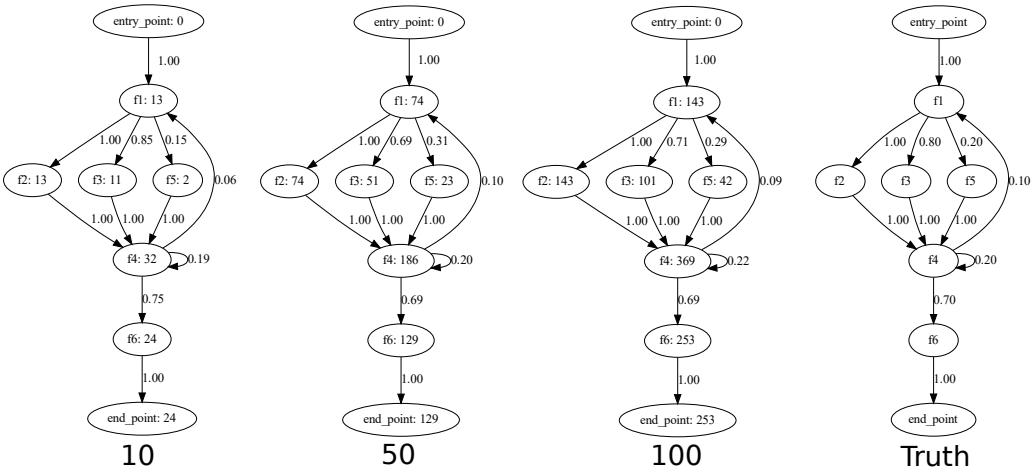

Figure 8: The graph representation of the application after 10, 50 and 100 runs. The rightmost graph is the actual representation of the graph. Each node has a number representing the number of invocations that function has seen.

5.2 Experiment 2

The result of experiment 2 was derived into two plots. One plot compares the accuracy between the pure EMA and the graph based latency estimation, as can be seen in Figure 9. The result shows that both the EMA and the graph based approach produces latency predictions relatively close to that of the actual latency, with a slightly higher accuracy achieved by EMA. The highest accuracy was achieved by pure-EMA with an alpha-value of 0.3, which achieved a 1.1% mean error. Figure 10 shows the mean error over all runs for both the EMA estimation and the graph based estimation.

6 Discussion

6.1 Interpretation of Results

The results show that, for the tested applications, the algorithm defined by Lin et al. can achieve a good accuracy for latency estimation and prediction with online data, where the best overall result giving a mean error of less than 3%. This result is slightly worse than that of Lin et al., which achieved a mean error of 2%. The pure-EMA estimation produced slightly better predictions, with a mean error less than 2% for all tested alpha values. The result show that the alpha value of the EMA algorithm can have a noticeable impact on the accuracy of both the pure EMA estimation, and the graph based estimation. Generally, the result show that a greater alpha value increases the responsiveness of the estimation, but does not result in a better mean accuracy.

The result also show that an application only has to be invoked a few times for the system to get a semi-accurate representation of the application, even if the application representation might be boosted by the presence of cycles and parallel sections in the application. For pure-EMA, a 95% accuracy could be achieved after 10 executions, depending on the application under test. The graph based approach achieved a steady > 95% accuracy for most applications

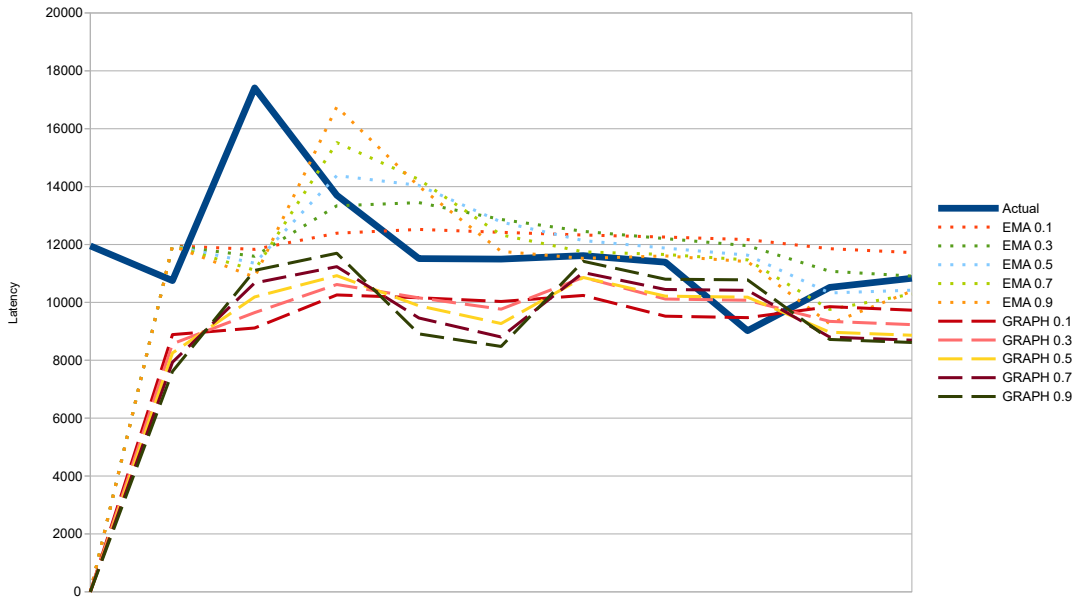

Figure 9: Estimation with EMA and graph compared to actual latency. The Y axis is the latency in miliseconds and the X axis is time. Estimation starts at 0 as there is no initial data.

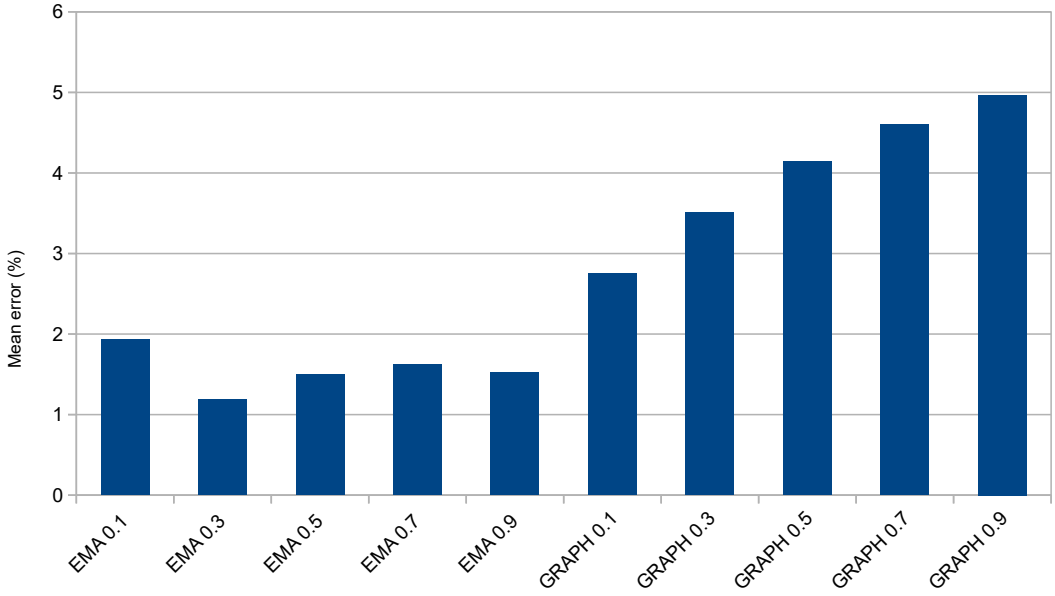

Figure 10: Mean error (in percent) of global EMA and graph based estimation of latency.

in the same amount of runs, but only on lower alpha values.

6.2 Scientific Explanations

The result of experiment 2 visualizes the accuracy between the two estimation approaches, over a set of different alpha-values. Noteworthy of the data shown in Figure 10 is that for the visualized test, there was a clear decay in accuracy as the alpha-value was increased for the graph-based estimation, but with no such pattern visible for the pure-EMA estimation. This could have its explanation in that the graph-based estimation consistently underestimated the latency of the application, and a lower alpha-value allowed the initial spike in latency to reduce the impact of the underestimation. The reason why there is no apparent optimal alpha-value could be due to the length of the tests, where the relatively short tests in the test environment might not be long enough to achieve any visible trends.

Additional tests showed that for different applications, and with different distributions of latencies, there was no clear best alpha value. For some applications, the mean error was reduced with a higher alpha-value, while a lower alpha-value improved the accuracy for some.

6.3 Limitations

The slightly worse accuracy of the graph estimation could come down to a suboptimal implementation of the graph algorithm, and the result indicates that the graph algorithm consistently underestimates the latency of the applications, which is an indication that there are improvements to be made. Adjusting the graph estimation with the mean error gives a much better result, and gives a result much closer to that of the pure EMA.

This study did not include an analysis of how the application structure affects the accuracy of the two estimation methods used. Applications containing edges with low probabilities would require more invocations for all nodes and edges to be recorded, and could thus be more difficult to estimate dynamically.

6.4 Conclusion and Recommendations

The purpose of the work of Lin et al. is not to estimate the latency of applications, but to use the estimations over different memory configurations to assign memory configurations that allow an application to execute within some bound, be it cost or time. In doing this, there might be a point to treat the application as a graph. For pure latency estimation or prediction the result of this study indicates that there is no real benefit in using the graph model over a plain estimation, as the plain EMA produced better results.

Another point in favour of using pure EMA for estimation could be both implementation complexity and resources required for estimation. The graph-based estimation used in this study utilizes EMA to generate the base graph, which is then processed and used for latency estimation. And, as the implementation requires one run of EMA for each node, the computational cost is higher for the graph approach even before graph processing can begin. Even if the accuracy of the pure EMA was equal to that of the graph approach, the pure EMA would thus be a more favourable choice in that it requires less resources in its estimation.

The experiments show that there was no optimal value found for the alpha-value for neither pure-EMA nor for the graph-based estimation. The structure of the application as well as the distribution of measured extreme values seemed to impact the accuracy over the tested alpha values.

The experiments showed that when using application-invocations as a metric, applications containing high-probability cycles and parallel sections got a more accurate representation during fewer runs, due to the fact that each function was invoked more times per application invocation. However, when using total execution time or cost as the total metric, such an application is more expensive to execute a fixed number of times.

7 Future work

Future work could look at if there is a connection between the structure of the application, and how well the performance of the application can be estimated with the defined algorithm. This could provide insights into how the algorithm could be improved or modified based on the application type or structure. The pure-EMA based approach should not be affected by the structure of the application, but it could be included for reference.

The functions and applications in this study does not transfer data between each other, which is common in real world applications. Adapting the algorithms to utilize knowledge about input could be investigated, as it could affect the accuracy of the algorithms and provide a new angle of optimization of the algorithms. The experiments found that the system could achieve an accurate representation of applications with many cycles and parallel sections with fever invocations than an application with fewer cycles and prallel sections. Since one motivation of using online data instead of offline data for estimation is cost, a runtime based execution cutoff could be investigated as an alternative to the fixed number of executions used in this study. This could allow for a more even cost distribution in estimating applications of different structures and complexities.

Future work could also investigate what role pure-EMA could have as a drop-in replacement for the algorithms proposed by Lin et al., with the same purpose of determining the best cost under a cost constraint or the lowest cost under a performance constraint, as the result of this study shows promising results in terms of accuracy for the pure-EMA approach.

References

[1] Robert Cordingly, Wen Shu, and Wes J Lloyd. Predicting performance and cost of serverless computing functions with saaf. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 640–649. IEEE, 2020.

[2] Docker. Docker. `https://docker.com`. Accessed: 2021-05-28.

[3] IBM. Ibm cloud functions. `https://cloud.ibm.com/docs/openwhisk`. Accessed: 2021-05-28.

[4] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[5] Changyuan Lin and Hamzeh Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632, 2020.

[6] OpenWhisk. Apache openwhisk. `https://openwhisk.apache.org`. Accessed: 2021-05-28.

[7] Amit M Potdar, DG Narayan, Shivaraj Kengond, and Mohammed Moin Mulla. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428, 2020.

[8] Sebastián Quevedo, Freddy Merchán, Rafael Rivadeneira, and Federico X Dominguez. Evaluating apache openwhisk-faas. In *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)*, pages 1–5. IEEE, 2019.

[9] Josef Spillner, Cristian Mateos, and David A Monge. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In *Latin American High Performance Computing Conference*, pages 154–168. Springer, 2017.

[10] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 19–24, New York, NY, USA, 2019. Association for Computing Machinery.

UMEÅ UNIVERSITY