COST MINIMIZATION IN FUNCTION-AS-A-SERVICE COMPUTING

A Thesis

by

Özgür Sedefoğlu

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Engineering

Özyeğin University
August 2021

COST MINIMIZATION IN FUNCTION-AS-A-SERVICE COMPUTING

Approved by:

Assoc. Prof. Hasan Sözer, Advisor
Department of Computer Science
Özyeğin University

Asst. Prof. İsmail Arı
Department of Computer Science
Özyeğin University

Assoc. Prof. Geylani Kardaş
International Computer Institute
Ege University

Date Approved: 18 August 2021

This thesis work is dedicated to my parents who has been taking care of my whole life. I am truly thankful for having such a good parents and sister in my life.

ABSTRACT

Cost in Function-as-a-Service computing is influenced by a number of factors. One of these factors is the amount of memory reserved during the deployment of serverless functions. Reservation of an excessive amount increases costs unnecessarily. On the other hand, decreasing this amount increases the function execution time, which is also a factor that contributes to cost. Moreover, insufficient memory can degrade the quality of service. In this thesis, we propose an automated approach for optimizing the amount of memory to be reserved for serverless functions. First, we measure the running time of a given function in various memory settings and derive a regression model. We define an objective function and a set of constraints based on this regression model and the configuration space. We obtain a nonlinear integer programming model, which is solved to determine the optimal memory setting for minimizing cost. We evaluate our approach with an industrial case study on the use of Amazon Web services in the context of Smart Home applications. We show that our approach is effective in accurately estimating the impact of memory settings on runtime performance and determining optimal settings leading to significant cost reductions. It is also useful in detecting functions with performance issues.

ÖZETÇE

Hizmet olarak işlev (Function-as-a-Service) mimarilerinde fiyatlandırma bir çok faktör hesaba katılarak yapılmaktadır. Bu faktörlerden birisi, bir bulut servis sağlayıcıya konuşlandırılan işlev için ayrılacak olan hafıza miktarıdır. Hafıza miktarının yüksek seçilmesi fiyatlandırmayı gereksiz arttıracaktır. Diğer yandan hafızanın düşük seçilmesi fiyatlandırmada rol oynayan diğer bir faktör olan çalışma süresini arttıracaktır. Ayrıca, yetersiz hafıza miktarı işlevin kalitesini de düşürecektir. Bu tez çalışmasında, sunucusuz işlevler için ayrılan hafıza miktarını otomatik olarak eniyileyen bir yaklaşım sunulmaktadır. İlk olarak söz konusu işlevin farklı hafıza değelerindeki performans ölçümleri kullanılarak bir regresyon modeli oluşturulmaktadır. Konfigürasyon uzayı ve regresyon modeli kullanılarak amaç fonksiyonu ve kısıtları belirlenmektedir. Bu verileri kullanarak elde edilen doğrusal olmayan programlama modelininin çözümü ise seçilmesi gereken en iyi hafıza miktarını vermektedir. Önerilen yaklaşımı değerlendirmek için Amazon Web hizmetlerinin kullanıldığı bir Akıllı Ev Sistemleri Bulut Uygulaması kullanılmıştır. Önerilen yaklaşım ile işlevlerin çalışma performansları göz önünde bulundurularak en düşük seviyede fiyatlandırma ve istenen kaliteyi sağlayacak hafıza seçimlerinin yapıldığı görülmüştür. Ayrıca, bu yaklaşımın kullanılması ile bazı işlevlere ilişkin performans problemlerinin de tespit edilebildiği ortaya çıkmıştır.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Hasan Sözer for all his guidance, help, everything that he has given me to finalise this work together in a pleasant way. I would also like to thank Dr. Geylani Kardaş and Dr. İsmail Arı for accepting to be part of the evaluation committee for this thesis. Finally, I would like to thank software developers at Vestel for sharing their code base with us and supporting our case study.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ABSTRACT . . . . . **iv**

ÖZETÇE . . . . . **v**

ACKNOWLEDGEMENTS . . . . . **vi**

LIST OF TABLES . . . . . **viii**

LIST OF FIGURES . . . . . **ix**

I INTRODUCTION **1**

II AWS LAMBDA FUNCTIONS **4**

III RELATED WORK **8**

IV THE TOOL AND THE OVERALL APPROACH **10**

V INDUSTRIAL CASE STUDY **14**

5.1 Experimental Setup 15

5.2 Results and Discussion 17

5.3 Threats to Validity 22

VI CONCLUSION **23**

APPENDIX A — RESULTS OF REGRESSION ANALYSIS FOR ALL FUNCTIONS OF THE SMART HOME CASE STUDY . . **24**

APPENDIX B — USE OF GEKKO FOR OPTIMIZATION **40**

APPENDIX C — FACT USER MANUAL **42**

REFERENCES **46**

VITA **49**

LIST OF TABLES

1 AWS Lambda functions that are used as experimental objects in the case study. 16

2 Parameter values used in the case study. 17

3 Results regarding the accuracy of estimation. 18

4 Results regarding the amount of cost reduction. 20

LIST OF FIGURES

1 The overall approach and components of FACT. 10

2 Result of the regression analysis for one of the functions (F1) used in the case study (See Table 1). 11

3 An overview of the use of AWS Lambda functions in the context of Smart Home applications. 14

4 Runtime performance measurements for functions F1 - F15. 19

5 Runtime performance measurements for functions F16 - F30. 19

6 Result of the regression analysis for F1. 24

7 Result of the regression analysis for F2. 25

8 Result of the regression analysis for F3. 25

9 Result of the regression analysis for F4. 26

10 Result of the regression analysis for F5. 26

11 Result of the regression analysis for F6. 27

12 Result of the regression analysis for F7. 27

13 Result of the regression analysis for F8. 28

14 Result of the regression analysis for F9. 28

15 Result of the regression analysis for F10. 29

16 Result of the regression analysis for F11. 29

17 Result of the regression analysis for F12. 30

18 Result of the regression analysis for F13. 30

19 Result of the regression analysis for F14. 31

20 Result of the regression analysis for F15. 31

21 Result of the regression analysis for F16. 32

22 Result of the regression analysis for F17. 32

23 Result of the regression analysis for F18. 33

24 Result of the regression analysis for F19. 33

25 Result of the regression analysis for F20. 34

26 Result of the regression analysis for F21. 34
27 Result of the regression analysis for F22. 35
28 Result of the regression analysis for F23. 35
29 Result of the regression analysis for F24. 36
30 Result of the regression analysis for F25. 36
31 Result of the regression analysis for F26. 37
32 Result of the regression analysis for F27. 37
33 Result of the regression analysis for F28. 38
34 Result of the regression analysis for F29. 38
35 Result of the regression analysis for F30. 39

CHAPTER I

INTRODUCTION

Function-as-a-Service (FaaS) paradigm takes the granularity of cloud computing to the function level [1]. Hereby, *serverless functions* are defined as single-purpose, stateless functions. They are invoked through the Internet and hosted by cloud providers, who are responsible for provisioning and managing servers where these functions are deployed. There are several cloud providers that support such a model. Amazon Web Services (AWS) Lambda Functions [2], Microsoft Azure Functions [3], Google Cloud Functions [4], and IBM Cloud Functions [5] are all examples of the FaaS model.

Economic benefits of serverless computing heavily depend on the execution behavior of deployed functions [6] and the application workload. Serverless functions that are developed by application providers are deployed on servers of cloud providers. They get executed by application users who make use of these functions. Application providers need to pay cloud providers for the computing resources used while these functions are getting executed. The pricing is based on several factors. One of these factors is typically the amount of memory reserved for the deployed function [2]. Reservation of excessive memory increases costs unnecessarily. On the other hand, decreasing the amount of available memory increases the function execution time, which is also a factor that contributes to cost [2]. Moreover, insufficient memory can degrade the quality of service.

In this thesis, we introduce an approach[1] for optimizing the amount of memory to be reserved for the deployed serverless functions and a tool FACT (**FA**aS **C**ost minimization **T**ool) that automates this approach. The tool is composed of 3 components.

[1] *A preliminary version of the approach was published [7] as a conference paper and presented as a poster at the 36th ACM/SIGAPP Symposium On Applied Computing.*

The first component analyzes the runtime performance of a given function in various memory settings and derives a regression model. This component can be replaced or adapted for various platforms such as AWS and Google Cloud. The second component is a generic configuration optimizer. It automatically creates a nonlinear integer programming model based on the performance model obtained from the first component and configuration options as input. The model is solved by a generic solver to determine the optimal memory setting for minimizing cost without compromising the required quality of service. The last component is a function profiler, which measures the performance of the function for the optimized configuration settings and generates a report.

To the best of our knowledge, this is the first study that focuses on the FaaS paradigm for optimization and that aims at minimizing costs for the application provider. There have been many studies in the literature on the optimization of computing resources in cloud computing. However, these studies have focused on the provisioning of virtual servers [8, 9, 10], optimization of the use of resources by these servers [11, 12] or job/service scheduling [13] on these servers.

We present an evaluation of our approach with an industrial case study on the use of AWS in the context of Smart Home applications. Our case study involves 30 real functions as experimental objects. Results show that our approach is effective in accurately estimating the impact of memory settings on runtime performance. These estimations are useful in two ways. First, they help in determining optimal settings leading to significant cost reductions in case the allocated memory is more than actually required. Second, they can be used for detecting functions that are prone to performance issues.

The remainder of this thesis is organized as follows. In the following Chapter, we provide background information on serverless functions and AWS Lambda functions in particular. In Chapter 3 we summarize related studies. In Chapter 4, we introduce

our approach. We introduce our case study and discuss the results in Chapter 5. Finally, we conclude the thesis in Chapter 6.

CHAPTER II

AWS LAMBDA FUNCTIONS

AWS Lambda [2] is a serverless computing platform that supports the FaaS paradigm. Customers of this platform deploy their code together with a set of specifications including the input/output of the function and a set of configuration settings. AWS Lambda executes functions in response to events and it automatically manages the computing resources required by the executed functions. It currently supports many programming languages like JavaScript, Java, Python, Go, C#, F# and Ruby.

Amazon CloudFormation [14] is the service that handles the management of AWS resources for the deployed functions based on their specifications. The input and output of functions are specified with JavaScript Object Notation (JSON) [15]. In addition, a *template* file must be specified. This is also a text file, which can be either JSON or YAML [16] formatted. Amazon CloudFormation initializes the necessary resources based on this template file, which basically lists the set of resource requirements as a set of option blocks. A sample template specification of a service and a Lambda function in YAML is provided in Listing 2.1.

```
AttachPolicyFunction:
 Type: 'AWS::Serverless::Function'
 Properties:
  FunctionName: !Join
   - ''
   - - omega_attach_policy_
   - !Ref Environment
   - '_'
   - !Ref BlueGreen
  Handler: 'iotpolicy.AttachPolicy::'
  VpcConfig:
   SecurityGroupIds: ...
   SubnetIds: ...
   Runtime: java8
   CodeUri:
    Key: !Ref BucketFileJar
    Bucket: !Ref BucketName
   MemorySize: 512
   Environment:
    Variables:
     CustomerPoolId: !Ref PoolId
     CustomerClientId: !Ref ClientId
     configResource: !Ref Environment
     region: !Ref Region
     databaseDriver: !Ref DBDriver
     databaseUrl: !Ref DBUrl
     databaseUser: !Ref DBUser
     databasePassword: !Ref DBPassword
     IOTMQTTClient: !Ref IotMqttClient
   Role: !Ref ExecutionRole
   Timeout: 15
   Tracing: Active
```

Listing 2.1: A sample template specification of a service and a Lambda function in YAML format.

Listing 2.1 shows a part of the template specification regarding a real Amazon Lambda function, which is one of the experimental objects in our case study. The listing shows several properties (Lines 3-32) of this function named *AttachPolicy*. The name of the function instance (i.e., *FunctionName*) is created dynamically by concatenating a set of strings and values (Lines 4-9). The native function *Join* performs this concatenation. Another intrinsic function, *Ref* returns the value of a resource, which is specified in another template file. So, a hierarchy of template files can be created and resources can be referenced from other template files by their name. We can see in Listing 2.1 that the majority of the properties are defined via such references. The handler class for the function (i.e., *Handler*) is the second property specified (Line 10) after *FunctionName*. The rest of the properties are grouped under Virtual Private Cloud (VPC) Configuration settings (i.e., *VpcConfig*) between lines 11 and 32 in Listing 2.1. The majority of these settings are further grouped as environment variables (Lines 19-29) including those that are necessary for configuring a database connection (Lines 25-28). The Code Uniform Resource Identifier (URI) property (i.e., *CodeUri*) that is specified between lines 15 and 17 includes two attributes: an executable jar file and the Amazon Simple Storage Service (S3) bucket. Some other settings we can see in Listing 2.1 include the runtime environment (set as *Java 8* in line 14), memory size (set as 512 MB in line 18) and time out duration (set as 15 seconds in line 31). These properties can also be altered in the deployment phase via the AWS Command Line Interface (CLI). We used this interface to automate the control of services and execution of our experiments through scripts.

Application providers deploy their AWS Lambda functions by providing the necessary configuration settings as illustrated in Listing 2.1. The setup process is steered by these settings. Once deployed, AWS Lambda functions can be accessed by the application end users. The cost of each function invocation depends on its memory configuration and execution time. Pricing of the service is based on a pay-per-usage

model. The overall cost is calculated as shown in Equation 1 below.

$$cost = t \times c \times (m/1024) \times p \tag{1}$$

The contributing terms in Equation 1 represent the following:

- t: average function execution **time** in seconds
- c: invocation **count**
- m: allocated **memory** in MB
- p: **price** per invocation in $

Note that m does not represent the (average) amount of memory used during function execution. AWS Lambda charges for full provisioned memory capacity. Therefore, memory capacity, which is allocated but not utilized, increases the cost unnecessarily. On the other hand, low memory capacity can increase the function execution time, which is also a contributing factor (t) for the overall cost. Even worse, insufficient memory capacity can make the function unavailable and degrade the quality of service. Hence, the optimal memory allocation must balance memory and runtime performance without violating resource and configuration constraints. In the following Chapter, we explain our approach for cost minimization.

CHAPTER III

RELATED WORK

Minimization of costs and optimal utilization of computing resources in the context of cloud computing have been studied in the literature for more than a decade [10, 12, 13]. These studies can be divided into three categories. These categories are based on the types of problems addressed as explained below.

The first problem is minimizing the cost for the customers of cloud providers while provisioning virtual servers. Some of these studies try to address the problem for the use of a particular cloud provider. For instance, an optimization approach was proposed [9] for the allocation of virtual machines on Amazon Elastic Compute Cloud (EC2) based on load levels of applications. Two other algorithms [8] were previously proposed to optimize the cost of provisioning virtual servers on Amazon EC2 considering both long-term and short-term utilization. There have been also efforts to minimize the cost of provisioning virtual servers from multiple, alternative cloud providers [17].

The second problem is to minimize the costs for cloud providers by optimal use of physical resources. One of the approaches [11] that focus on this problem, optimizes resource utilization of physical machines based on workload prediction of virtual machines in cloud data centers. Data transfer among physical servers also constitutes a major factor of costs in these centers. Therefore, a nonlinear programming based approach [18] was proposed to minimize the cost of data transfer among servers. This was proposed as an alternative for the standard nearest source approach adopted by Amazon CloudFront [19].

The third problem is about effective scheduling of jobs or services. This is also

related to the second problem because the goal is to attain the optimal use of resources from the perspective of cloud providers. There exist a study [20] that aims at minimizing service response time by controlling service deployment and service request scheduling in dynamic cloud environments. Another study [21] proposes task scheduling techniques for cost minimization in hybrid clouds. Hereby, cost can be measured in various terms such as price, performance and energy. For instance, energy consumption can be minimized in data centers by effective scheduling [22].

Optimization efforts reported in the literature so far have focused on resource utilization of cloud providers and cost minimization for virtual server allocation by their customers. To the best of our knowledge, there have been no studies yet that aim at minimizing costs from the application provider perspective in the FaaS paradigm. There are three parties in this paradigm: cloud provider, application provider and end user. Application providers deploy their functions on servers provisioned by cloud providers. However, they do not reserve a particular (virtual) server for this purpose. They only specify basic requirements such as the amount of memory that should be available. Functions get executed when end users access the corresponding applications. Then, the pricing for the application providers is determined by the number of function invocations and computing resources used during these invocations. In the following Chapter, we present our approach and tool that can be used for minimizing costs for application providers.

CHAPTER IV

THE TOOL AND THE OVERALL APPROACH

Figure 1 depicts our overall cost minimization approach that is automated with our tool FACT. FACT is available online[1] and it is composed of 3 main components: *i) Performance Analyzer*, *ii) Configuration Optimizer* and *iii) Function Profiler*.

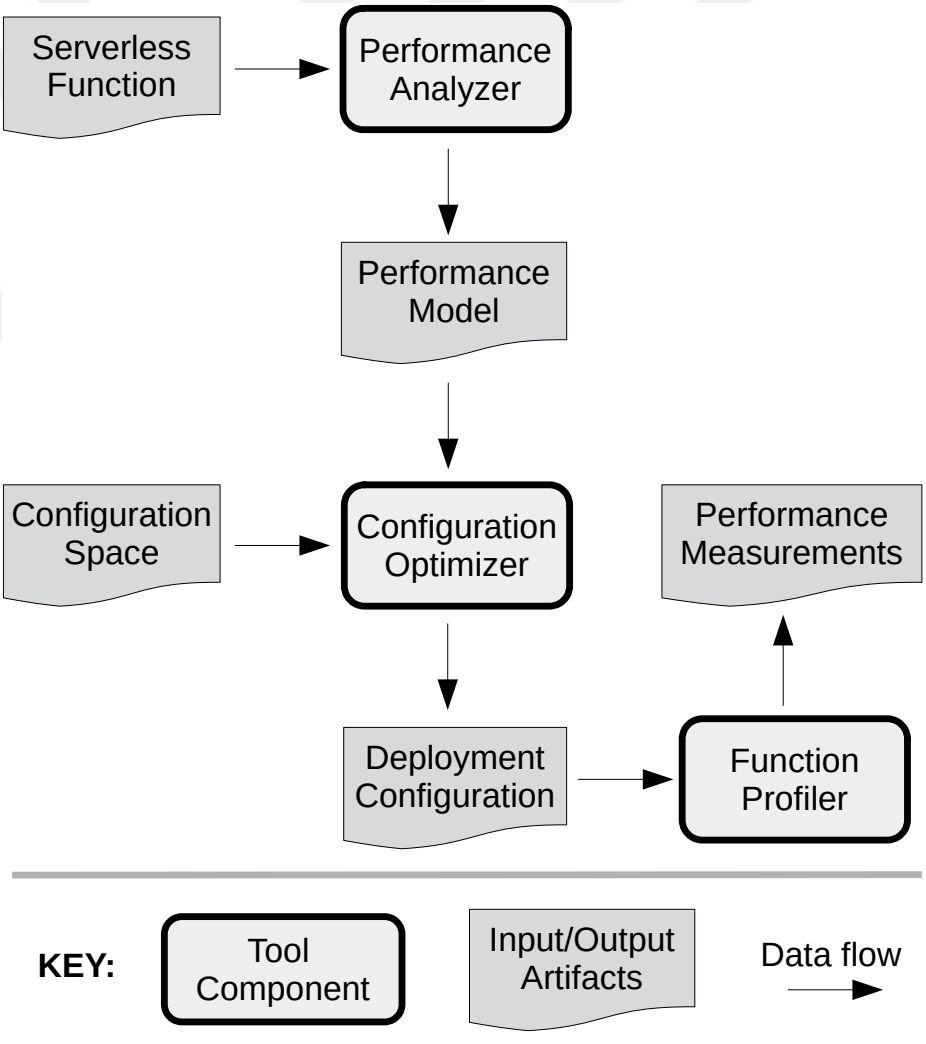

Figure 1: The overall approach and components of FACT.

[1] `https://github.com/ozgursedef/fact`

Performance Analyzer takes the serverless function to be deployed as input. It executes the function in various memory settings and measures its running time for every setting multiple times. The algorithm explains in Appendix B.

Measurements are currently repeated 10 times for every memory setting that is a multiple of 64 MB. The average of measurements for each setting is recorded. Then regression analysis (curve fitting) is performed to find a model that describes the relation between the available memory for the function and its running time. A sample result of the regression analysis for one of the functions used in our case study is depicted in Figure 2. Hereby, the relationship is modelled as a power function [23]. *Performance Analyzer* tries to fit alternative functions and computes the correlation coefficient for each fitting to find a model with a strong correlation. The performance of functions that are used as experimental objects in our study (See Concept 5) were best modelled with either a linear function or power function.

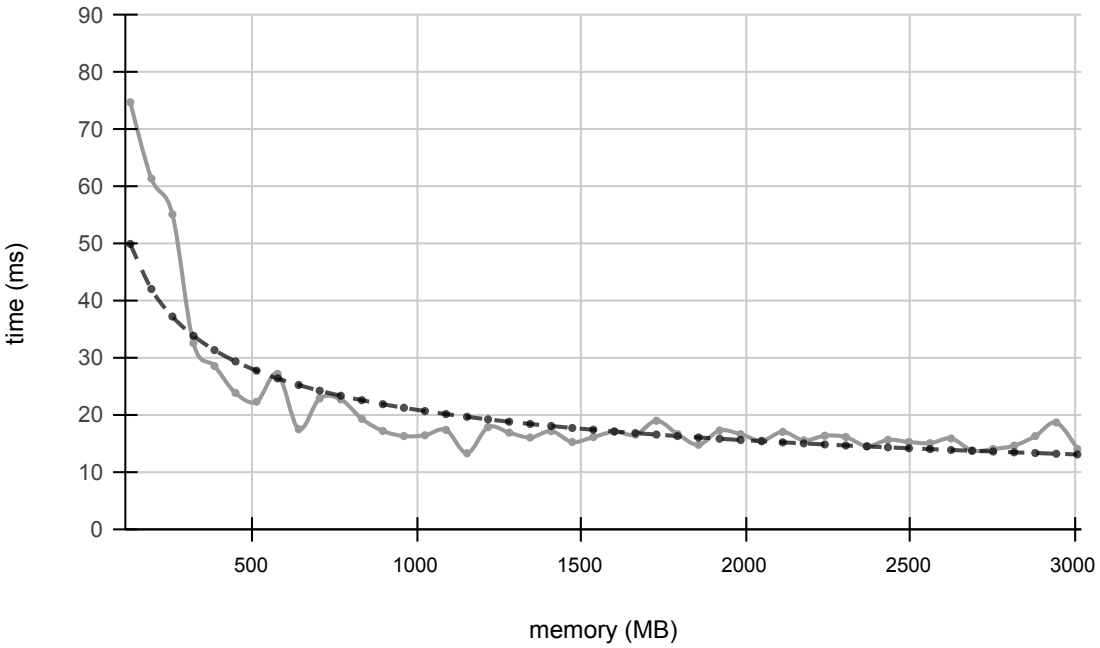

Figure 2: Result of the regression analysis for one of the functions (F1) used in the case study (See Table 1).

Configuration Optimizer takes the result of regression analysis as input. This result is actually the running time (t) as a function of the available memory ($f(m)$) as depicted in Figure 2[2]. This function is used both for defining the objective function and for defining one of the constraints in the optimization process since t must be kept below a certain limit for not compromising the quality of service. The second input of *Configuration Optimizer* is the configuration space for the deployment of a serverless function. This basically specifies the possible memory settings that can be used. The model used for optimization is listed in Equations 2-7.

$$z = \min m \times f(m) \tag{2}$$

$$s.t.$$

$$m \leq MAX, \tag{3}$$

$$m \geq MIN, \tag{4}$$

$$m - w \times STEP = 0, \tag{5}$$

$$w \geq 0, \tag{6}$$

$$f(m) \leq LIMIT \tag{7}$$

The objective (Equation 2) is to minimize $m \times f(m)$. $f(m)$ gives the running time of the function, t in terms of m. These two factors, m and t are the only controlled contributing factors for cost (See Equation 1). The first 4 constraints are defined based on the configuration space. The first and second constraints (Equations 3 and 4) define the maximum (MAX) and minimum (MIN) memory settings available, respectively. The third and the fourth constraints (Equations 5 and 6) specify the step size ($STEP$). These two equations basically enforce that m is a multiple of $STEP$, i.e., $m \equiv 0 mod(STEP)$. For instance, one can allocate memory for AWS Lambda

[2] *Regression analysis results for the other functions can be found in Appendix A*

functions as multiples of 64 MB. Hence, the *STEP* parameter should be set as 64 for AWS. The last constraint (Equation 7) specifies the limit on t ($LIMIT$), which is specified as a function of m (i.e., $f(m)$) as derived by *Performance Analyzer*. The model listed in Equations 2-7 is instantiated with concrete values for parameters *MAX*, *MIN*, *STEP*, *LIMIT* and solving this model instance provides the optimal value of m for the server configuration.

Configuration Optimizer uses the GEKKO Optimiation Suite [24, 25] to solve the nonlinear model listed in Equations 2-7. The model is instantiated for each function based on the configuration space of the function and the performance model provided by *Performance Analyzer*. A code snippet from FACT is shared in Appendix B, which illustrates the use of GEKKO for solving the constructed optimization model. Note that this is a generic model and it can be provided to any generic integer nonlinear programming solver such as Couenne [26] in principle.

Function Profiler takes the optimized configuration (i.e., the optimal m value computed) and runs a number of tests to observe the behavior of t for this configuration. We currently set the number of tests as 100. *Function Profiler* reports performance measurements as output. These measurements show the variation in runtime performance of the function and they can be used for validating that t remains below the specified limit. One might consider starting over with the process and iteratively refine constraints in case expectations are not met according to the output of *Function Profiler*. In the following Chapter, we present an industrial case study conducted to evaluate our approach.

CHAPTER V

INDUSTRIAL CASE STUDY

In this Chapter, we introduce our case study on Smart Home applications being developed and maintained at Vestel[1], which is one the largest consumer electronics companies in Europe. Vestel develops and maintains many AWS Lambda functions, which are used by/for smart home appliances all over the world. In addition to the products of its own, Vestel is producing these appliances for 157 different brands from 145 different countries [27]. Home appliances were historically standalone electromechanical devices only. However, they have been transformed into smart Internet of Things (IoT) products [28] such as smart TVs, smart ovens, smart refrigerators, smart air conditioners and smart washing machines.

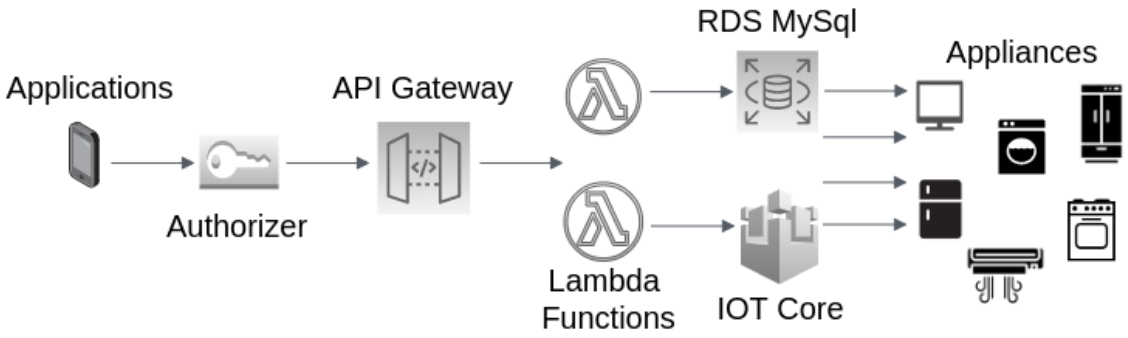

Figure 3: An overview of the use of AWS Lambda functions in the context of Smart Home applications.

Figure 3 depicts the use of AWS Lambda functions in the context of Smart Home applications. Hereby, a request that is sent by application users is first authorized with client identification and user pool signature. The Application Programming Interface (API) Gateway dispatches successfully authorized requests to corresponding Lambda

[1]http://www.vestel.com.tr

functions, which execute the service logic. Some of these functions can execute queries on a database. Some of them can control home appliances. Amazon IoT Core [29] facilitates a publish/subscribe mechanism, where all these appliances get connected as IoT devices.

We conducted a case study by selecting 30 AWS Lambda functions as our experimental objects (See Table 1). We applied our approach for optimizing the memory settings for these functions, which are maintained and deployed by Vestel for its Smart Home applications. We aimed at answering the following research questions with our case study:

- **RQ1**: Is it possible to estimate the performance impact of memory settings accurately?
- **RQ2**: How much cost reduction can be achieved without compromising the quality of service?

The first question is about the accuracy of our estimation, which is used for defining the objective function and constraints of the optimization model. The second question is about the usefulness of the approach. Given that we can give a positive answer for *RQ1*, we would like to know if our approach can reduce costs when applied to real life systems.

We describe our experimental setup in the following subchapter. Then we present and discuss the results. We conclude the Chapter by elaborating on threats to validity of our case study.

5.1 Experimental Setup

Table 1 lists the AWS Lambda functions used in our case study. The first column enumerates these functions. The second column lists the programming languages used in their implementation. The third column lists the lines of code for each function. The fourth and the fifth columns specify whether these functions access to any

other external services, and a database, respectively. The last column lists the maximum latency allowed for each function as performance requirements. Hence, function execution times are not supposed to exceed these values.

Table 1: AWS Lambda functions that are used as experimental objects in the case study.

Func. #	Lang.	LOC	Usage of other services	Access to Database	Latency Limit (ms)
F1	Java	48	No	Yes	400
F2	Java	72	Yes	Yes	800
F3	Node	75	Yes	Yes	400
F4	Node	91	Yes	No	400
F5	Node	101	Yes	Yes	400
F6	Node	114	No	Yes	400
F7	Java	527	Yes	Yes	800
F8	Node	138	Yes	Yes	400
F9	Node	50	No	Yes	400
F10	Java	102	Yes	Yes	400
F11	Java	78	Yes	Yes	400
F12	Node	156	No	Yes	400
F13	Java	96	Yes	Yes	400
F14	Node	123	Yes	Yes	400
F15	Java	89	No	Yes	400
F16	Java	56	No	Yes	800
F17	Java	78	Yes	No	800
F18	Java	83	Yes	No	800
F19	Java	145	No	Yes	800
F20	Java	167	Yes	Yes	800
F21	Java	185	Yes	Yes	800
F22	Java	176	Yes	Yes	800
F23	Java	198	Yes	Yes	800
F24	Java	134	Yes	Yes	800
F25	Node	145	No	Yes	400
F26	Go	76	Yes	Yes	400
F27	Java	89	No	Yes	800
F28	Go	67	Yes	Yes	400
F29	Java	124	Yes	Yes	800
F30	Java	98	Yes	Yes	800

The specified values are set as the *LIMIT* parameter in the optimization model (See Equations 2-7) Note that the function $f(m)$ that is derived by *Performance*

Analyzer provides an estimation of the expected execution time of a given function, rather than the worst-case execution time.

There are 3 additional parameters other than the *LIMIT* parameter for initializing the optimization model provided in Equations 2-7. These parameters are listed in Table 2 together with the values assigned for each. These values are determined based on the configuration space currently provided by AWS Lambda. Hereby, the minimum (MIN) and maximum (MAX) amounts of memory that can be allocated for a serverless function are 128 MB and 3,008 MB, respectively. The *STEP* parameter is set as 64 since the amount of memory allocated can only be a multiple of 64. We calculated the correlation coefficient, r of regression [23] and analyzed performance measurement reports of *Function Profiler* to answer *RQ1*. We calculated the amount of cost reduction achieved per function invocation to answer *RQ2*. We present and discuss the results in the following subchapter.

Table 2: Parameter values used in the case study.

Parameter	Value
MAX	3008
MIN	128
STEP	64

5.2 Results and Discussion

Results on the accuracy of estimation are listed in Table 3. Hereby, the second column lists the correlation coefficient, r of regression for each subject function. The smallest r value is obtained as 0.75 for F8 and F14. Even this value is interpreted as strong correlation [23]. We can observe that r values are above 0.9 for half of the functions. These values suggest very strong correlation. The third column lists the estimated running time for the optimal memory setting. The last 3 columns are obtained from *Function Profiler*. They list the minimum and maximum running times measured as well as the mean of all the measurements. These values show that test results are

Table 3: Results regarding the accuracy of estimation.

Func. #	r	Estimated t value (ms)	Measured t values (ms) min	max	mean
F1	0.88	50	7.79	80.25	25.16
F2	0.91	790.75	384.72	819.89	498.77
F3	0.92	56.70	9.51	58.7	25.32
F4	0.96	70.45	11.62	67.51	35.08
F5	0.85	43.54	10.52	75.81	29.41
F6	0.87	57.73	7.3	59.05	21.21
F7	0.85	786.22	448.86	941.89	576.26
F8	0.75	376.88	141.01	440.17	209.17
F9	0.78	108.64	78.71	289.68	126.19
F10	0.92	198.37	104.11	667.39	202.42
F11	0.96	390.29	192.07	418.37	265.61
F12	0.82	98.83	44.11	220.19	79.21
F13	0.93	396.32	103.94	318.35	166.40
F14	0.75	68.42	24.32	155.6	47.83
F15	0.83	326.16	170.35	864.86	366.10
F16	0.90	777.37	532.3	941.04	656.68
F17	0.87	782.32	603.89	983.03	736.04
F18	0.95	789.42	359.28	879.53	466.36
F19	0.90	794.82	571.05	907.57	708.45
F20	0.90	795.98	573.02	841.43	685.24
F21	0.92	790.59	327.25	674.8	430.56
F22	0.92	404.99	255.11	445.79	329.93
F23	0.91	370.48	134.19	776.33	317.01
F24	0.87	793.66	517.16	908.86	641.98
F25	0.86	73.46	27.2	152.29	69.54
F26	0.76	117.46	8.75	393.35	272.58
F27	0.90	789.89	350.63	922.03	499.81
F28	0.77	91.71	23.68	296.41	118.45
F29	0.82	756.15	425.77	787.88	543.40
F30	0.91	725.78	433.46	798.43	548.49

consistently close to the estimations. The box plot in Figure 4 and Figure 5 shows the overall distributions of all the measurements performed by *Function Profiler* for all the functions that are enumerated on the x-axis. Time measurements are all in milliseconds as listed on the y-axis. We can see that measurements are mainly clustered around the estimated value and several minimum and maximum values actually

represent outliers. Hence, we can conclude regarding *RQ1* that it is indeed possible to estimate the performance impact of memory settings accurately by performing regression analysis.

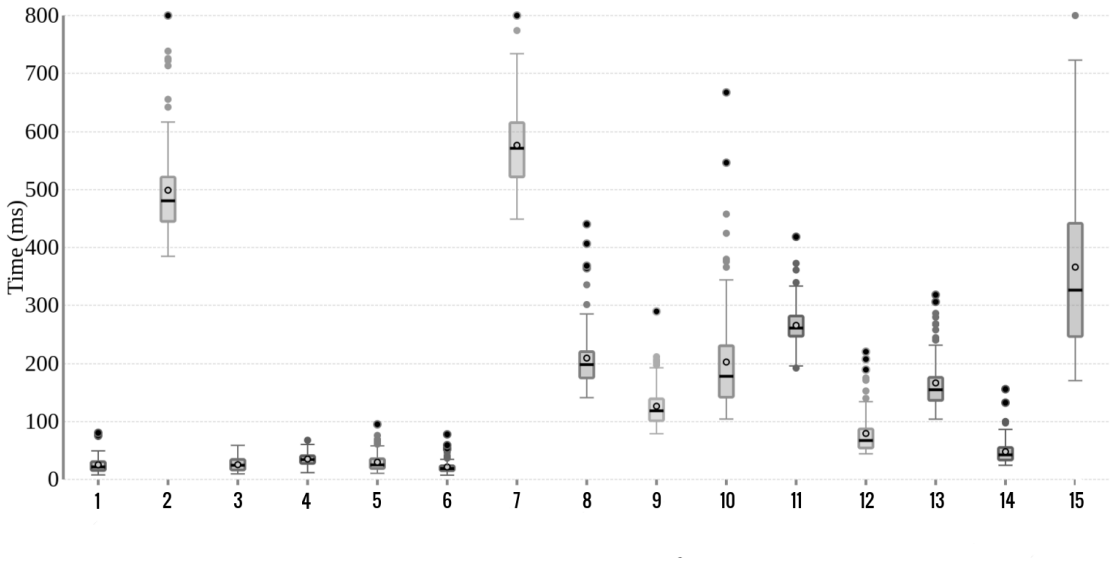

Figure 4: Runtime performance measurements for functions F1 - F15.

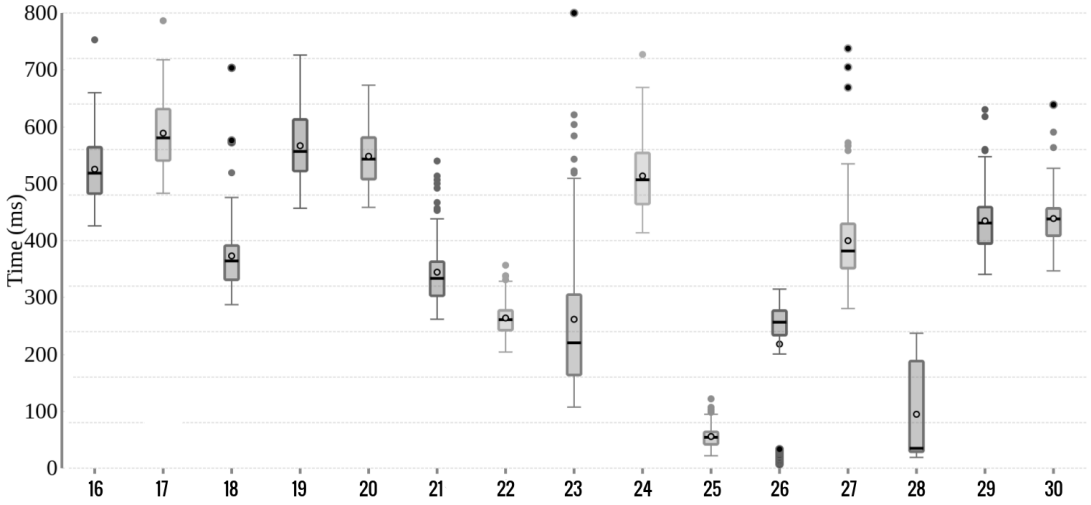

Figure 5: Runtime performance measurements for functions F16 - F30.

Results on the amount of cost reduction are listed in Table 4. Hereby, the second column lists the allocated memory for each function, which is set as 512 MB by

default. The third column lists the optimized memory settings. The last column lists the amount of profit that is gained as a result of the change of settings. The amount of profit is calculated in US dollars, per one million invocations of a function. We can see that cost reduction was possible for 16 of the 30 functions, where the

Table 4: Results regarding the amount of cost reduction.

Func. #	Default memory setting (MB)	Optimal memory setting (MB)	Profit per 1M invocations ($)
F1	512	128	6.26
F2	512	1152	-
F3	512	128	6.26
F4	512	128	6.26
F5	512	128	6.26
F6	512	128	6.26
F7	512	1344	-
F8	512	256	4.17
F9	512	128	6.26
F10	512	320	3.13
F11	512	1728	-
F12	512	128	6.26
F13	512	768	-
F14	512	128	6.26
F15	512	320	3.13
F16	512	2944	-
F17	512	2880	-
F18	512	1344	-
F19	512	2624	-
F20	512	2688	-
F21	512	1024	-
F22	512	2368	-
F23	512	384	2.78
F24	512	1728	-
F25	512	128	6.26
F26	512	128	6.26
F27	512	1216	4.17
F28	512	128	6.26
F29	512	1536	-
F30	512	1536	-

optimized memory setting turns out to be less than 512 MB. These listed figures

might seem insignificant at the first sight. However, they lead to significant cost reductions due to a high number of connected devices and an excessive number of function invocations. There will be around 400,000 new connected home appliances within this year according to the forecast of the Vestel Sales Department. Some of these devices communicate with serverless functions very frequently. For instance, a smart oven currently performs 70 function calls during a single cooking cycle of one hour. As a result, some of these functions are expected to be called at least hundreds of millions of times by the end users of appliances every month. Moreover, these numbers are expected increase every year. Hence, we can conclude regarding *RQ2* that significant cost reduction is possible by utilizing our approach.

An additional finding of our case study was about the 14 out of 30 functions, for which cost reduction was not possible. In fact, the optimal memory settings turned out to be higher than 512 MB for these functions. The overall cost can still be the same or lower due to the increased latency. However, we did not calculate the actual cost for these functions. The latency values for them were not acceptable anyhow. Some of them were subject to known performance issues as reported from the field. Thundra[2] was being used as a framework to speed up these functions. Thundra provides warm-up support for AWS Lambda functions to prevent cold starts. It facilitates periodic invocation of a set of functions, for which a set of important objects can be created to be available beforehand. Eventually, some of the 14 functions were refactored due to performance issues. Some of them were implemented from scratch. Therefore, our approach was useful for not only reducing costs but also detecting performance issues. We discuss the threats to validity of our study in the following section.

[2]https://github.com/thundra-io/thundra-lambda-warmup

5.3 Threats to Validity

We have performed a single case study in the context of a company. Therefore, it is subject to an external validity threat [30]. More case studies can be conducted in different contexts to increase the generalizability of results.

Our conclusions are supported by statistically significant correlations between our estimations and measurements. However, our measurements are subject to internal validity threats since we do not have the complete control of the overall environment. Runtime performance can be affected by many factors including network delays. We repeated our measurements multiple times to mitigate internal validity threats. Nevertheless, it is very hard, if possible, to estimate the worst-case execution time of serverless functions. Our regression model provides an estimation of the expected runtime performance. Therefore, one might consider to include an additional offset while determining a threshold limit of latency for optimizing the memory settings.

CHAPTER VI

CONCLUSION

We introduced an automated approach for optimizing the amount of memory to be reserved for servers where serverless functions are deployed. In our approach, we use a regression model to relate available memory and the runtime performance of a function. Then, we define a nonlinear integer programming model for optimizing the allocated memory to minimize costs without violating configuration and quality constraints. We evaluated our approach with an industrial case study on the use of Amazon Web services in the context of Smart Home applications. We applied our approach on 30 real AWS Lambda functions. We showed that our approach is effective in accurately estimating the impact of memory settings on runtime performance and determining optimal settings leading to significant cost reductions. Our results also pinpointed performance issues caused by default settings.

APPENDIX A

RESULTS OF REGRESSION ANALYSIS FOR ALL FUNCTIONS OF THE SMART HOME CASE STUDY

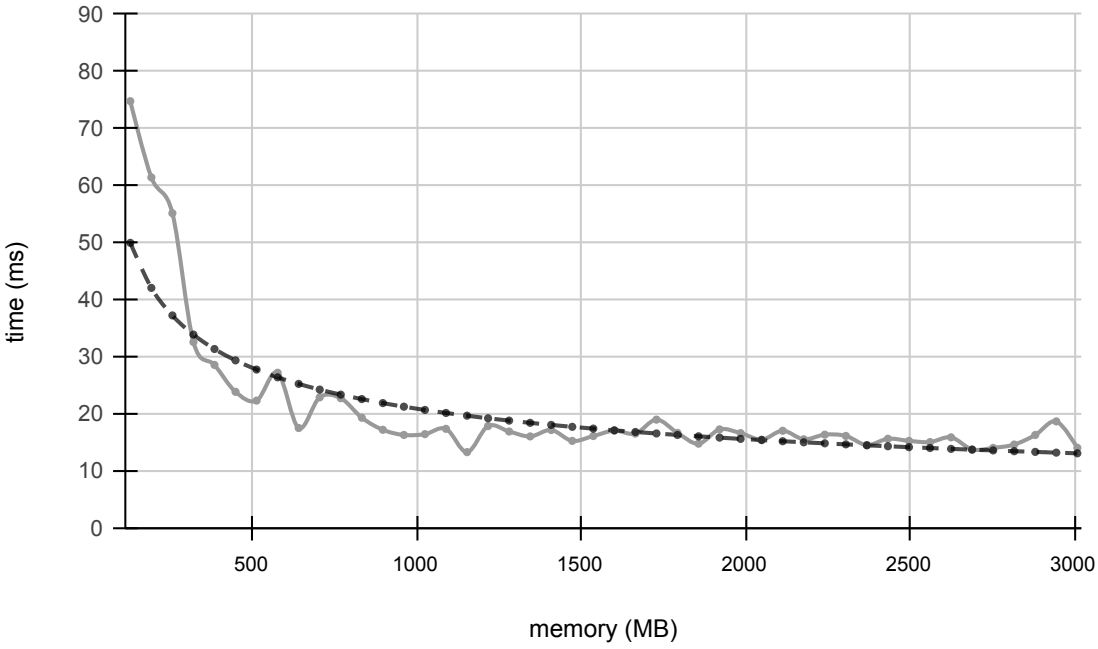

Figure 6: Result of the regression analysis for F1.

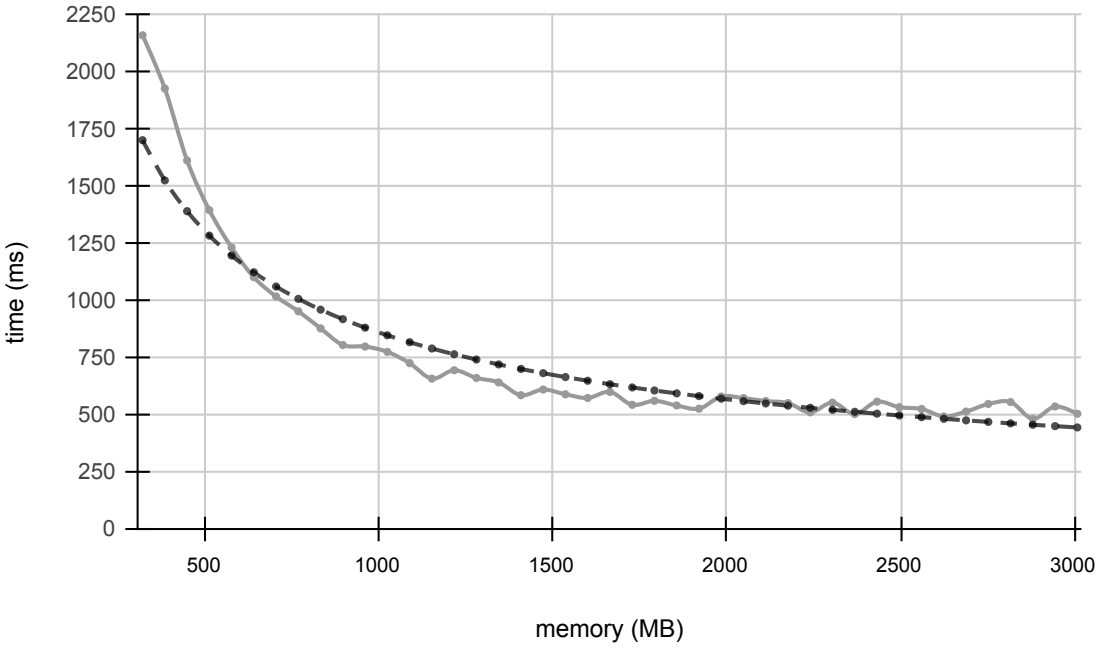

Figure 7: Result of the regression analysis for F2.

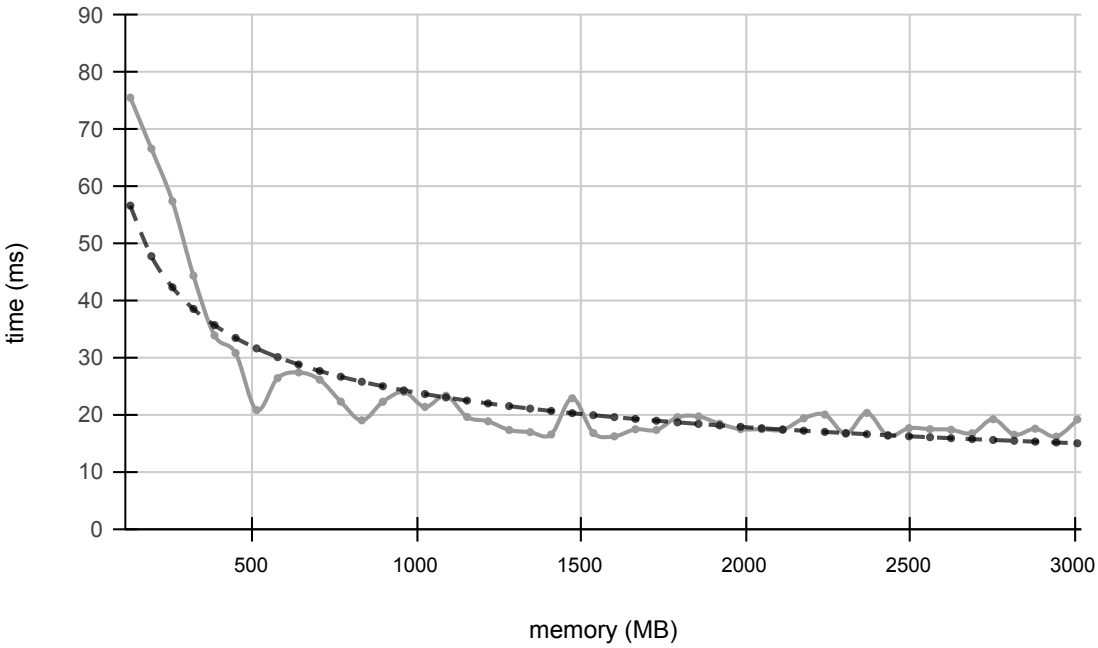

Figure 8: Result of the regression analysis for F3.

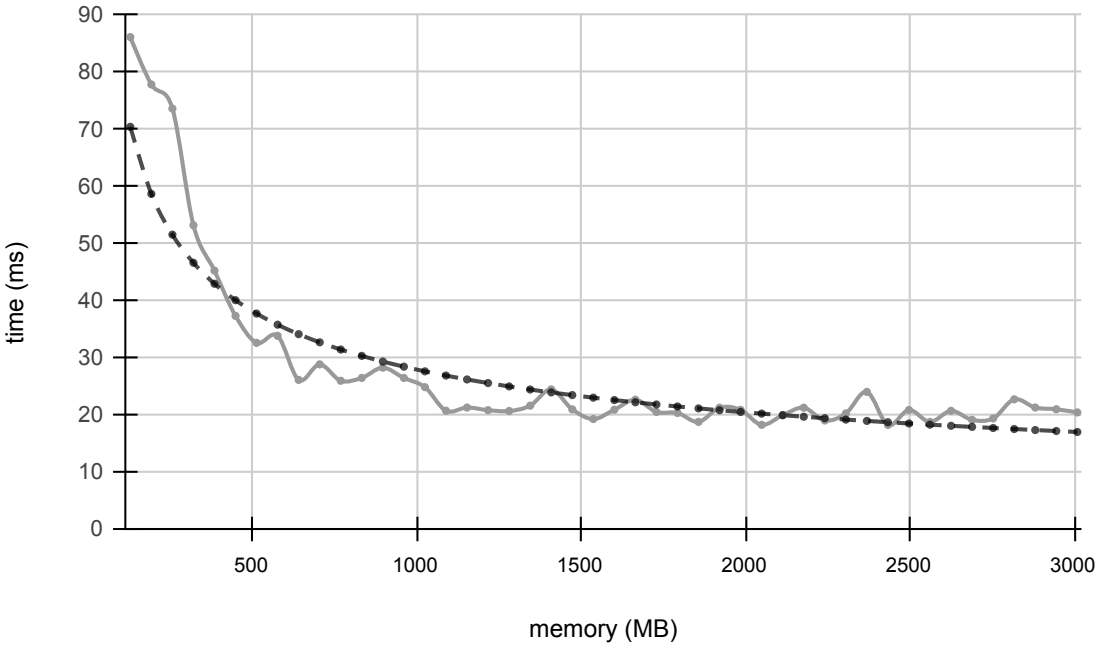

Figure 9: Result of the regression analysis for F4.

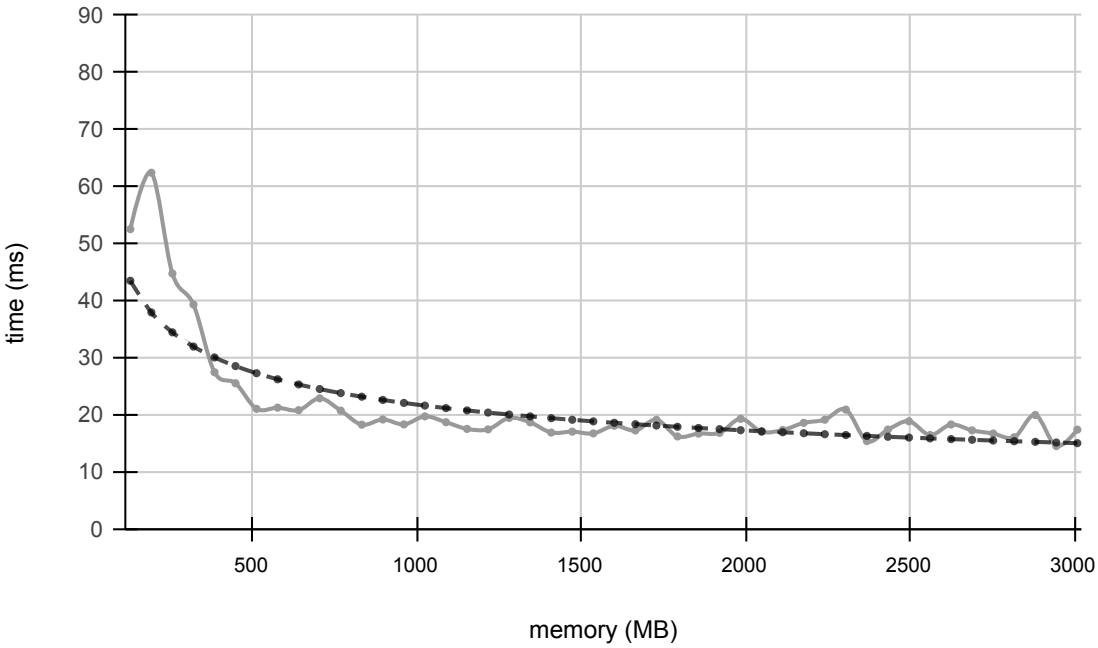

Figure 10: Result of the regression analysis for F5.

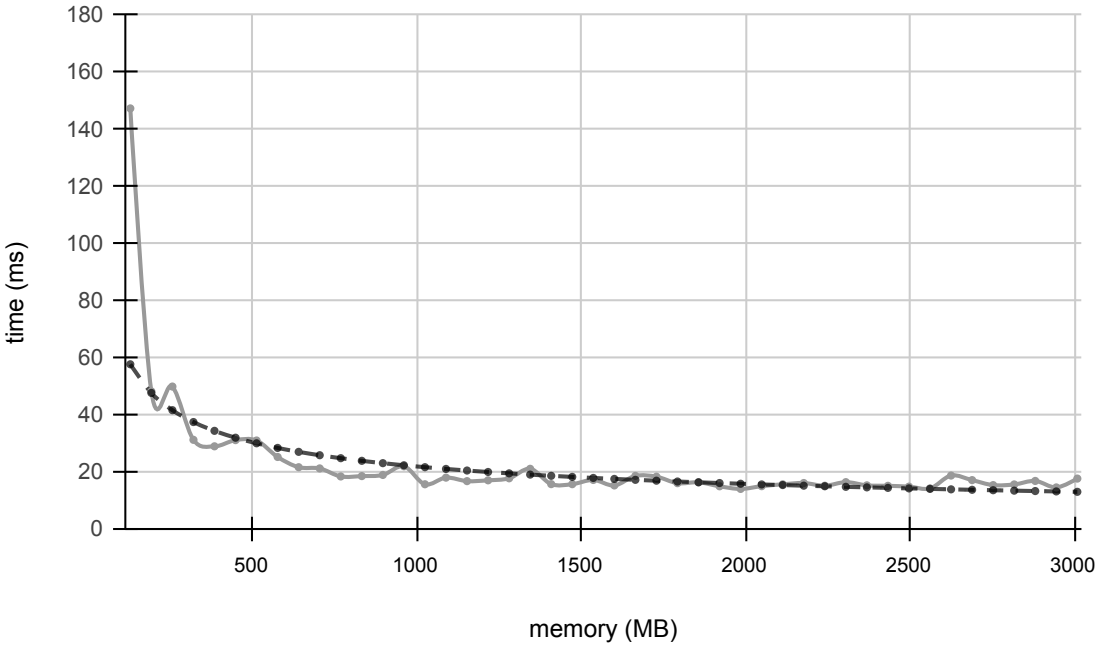

Figure 11: Result of the regression analysis for F6.

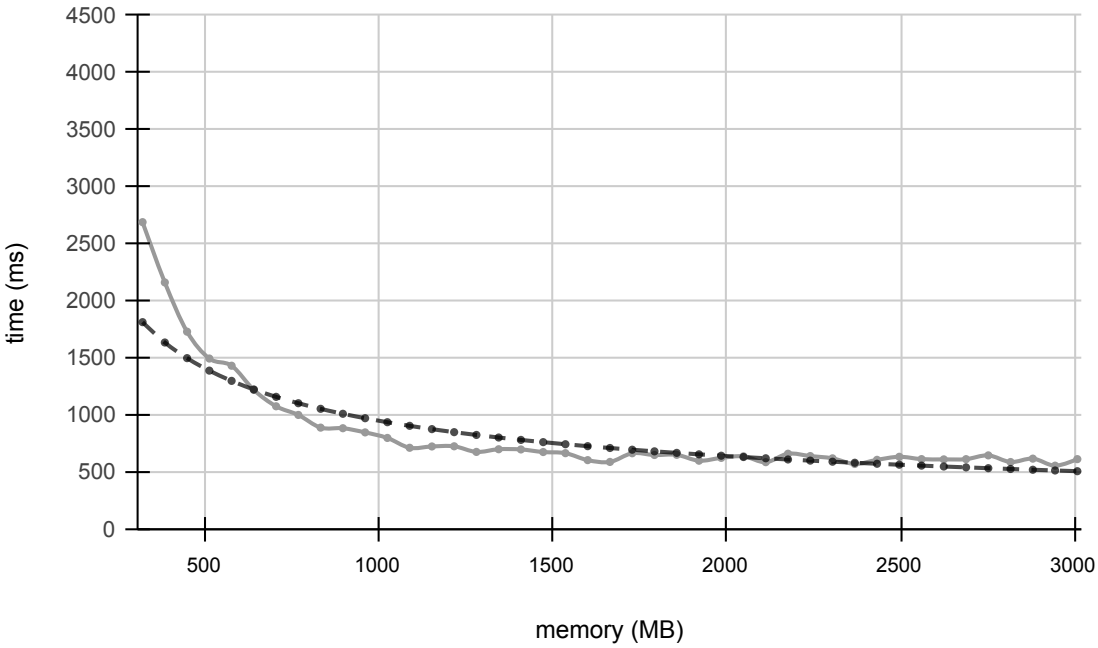

Figure 12: Result of the regression analysis for F7.

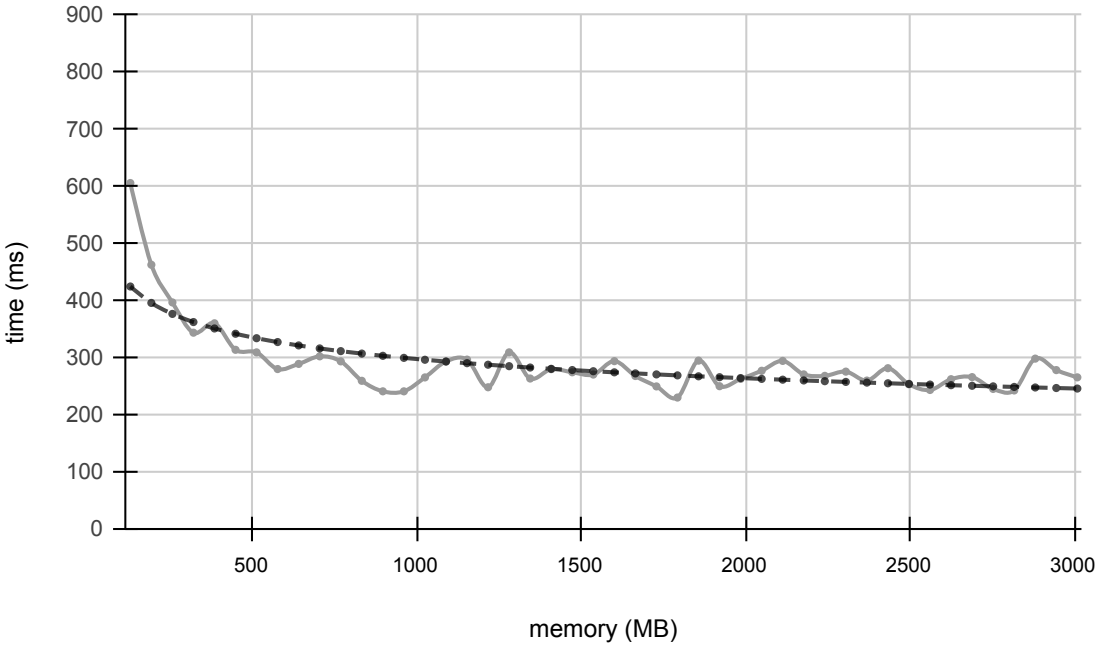

Figure 13: Result of the regression analysis for F8.

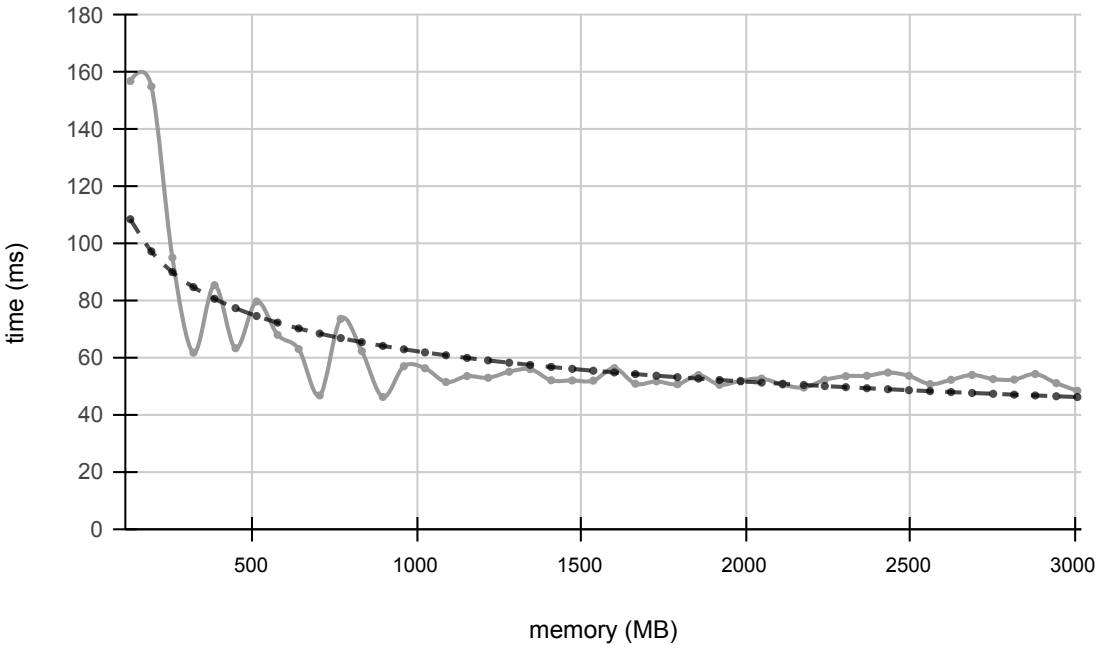

Figure 14: Result of the regression analysis for F9.

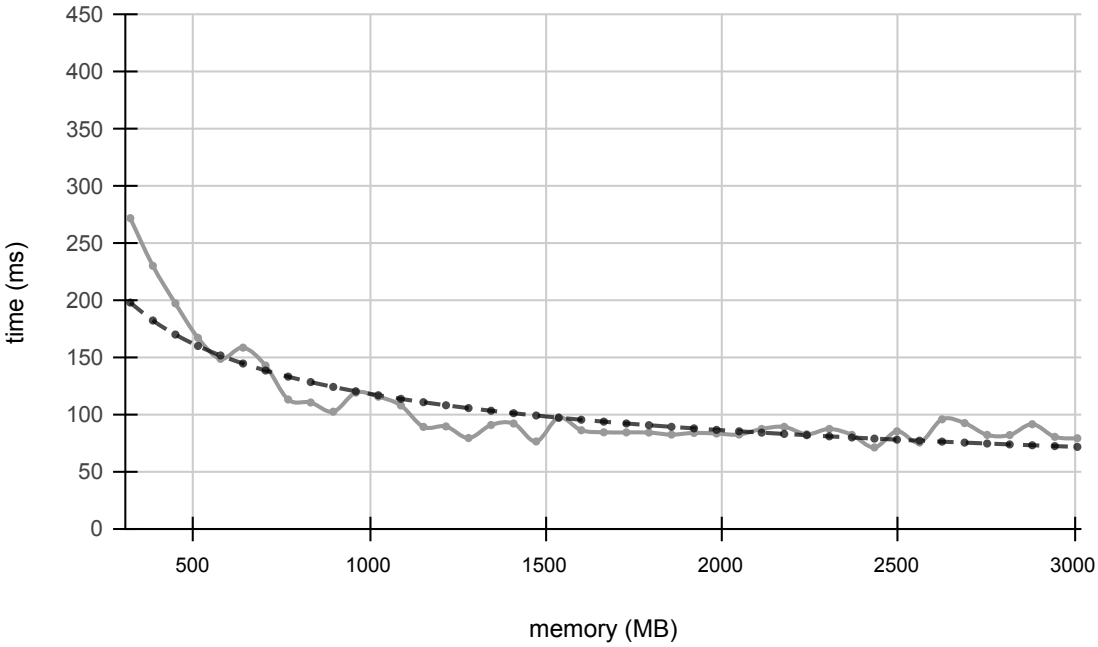

Figure 15: Result of the regression analysis for F10.

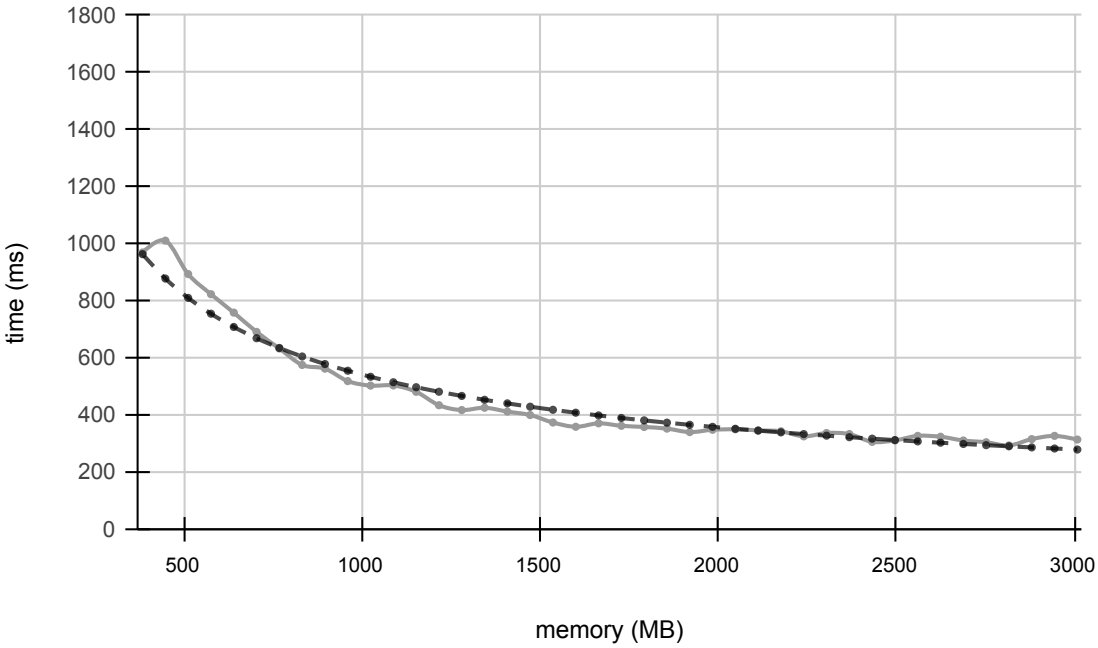

Figure 16: Result of the regression analysis for F11.

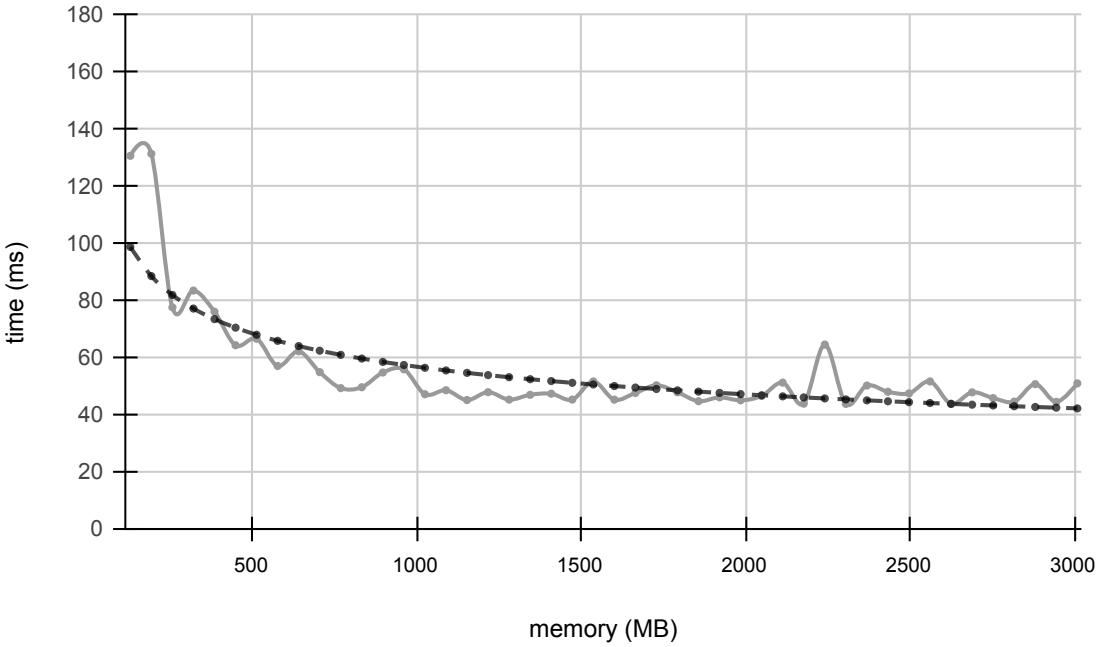

Figure 17: Result of the regression analysis for F12.

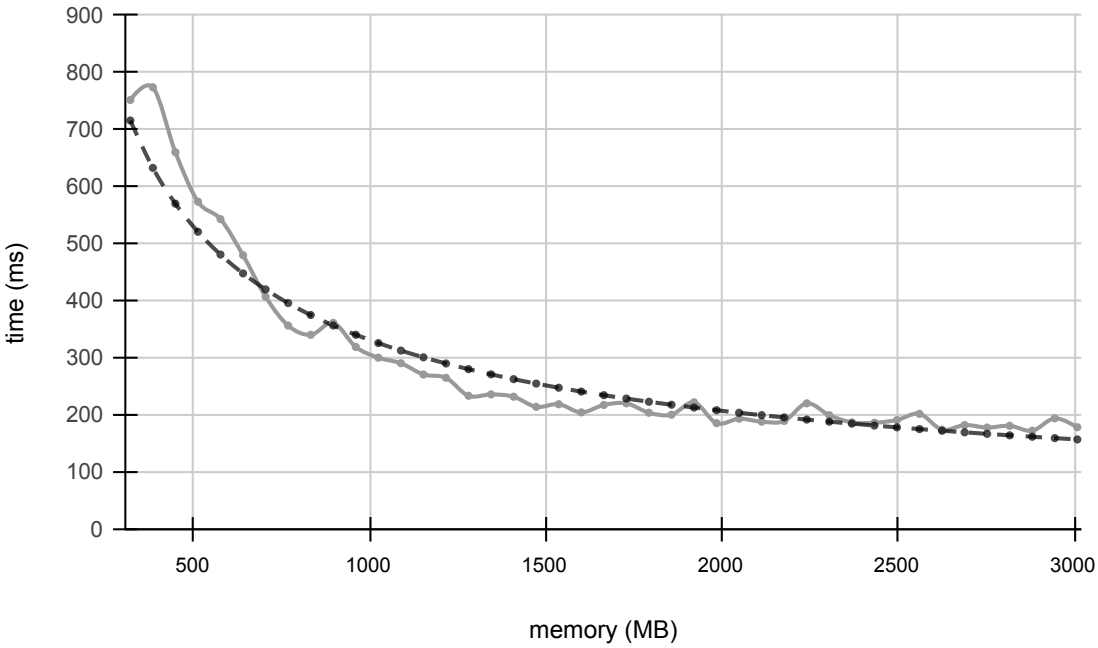

Figure 18: Result of the regression analysis for F13.

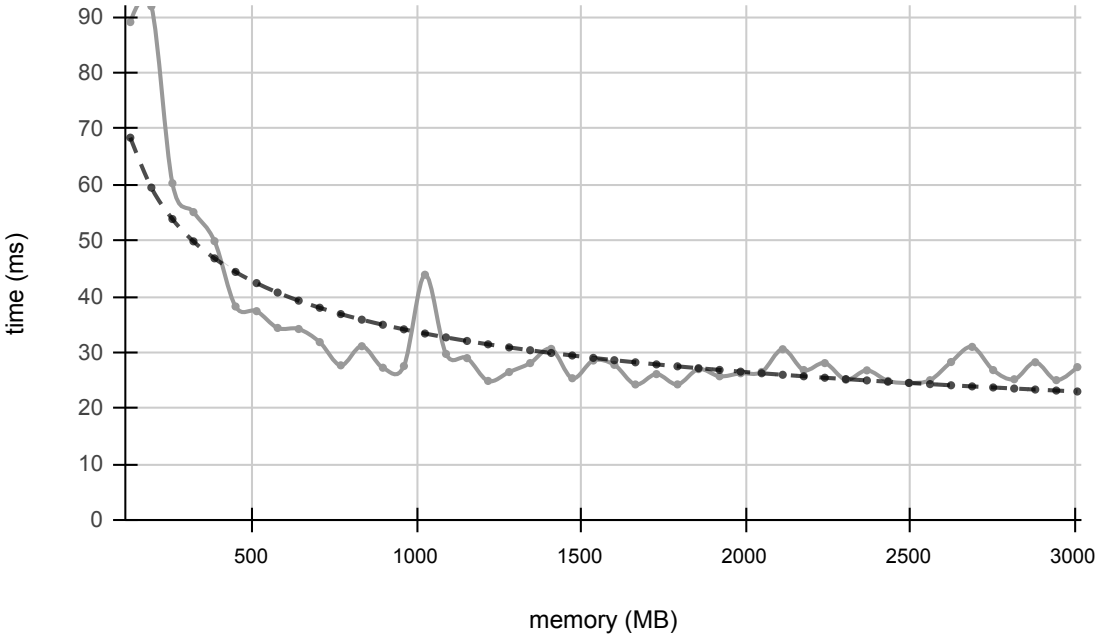

Figure 19: Result of the regression analysis for F14.

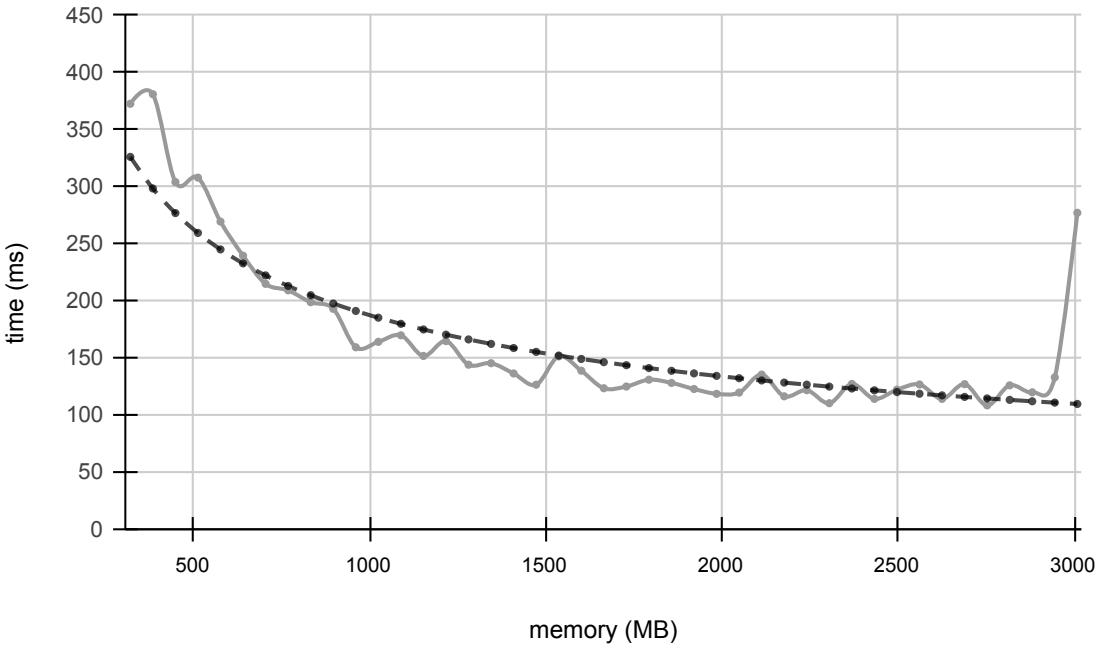

Figure 20: Result of the regression analysis for F15.

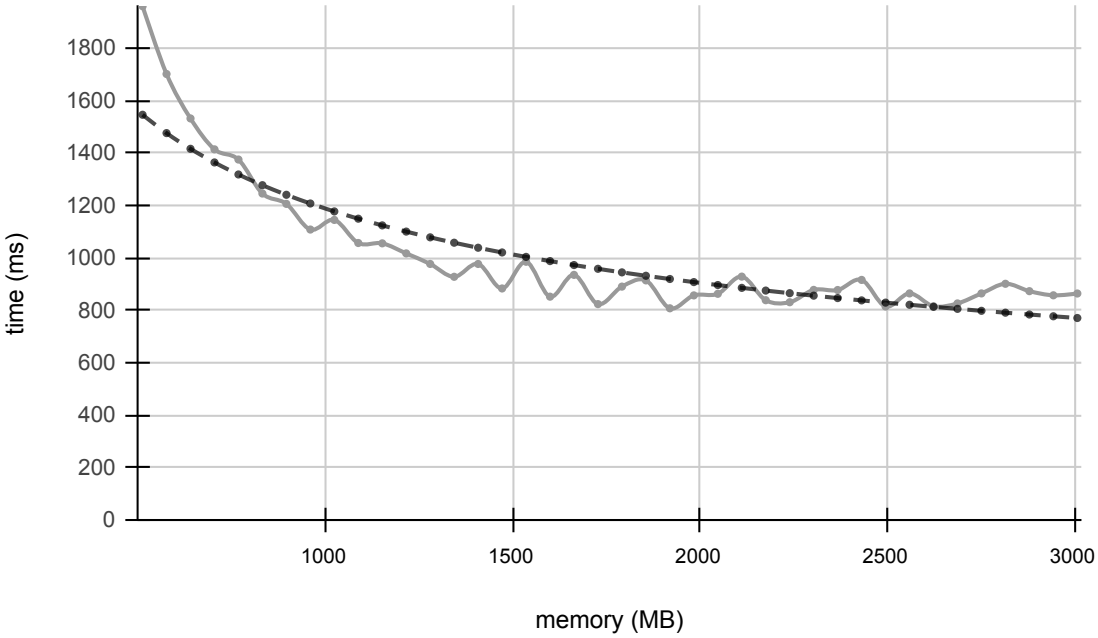

Figure 21: Result of the regression analysis for F16.

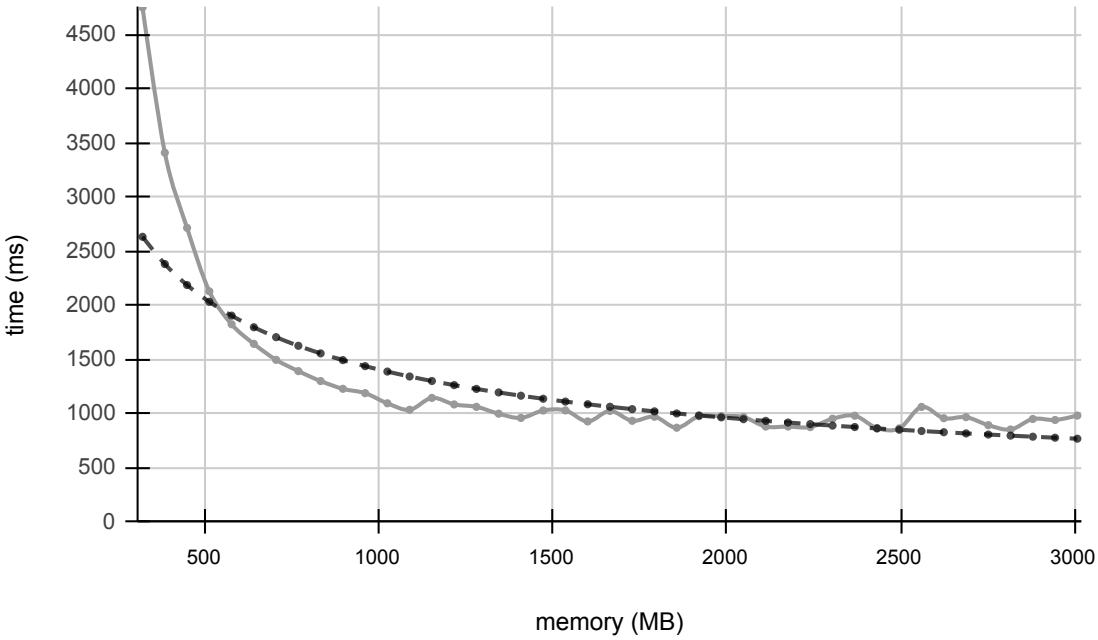

Figure 22: Result of the regression analysis for F17.

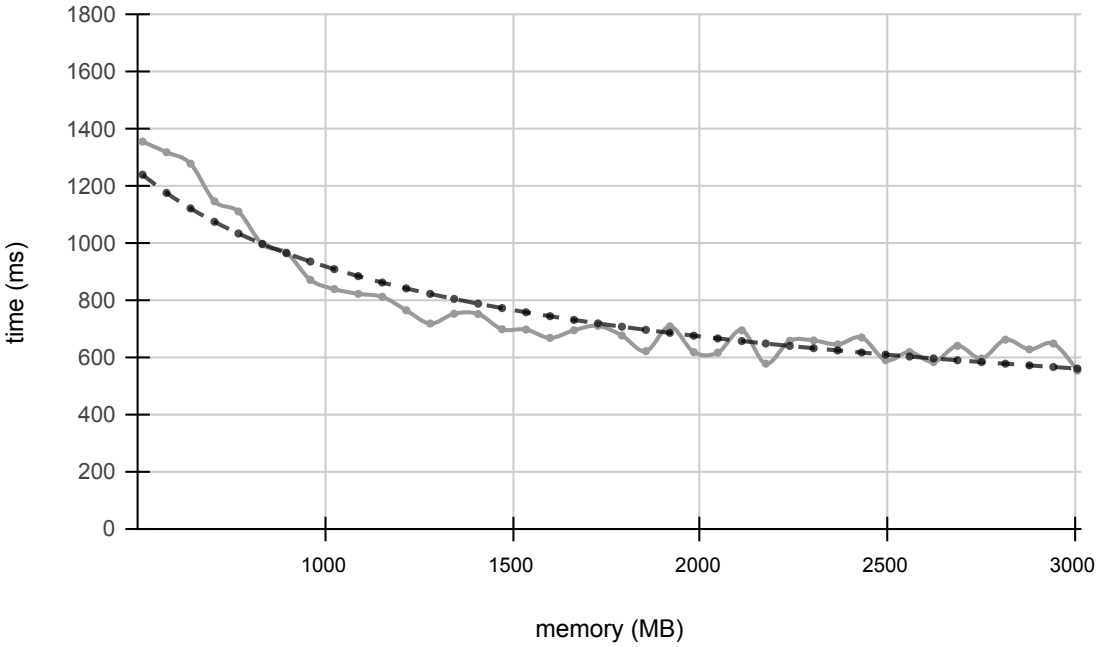

Figure 23: Result of the regression analysis for F18.

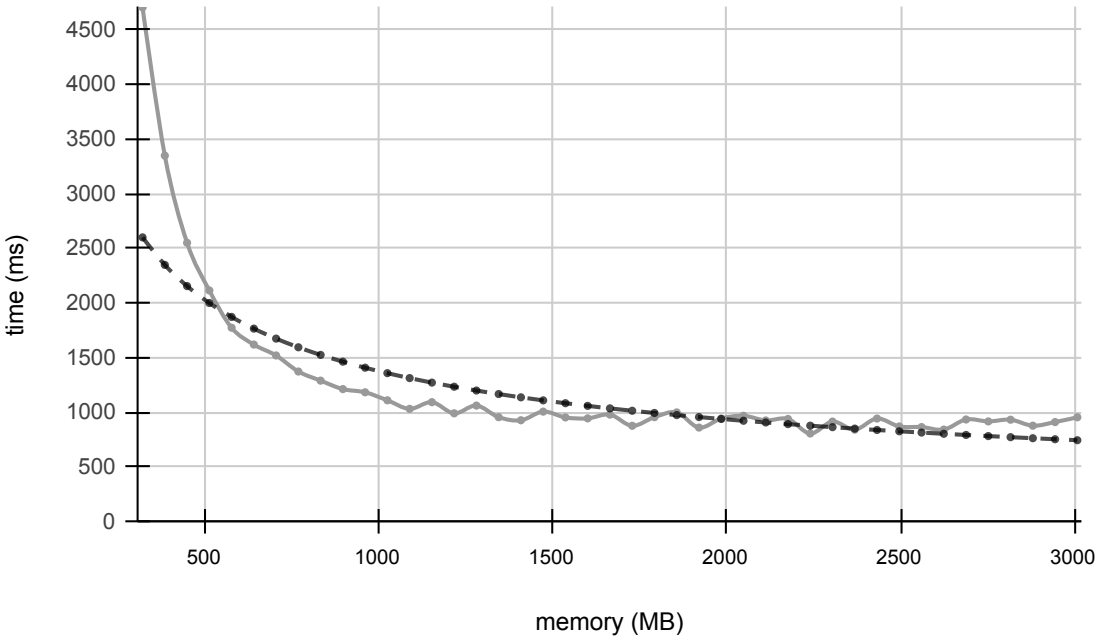

Figure 24: Result of the regression analysis for F19.

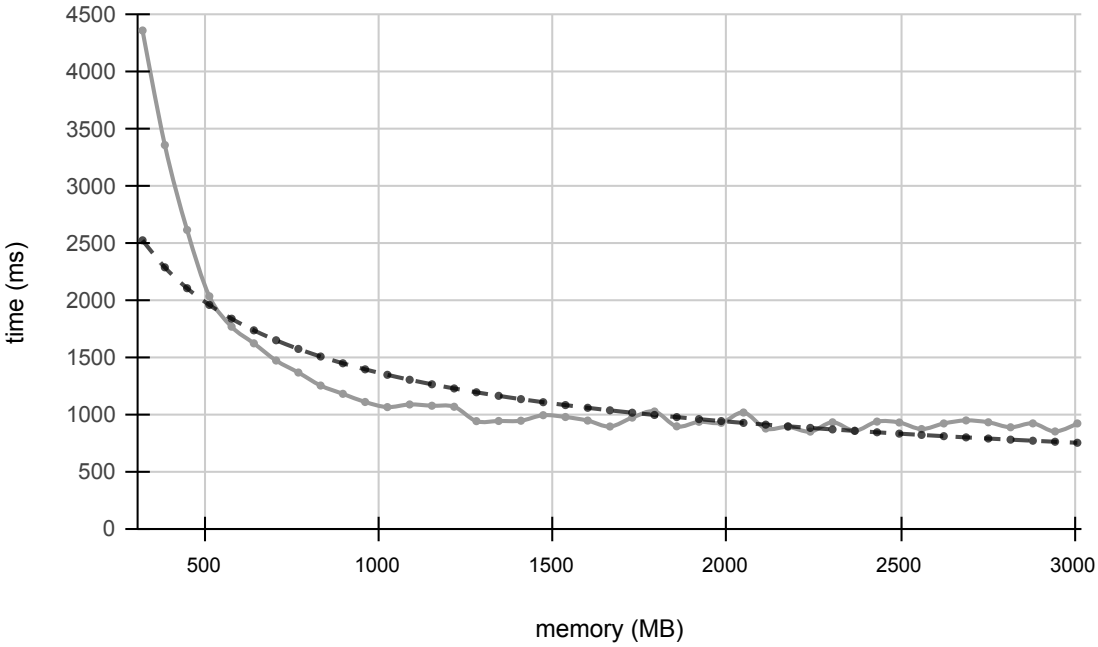

Figure 25: Result of the regression analysis for F20.

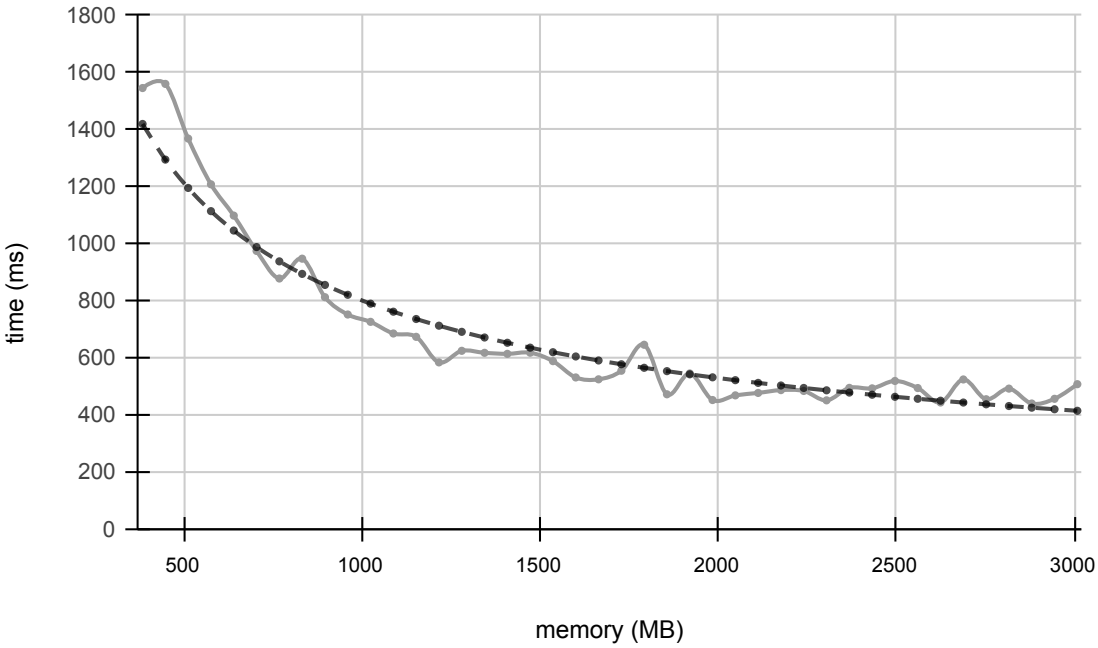

Figure 26: Result of the regression analysis for F21.

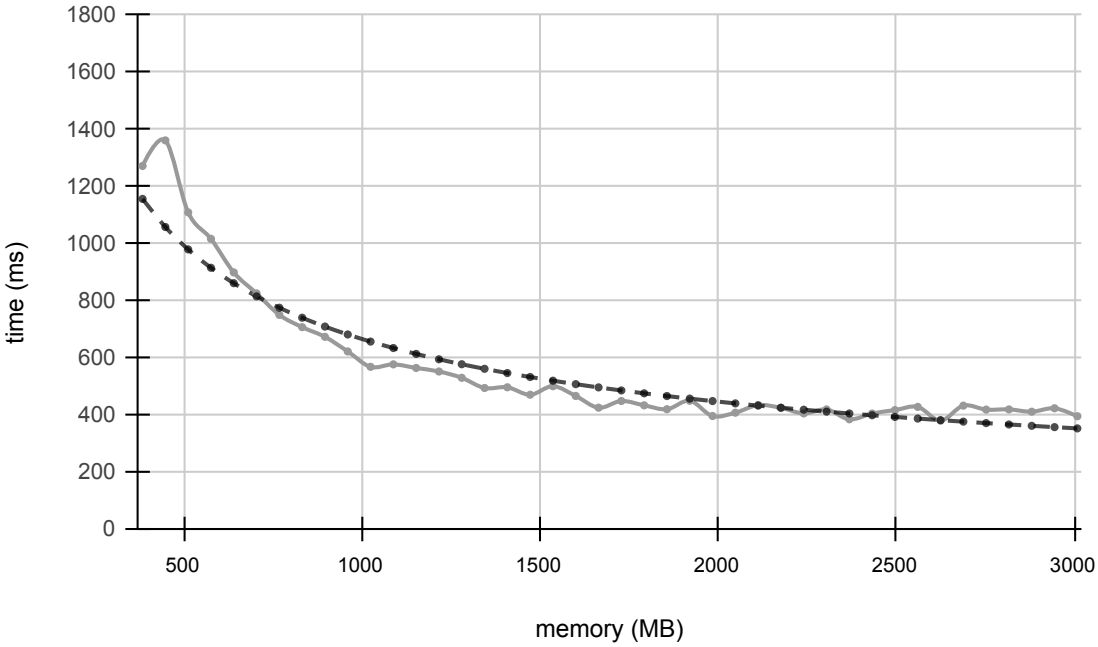

Figure 27: Result of the regression analysis for F22.

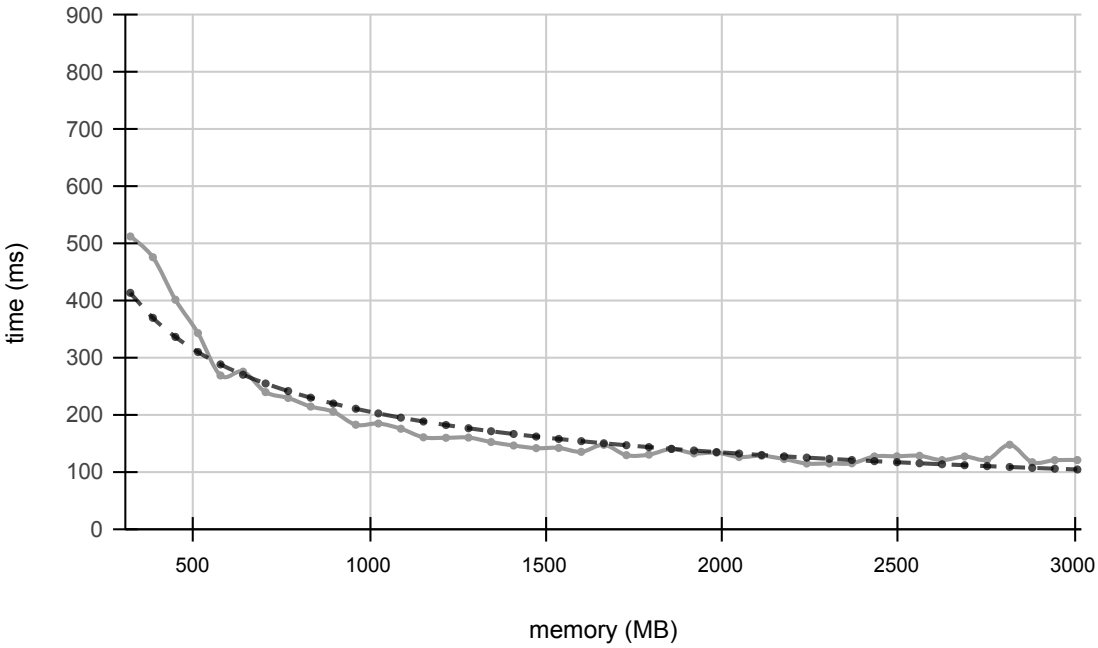

Figure 28: Result of the regression analysis for F23.

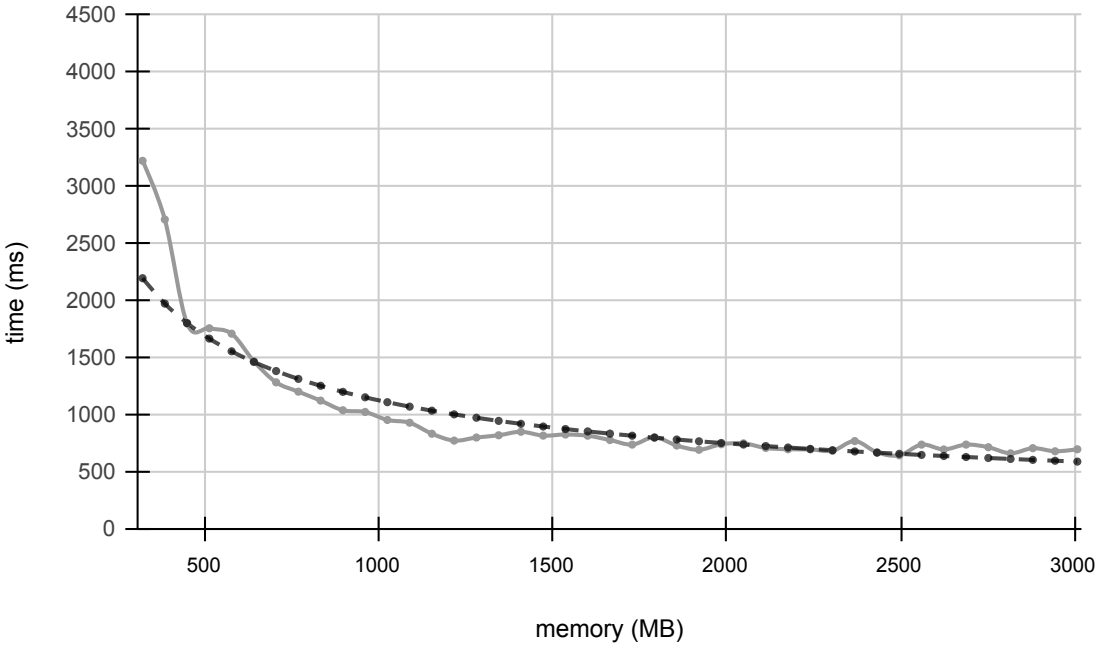

Figure 29: Result of the regression analysis for F24.

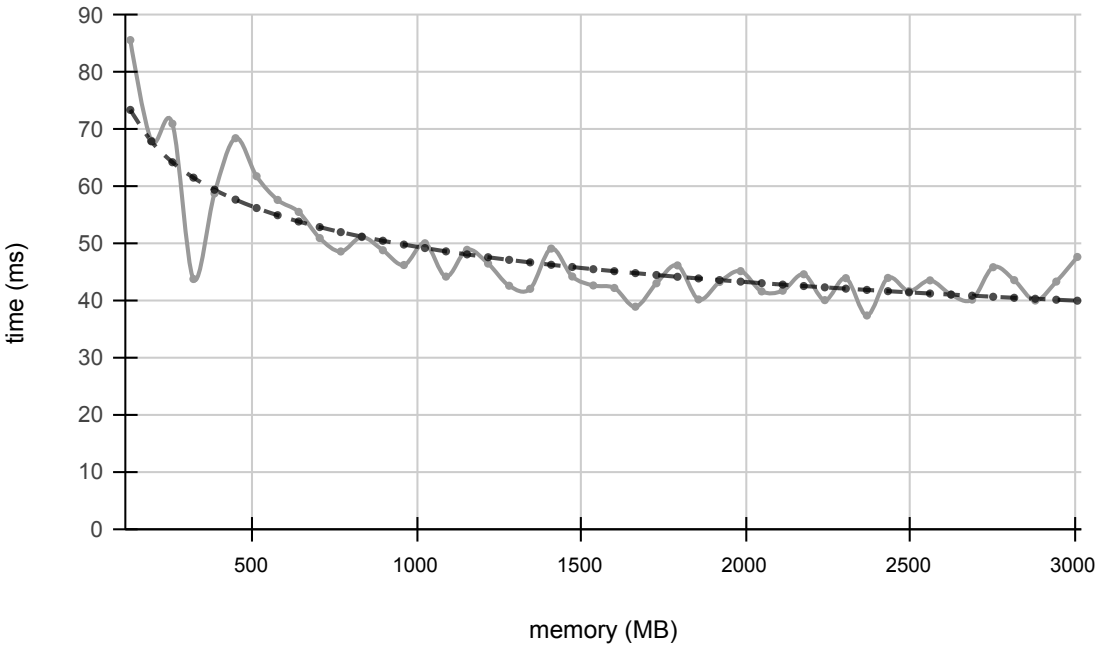

Figure 30: Result of the regression analysis for F25.

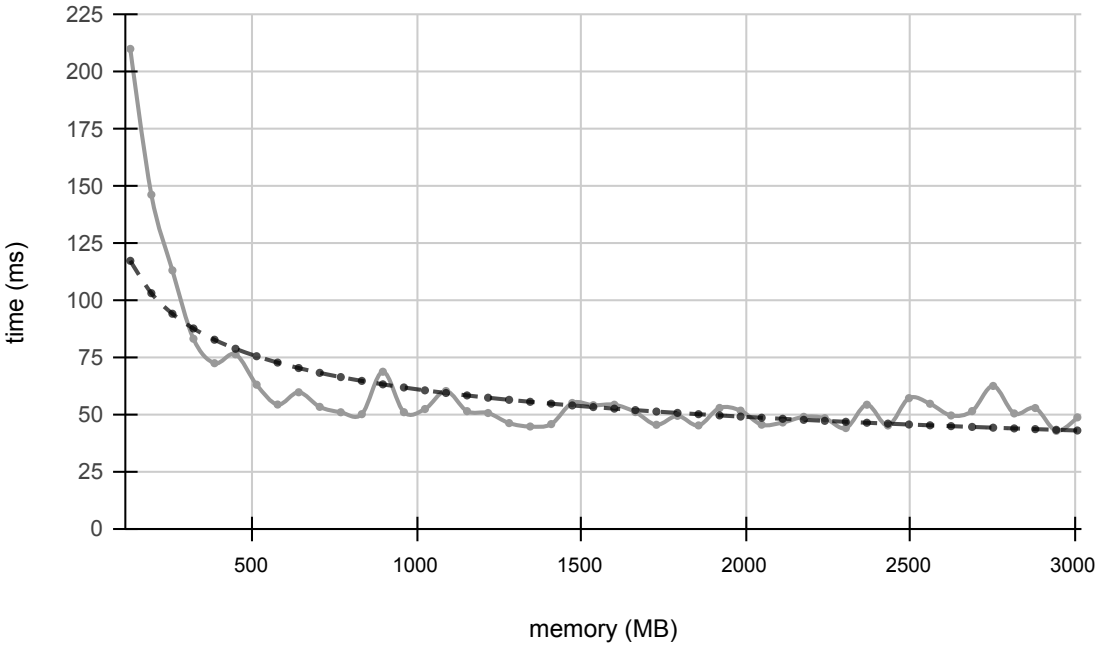

Figure 31: Result of the regression analysis for F26.

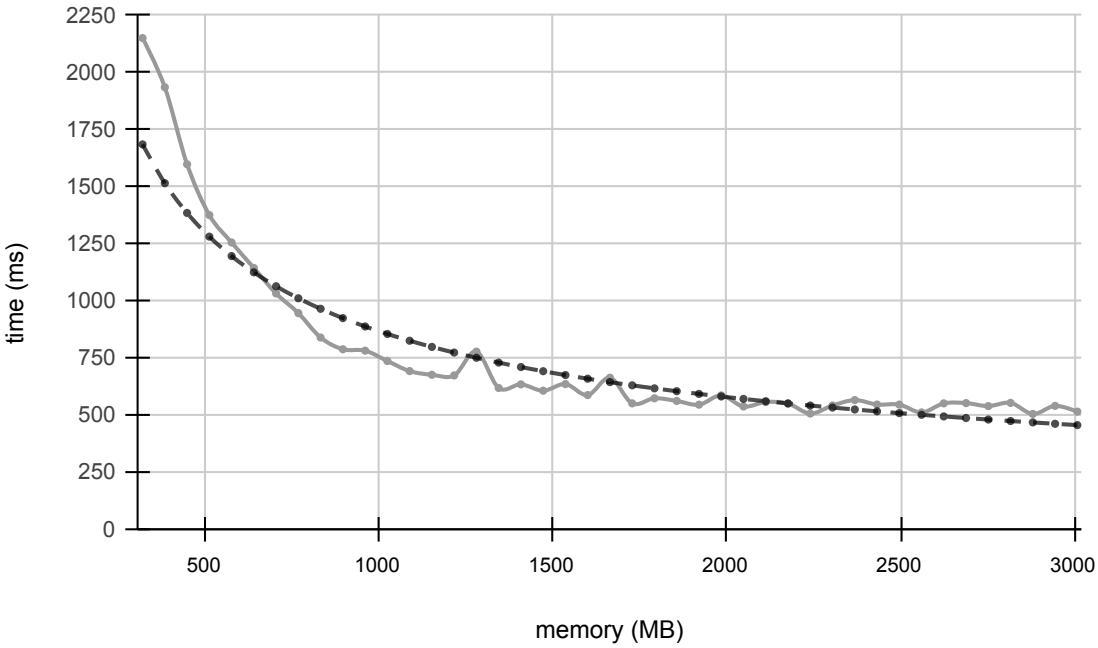

Figure 32: Result of the regression analysis for F27.

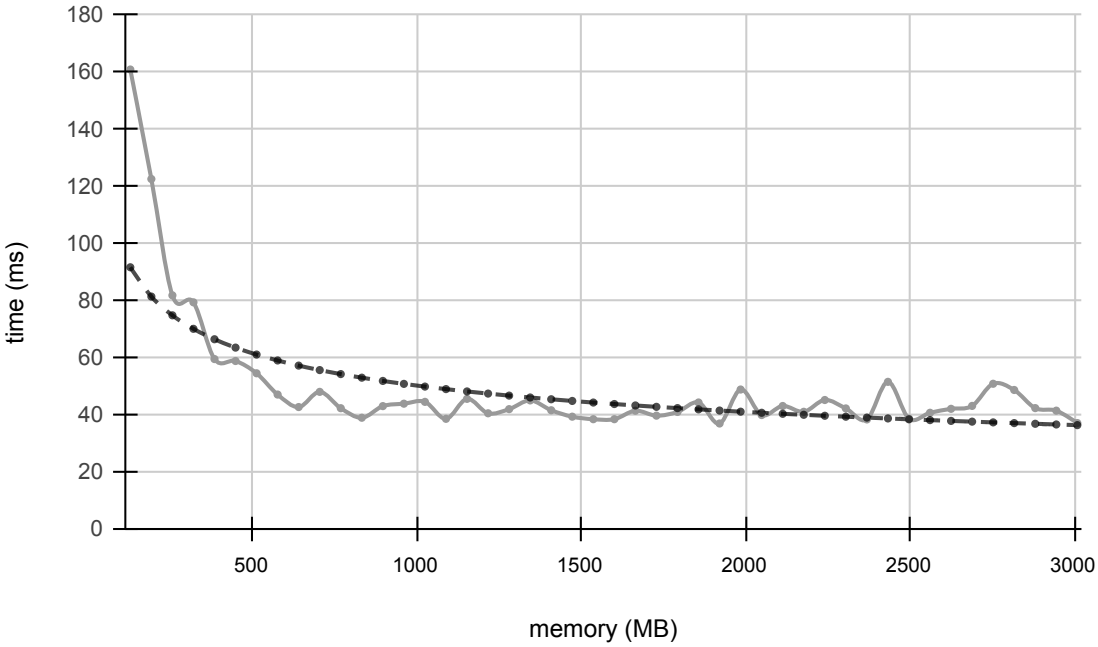

Figure 33: Result of the regression analysis for F28.

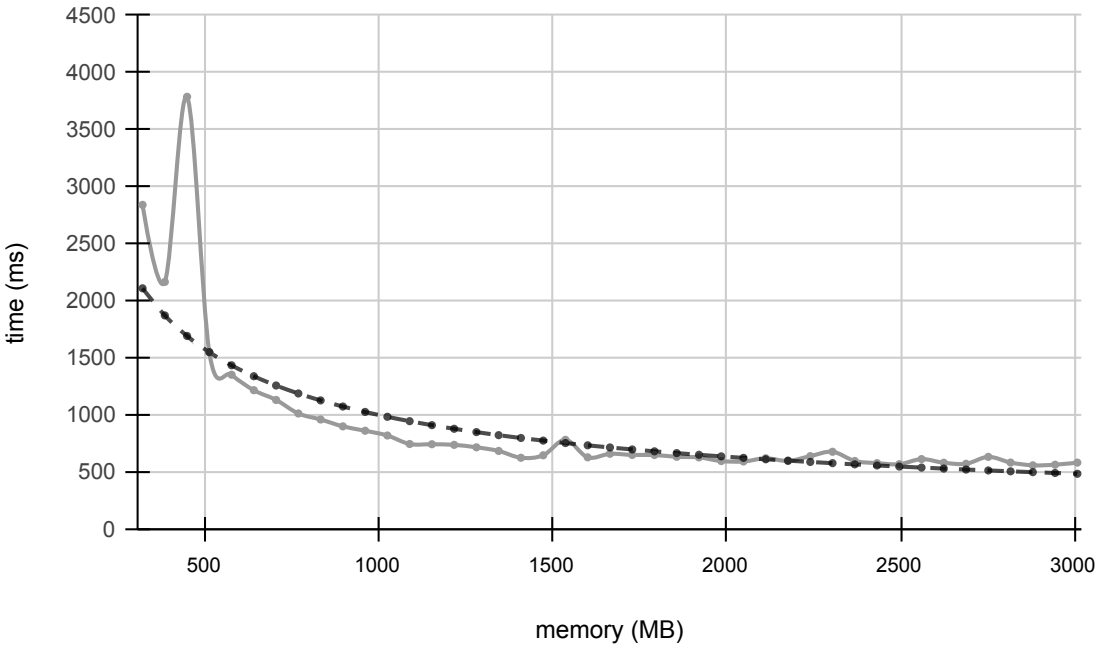

Figure 34: Result of the regression analysis for F29.

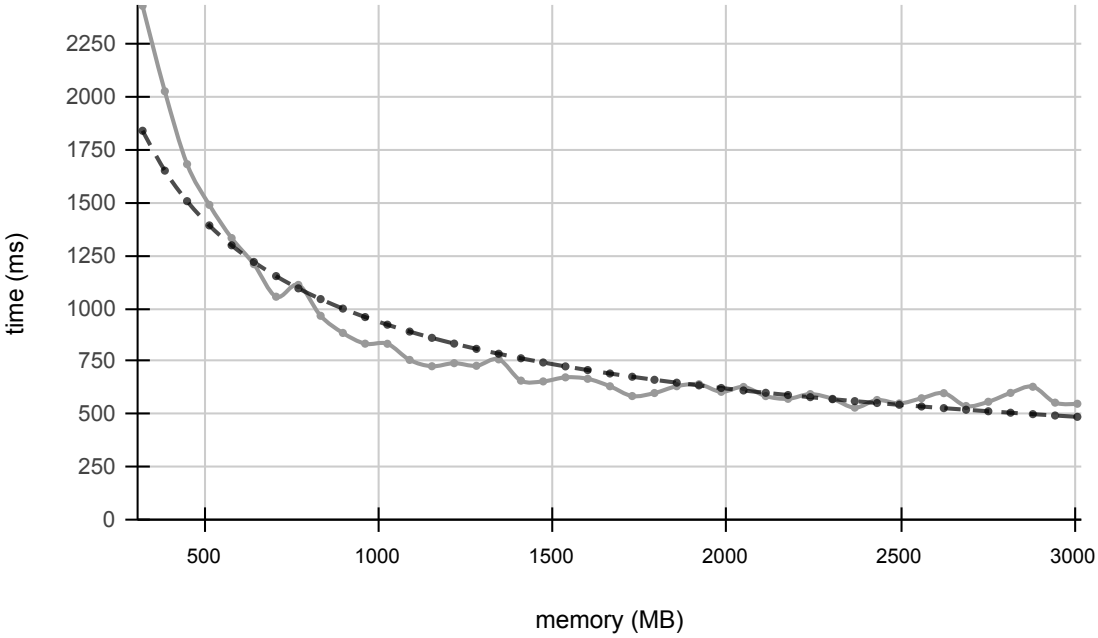

Figure 35: Result of the regression analysis for F30.

APPENDIX B

USE OF GEKKO FOR OPTIMIZATION

```
from gekko import GEKKO
import numpy as np
import sys

#Initialize Model
m = GEKKO(remote=True)

#define parameter
max = m.Const(value=3008)
min = m.Const(value=128)
step = m.Const(value=64)
limit = m.Const(value=400)
a = m.Const(value=sys.argv[2])
b = m.Const(value=sys.argv[2])

#initialize variables
x = m.CV(integer=True)
w = m.Var(integer=True)

#lower bounds
x.lower = min
w.lower = 0

#upper bounds
x.upper = max

#equations
```

```
m.Equation(x - w * step == 0)
m.Equation(a*x+b<=limit)

#objective
#min by default
m.Obj(a*x**2+b*x)

#set global options
m.options.IMODE = 3 #steady state optimization
m.options.SOLVER = 1 #set integer non-linear programming

#solve simulation
m.solve()

print('')
print('Result')
print('x: ' + str(x.value))
```

APPENDIX C

FACT USER MANUAL

FACT tool aims at minimizing the cost of deployed serverless functions from the application provider perspective. Detailed information regarding the tool can be obtained at the Wiki page[1] of the GitHub repository that includes 3 separate Java 8 Maven projects. One of these projects, configuration optimizer, uses the GEKKO python3 framework. AWS Command Line Interface (CLI) is used for executing lambda functions. The tool can be installed using the command below at the root project folder, where the pom.xml file takes place:

```
mvn clean; mvn install -DskipTests
```

The usage of the tool for the overall approach involves 3 stages:

Performance Analyzer reads lambda function configurations from a file and executes the function under various memory settings. It calculates power/linear regression results for the next stage. Configuration Optimizer reads power/linear regression results and calculates the optimum amount of memory to be allocated for the function.

Function Profiler executes the function for 100 times, where the allocated memory is determined by the previous stage. This stage verifies if the allocation is enough for keeping the latency of the function under a specified LIMIT.

C.1 Performance Analyzer

In the fact/performance analyzer/src/main/java/com/vestel/iot/App.java class there exist the following variables that can be altered for specific configuration options.

[1] `https://github.com/ozgursedef/fact/wiki`

```
static final int internalLoop = 10;
static final int MINMEMORY = 128;
static final int MAXMEMORY = 3008;
static final int STEP = 64;
```

Use the commands below for compilation:

```
cd performance analyzer
mvn clean; mvn install -DskipTests
```

This step is for retrieving function parameters from the lambdaConfiguration.json file that is located at the folder fact/(root). These parameters include the name of the lambda service and payload, if any available, e.g.,

```
"service": { "name":"example_service_name", "payload":"{\"example
    \":\"payload\"}" }
```

This stage prints the results exported to the performance model.txt file for the predicted values. It prints out the regression analysis results to the r results.txt file.

To run and debug: Use VsCode Studio run and debug feature in the fact/performance analyzer/src/main/java/com/vestel/iot/App.java class

C.2 Configuration Optimizer

Python3 GEKKO library is used to calculate non-linear programming model results.

Use the commands below for compilation:

```
cd configuration_optimizer
mvn clean; mvn install -DskipTests
```

This step is for reading function regression results from the r results.txt file that is located at the folder fact/(root). It prints optimization results to the o results.txt file.

To run and debug: Use VsCode Studio run and debug feature in the fact/configuration optimizer/app/src/main/java/com/vestel/iot/App.java class

To run standalone:

```
python3 configuration_optimizer/app/scripts/pr.py -a a -b b -min m -
    index i -limit l
```

C.3 Function Profiler

Use the commands below for compilation:

```
cd function_profiler
mvn clean; mvn install -DskipTests
```

This step is for reading function optimization results from the o results.txt file that is located at the folder fact/(root). It prints optimization results to the plot data.txt file.

To run and debug: Use VsCode Studio run adn debug feature in the fact/function profiler/app/src/main/java/app/App.java

C.4 Adding a new cloud provider to the source code

In the repository, open performance-analyzer project. Under performance-analyzer root folder simply go;

```
fact/performance_analyzer/src/main/java/com/vestel/iot/
```

- App.java
- AwsCli.java
- ConfigurationManager.java
- LinearRegression.java
- PowerRegression.java
- Service.java

A new cloud provider java class should be created with implementing Cloud-Provider.java interface with following override methods;

```
fact/performance_analyzer/src/main/java/com/vestel/iot/
```

- Gcloud.java

```
public class Gcloud implements CloudProvider{

@Override
public void setMemory(double memorySize) throws IOException {

//logic for set memory using gcloud cli

}
@Override
public String invoke() throws IOException, InterruptedException {

//logic for invoking Google Function
//Simply parse duration and return

}}
```

Finally new cloud provider java class instance should be created at App.java file.

```
CloudProvider cp = new Gcloud{service}
```

Bibliography

[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*. Singapore: Springer Singapore, 2017, pp. 1–20.

[2] A. AWS. (2021) Lambda functions. (accessed 15 April 2021). [Online]. Available: https://aws.amazon.com/lambda

[3] M. Azure. (2021) Azure functions. (accessed 15 April 2021). [Online]. Available: https://azure.microsoft.com/en-us/services/functions

[4] G. Cloud. (2021) Cloud functions. (accessed 15 April 2021). [Online]. Available: https://cloud.google.com/functions

[5] I. Cloud. (2021) IBM cloud functions. (accessed 15 April 2021). [Online]. Available: https://cloud.ibm.com/functions

[6] A. Eivy and J. Weinman, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[7] O. Sedefoglu and H. Sozer, "Cost minimization for deploying serverless functions," in *Proceedings of the 36th ACM/SIGAPP Symposium On Applied Computing*, 2021, pp. X–Y.

[8] S. Chaisiri, R. Kaewpuang, B. Lee, and D. Niyato, "Cost minimization for provisioning virtual servers in amazon elastic compute cloud," in *Proceedigns of the 19th IEEE Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011, pp. 85–95.

[9] J. Entrialgo, J. Díaz, J. Garcia, M. Garcia, and D. Garcia, *Cost Minimization of Virtual Machine Allocation in Public Clouds Considering Multiple Applications*. Springer International Publishing, 2017, pp. 147–161.

[10] S. Chaisiri, B. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 164–177, 2012.

[11] F. Tseng, X. Wang, L. Chou, H. Chao, and V. C. M. Leung, "Dynamic resource prediction and allocation for cloud data center using the multiobjective genetic algorithm," *IEEE Systems Journal*, vol. 12, no. 2, pp. 1688–1699, 2018.

[12] Z. A. Mann, "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–34, 2015.

[13] S. R. Shishira, A. Kandasamy, and K. Chandrasekaran, "Survey on meta heuristic optimization techniques in cloud computing," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, 2016, pp. 1434–1440.

[14] A. AWS. (2021) Amazon CloudFormation. (accessed 15 April 2021). [Online]. Available: https://aws.amazon.com/tr/cloudformation/

[15] JSON. (2021) Introducing JSON. (accessed 15 April 2021). [Online]. Available: https://www.json.org/

[16] YAML. (2021) The official YAML web site. (accessed 15 April 2021). [Online]. Available: https://yaml.org/

[17] F. Legillon, N. Melab, D. Renard, and E. Talbi, "Cost minimization of service deployment in a multi-cloud environment," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2013, pp. 2580–2587.

[18] S. Pandey, A. Barker, K. K. Gupta, and R. Buyya, "Minimizing execution costs when using globally distributed cloud services," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 222–229.

[19] A. AWS. (2021) Amazon CloudFront. (accessed 15 April 2021). [Online]. Available: https://aws.amazon.com/cloudfront/

[20] K. Huang, Y. Lu, M. Tsai, Y. Wu, and H. Chang, "Performance-efficient service deployment and scheduling methods for composite cloud services," in *Proceedings of the IEEE/ACM 9th International Conference on Utility and Cloud Computing*, 2016, pp. 240–244.

[21] Y. Zhang, J. Sun, and J. Zhu, "An effective heuristic for due-date-constrained bag-of-tasks scheduling problem for total cost minimization on hybrid clouds," in *Proceedings of the International Conference on Progress in Informatics and Computing*, 2016, pp. 479–486.

[22] J. Luo, L. Rao, and X. Liu, "Data center energy cost minimization: A spatio-temporal scheduling approach," in *Proceedings of the 32nd IEEE International Conference on Computer Communications*, 2013, pp. 340–344.

[23] S. Chapra and R. Canale, *Numerical methods for engineers*, 6th ed. McGraw-Hill, 2010.

[24] L. Beal, D. Hill, R. Martin, and J. Hedengren, "Gekko optimization suite," *Processes*, vol. 6, no. 8, 2018.

[25] GEKKO. (2021) GEKKO optimization suite. (accessed 15 April 2021). [Online]. Available: https://gekko.readthedocs.io/

[26] C. I. for Operations Research (COIN-OR). (2021) Couenne exact solver. (accessed 15 April 2021). [Online]. Available: https://projects.coin-or.org/Couenne

[27] C. S. Gebizli and H. Sozer, "Model-based software product line testing by coupling feature models with hierarchical markov chain usage models," in *Proceedings of the 6th IEEE International Workshop on Model-Based Verification and Validation*, 2016, pp. 278–283.

[28] M. Alaa, A. Zaidan, B. Zaidan, M. Talal, and M. Kiah, "A review of smart home applications based on internet of things," *Journal of Network and Computer Applications*, vol. 97, pp. 48 – 65, 2017.

[29] A. AWS. (2021) Amazon IoT Core. (accessed 15 April 2021). [Online]. Available: https://aws.amazon.com/iot-core/

[30] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2012.