2014 IEEE 6th International Conference on Cloud Computing Technology and Science

Optimal Resource Provisioning for Scaling Enterprise Applications on the Cloud

Satish Narayana Srirama, Alireza Ostovar
Institute of Computer Science, University of Tartu
J. Liivi 2, Tartu, Estonia
{*srirama, alireza*}*@ut.ee*

***Abstract*—Over the past years organizations have been moving their enterprise applications to the cloud to take advantage of cloud's utility computing and elasticity. However, in enterprise applications or workflows, generally, different components/tasks will have different scaling requirements and finding an ideal deployment configuration and having the application to scale up and down based on the incoming requests is a difficult task. This paper presents a novel resource provisioning policy that can find the most cost optimal setup of variety of instances of cloud that can fulfill incoming workload. All major factors involved in resource amount estimation such as processing power, periodic cost and configuration cost of each instance type and capacity of clouds are considered in the model. Additionally, the model takes lifetime of each running instance into account while trying to find the optimal setup. Benchmark experiments were conducted on Amazon cloud, using a real load trace and through two main control flow components of enterprise applications, AND and XOR. In these experiments, our model could find the most cost-optimal setup for each component/task of the application within reasonable time, making it plausible for auto-scaling any web/services based enterprise workflow/application on the cloud.**

***Keywords*-Cloud computing; auto-scaling; enterprise applications; resource provisioning; optimization; control flows;**

I. Introduction

Cloud computing [1] has gained significant popularity over past few years. Employing service-oriented architecture and resource virtualization technology, cloud provides the highest level of scalability for enterprise applications with variant load. While cloud with its intrinsic features like elasticity and utility computing is interesting, for migrating enterprise applications to the cloud, one should worry about the ideal deployment configuration of the applications. In N-tier enterprise applications or workflows, generally, different components will have different scaling requirements and finding an ideal configuration and having it to scale up and down based on the incoming requests is a difficult task.

To be precise, since each deployment component or web service of an enterprise application, or task of a workflow requires different processing power to perform its operation, at time of load variation it must scale in a manner fulfilling its specific requirements the most. Scaling can be done manually, provided that the load change periods are deterministic, or automatically, when there are unpredicted spikes and slopes in the workload. A number of auto-scaling policies have been proposed so far. Some of these methods try to predict next incoming loads, while others tend to react to the incoming load at its arrival time and change the resource setup based on the real load rate rather than predicted one [2], [3]. However, in both methods there is need for an optimal resource provisioning policy that determines how many servers must be added to or removed from the system in order to fulfill the load while minimizing the cost.

Current methods in this field take into account several of related parameters such as incoming workload, CPU usage of servers, network bandwidth, response time, processing power and cost of the servers [4], [5]. Nevertheless, none of them incorporates the life duration of a running server and current deployment configuration, the metrics that can contribute to finding the most optimal policy. These parameters find importance when the scaling algorithm tries to optimize the cost with employing a spectrum of instance types featuring different processing powers and costs, which is very common in cloud computing environment.

In this paper, we propose a generic LP (linear programming) model that takes into account all major factors involved in scaling including periodic cost, configuration cost and processing power of each instance type, instance count limit of clouds, and life duration of each instance with customizable level of precision, and outputs an optimal combination of possible instance types suiting each component of an enterprise application the most. We created a simulation tool based on the proposed model and used 24-hour workload of ClarkNet Internet service provider (ISP) [6] to conduct real-time benchmark experiments on Amazon cloud infrastructure. The cost and efficiency of the model are also compared with Amazon AutoScale [4]. The results of the experiments suggest that our optimal policy is plausible for auto-scaling any web/services based enterprise workflow/application on the cloud. The paper is organized as follows:

Section II discusses scaling enterprise applications on the cloud. Section III describes the LP model along with the intuition and implementation details. Section IV discusses how the model can be applied and section V produces a detailed analysis of the model applied to XOR control flow. Section VI later discusses the related work while section VII concludes the paper with future research directions.

978-1-4799-4093-6/14 $31.00 © 2014 IEEE
DOI 10.1109/CloudCom.2014.24

II. Scaling enterprise applications on the cloud

Service Oriented Architecture (SOA) [7] is a trend in information systems engineering and the software industry's response to the problem of managing large monolithic applications. SOA is a component model that delivers application functionality as services to end-user applications and other services, bringing the benefits of loose coupling and encapsulation to the enterprise application integration, and thus helping in building complex enterprise applications based on proper enterprise integration principles [8].

Similarly, a workflow is composed of a set of activities utilizing computer systems to achieve a particular goal. Each component/activity/task runs a piece of the main process and the result will be the input to next one and this chain of components complete the initial large job [9]. Nowadays, creating web services hosted on the cloud is the most popular approach of implementing SOA and workflows.

Cloud computing is a style of computing in which, typically, resources scalable on demand are provided "as a service (aaS)" over the Internet to the users who need not have knowledge of, expertise in, or control over the cloud infrastructure that supports them. The provisioning of cloud services occurs at the Infrastructural level (IaaS) or Platform level (PaaS) or at the Software level (SaaS). With cloud, resources can be allocated to the applications on demand, meaning that you pay more just when you need to. The numerous and wide variety of servers (with different hardware power such as memory and processing capabilities, made feasible by the virtualization technology) supplied by clouds provide a great environment for deployment of any enterprise application including workflows.

As already mentioned, enterprise/workflow applications are composed of tasks, some of which are highly resource-intensive and some just perform light computations. The popular mechanism for running each service/task is binding them a cluster of instances managed by a *load balancer* (LB) which distributes the requests among them [10]. Servers can be added to or removed from the cluster based on varying load. This necessitates use of dynamic resources allocation of cloud computing, so that applications can employ more resources at time of load increase and return them when there is no need. This dual capability of cloud is called elasticity and plays the primary role in migrating enterprise applications to the cloud.

Load variation of applications follow different models, depending on the services they provide. If load changes are known in advance, the resource allocation can be performed manually, with adding or removing static number of instances. But, all applications incur some unpredicted load fluctuation, necessitating an automatic mechanism for supplying servers. Auto-scaling is a feature of cloud computing, e.g. Amazon AutoScale [4], which helps to add and remove instances on demand and transparent to the user. The user only has to configure an auto-scaling plan matching the workload of their application. Auto-scaling is highly suitable for applications that experience hourly, weekly or monthly variation in their workload.

The efficiency of an auto-scaling mechanism mainly depends on its resource allocation policy, evaluated by request loss rate and total resource cost. So the big challenge here is trying to maximize throughput while minimizing the cost. Some methods address this problem through forecasting future loads and supplying resources beforehand, and some choose to react to the incoming load after their arrival, being cautious in resource provisioning. Predictive methods' request loss rate is usually lower, but cost will be higher. There are also approaches that tackle this challenge using a combination of predictive and reactive models [2], [3].

However, designing an auto-scaling model for entire enterprise application/workflow is a complex task. In these applications, generally, each component will have its own specific requirements to scale. Moreover, in every public cloud there are a range of instance types, each one having a different power/price rate. Whereas a *large* instance might be more beneficial for one task, a *medium* might have a better performance for another one, so we shall allocate different instance types to different tasks in order to reduce the cost. Sometimes the optimal allocation for a task can be a composition of multiple types of instances. In addition, due to SLA (service-level agreement) of applications, some of the tasks may even have to run on completely different clouds, to support customers from different regions. Furthermore, the hourly pricing model popular among prominent cloud vendors such as Amazon also offers several challenges. Based on this 'pay as you go' model policy, cloud provider charges the cost of the whole hour once the instance enters another hour of its life, disregarding if it will fill that hour or it will live just for a fraction of it.

We studied the problem in detail and came up with a LP model based solution. In this model, each service can be located in a separate cloud, possessing different policies and infrastructure. Major inputs to our model include incoming workload of each task, processing power, periodic cost and configuration time of instance types, maximum instance count limit of the cloud, and age of each running instance. The model provides the optimal number of instances from each type that must be added to or removed from cluster of each task, resulting in handling the workload and minimizing the cost. Based on this model we have created a RESTful web service which is accessible through a HTTP request and provides the ideal deployment configuration for the applications based on the varying loads.

III. The optimization model

Before elaborating the model we will try to clarify the intuition behind the model and make a few definitions clear that will be used for explaining it.

A. Intuition behind the model

Let us assume that an enterprise application (with one scalable component) is provided which is to be migrated to the cloud. Let us also assume that we have two instance types of *small* and *medium* available for our application, with costs \$0.25 and \$0.4 per hour respectively. After performance testing the application with each of the instances we found out that the small type can handle 6 requests per second (r/s) and the medium one 12 r/s. We assume that current workload is 6 r/s and therefore we have one small instance running. Suppose that after 50 minutes the workload increases to 12 r/s. If we do not consider age of instances the best solution is to add another small instance, however, this is not the optimal solution. If we add another small instance, assuming that 12 r/s workload persists, after ten minutes we must pay for the first instance as well. As a result we will have spent \$0.5 for the two small instances after ten minutes.

Now let us add a medium instance instead of the small one at time of load change (after 50 minutes). In this case, when the first instance fills its paid hour it will be shut down, since our medium instance can handle 12 r/s and we do not need the initial small one anymore. As a result we have paid less (\$0.4) and we still satisfy the workload. The only issue left is that 10-minute time that first small instance can still live, which must be involved in the calculations. In first scenario, since we take advantage of this instance for 10 minutes, we must subtract this profit from the calculated cost, and in the second scenario this amount of money is an additional cost, because we can already handle the incoming load using the added medium instance and these last ten minutes the application is overprovisioned with servers. The 10-minute cost of a small instance is \$0.04 (0.25 * 10 / 60). A summary of all stated calculations is listed here:

$$\begin{aligned} &Cost\ of\ scenario\ 1\ =\ (cost\ of\ two\ small\ instances)\ - \\ &(10 - min\ profit\ of\ small\ instance) = 0.5 - 0.04 = 0.46 \\ &Cost\ of\ scenario\ 2\ =\ (cost\ of\ a\ medium\ instance)\ + \\ &(10 - min\ cost\ of\ small\ instance) = 0.4 + 0.04 = 0.44 \end{aligned} \tag{1}$$

So we will save \$0.02 by adding a medium instance instead of a small one. Hence, we need a model which takes account of life duration of current instances and outputs the most optimal setup of instances.

B. Key definitions

Region: Based on the operation that each component/task in an enterprise application/workflow performs it might have different performances in different clouds or regions of a specific cloud. So the best practice is to host each task in the cloud suiting it the most. Hence, the model considers a region with its own independent characteristics for each task, this region can be in a different cloud or the same cloud as other tasks. Each region will scale independently based on its own incoming load entering its load balancer. Each region can also have its own capacity of instances as a parameter which will be shown by *CC*. The set of regions is shown with *R*, where each region hosts a component of the enterprise application. We will use *r* as region notation.

Instance Type: Since each cloud provides a wide spectrum of instance types with different resources and prices, it is more beneficial for the application to take advantage of multiple instance types. Therefore, each region in our model can include multiple instance types, each having its own processing power (P), price per period (C), capacity constraint (CCT), and configuration time (CT). Configuration time specifies the duration needed for an instance to initialize and switch to running status. During this time the instance is not usable, causing a further cost which must be considered in addition to the periodic cost of the instance. This additional cost is named as configuration cost. The set of instance types of region r is shown with T_r notation. Instance type will be marked with *t* letter.

Time Bag: Instances are typically charged on a periodic basis, such that when an instance enters a new period of its life it will be charged for the whole period. Depending on how we divide this period, during the period the instance resides in several time intervals, till it totally fills the period. In our model, each of the time intervals is called *time bag*. The length of this period can be set as a parameter, and based on desired level of granularity, number of time bags can be different, however, by default we consider one hour for period length, which is the common denomination of pay-as-you-go model of popular clouds like Amazon EC2, and 60 number of time bags (one per each minute) for each instance type. Time bags will help us to position each instance at any given point of time. Each time bag can contain several instances running at this point of time, and over the time these instances travel through all time bags till end of their hourly life. Time bags of each instance type have the same fixed price calculated by dividing price per period of an instance type by total number of time bags. The set of time bags of instance type t in region r is shown with $TB_{r,t}$ notation. We will show time bags with *tb* notation.

Killing Cost: The money we lose when we kill an instance before it fills its paid period is called killing cost. It is calculated by first subtracting number of the time bag containing the instance from the total number of time bags and then multiplying the result by price of a time bag of containing instance type. Killing cost will be noted as *KC*.

Retaining Cost: Retaining cost stands the opposite of killing cost. It is calculated by multiplying number of the time bag containing the instance by price of each time bag. Basically this is the cost of the lived duration of the paid period of an instance. Retaining cost will be noted as *RC*.

C. Method Description

Linear programming is a mathematical model targeting problems that must find the optimal solution among several

possible solutions. Each LP model consists of a linear objective function that must be optimized, subject to a set of equality or inequality constraints. These constraints make a feasible region for the problem, while the algorithm must try to find a point in this region that maximizes or minimizes the objective function. There are a set of parameters that are adjusted before running the model and there are variables that are assigned different values by the model in order to optimize the objective function.

Parameters of our model are listed below:

-$C_{r,t}$: Cost of a time period of instance type t running in region r.

-$CTB_{r,t}$: Cost of a time bag from instance type t running in region r. This cost is calculated by dividing the cost of a period of instance type t by total number of time bags.

-$CT_{r,t}$: Configuration time of instance type t running in region r. This value must be specified by time bag metric. For example in our experiments in next section, we consider first 3 time bags as configuration time, a number approximated by the time taken by instances from Amazon EC2 to start up and customize the configuration.

-$KC_{r,t,tb}$: Killing Cost of time bag tb from instance type t running in region r.

-$RC_{r,t,tb}$: Retaining Cost of time bag tb from instance type t running in region r.

-$X_{r,t,tb}$: The number of instances in time bag tb from instance type t running in region r.

-$P_{r,t}$: Processing power of instance type t running in region r.

-$CCT_{r,t}$: Capacity constraint (or instance count limit) of instance type t running in region r.

-W_r: Workload of region r. This is the current incoming workload to the system and must be provided by the same metric as $P_{r,t}$. meaning that if $P_{r,t}$ is calculated by request per second, $W_{r,t}$ must be provided as request per second too.

-CC_r: Capacity constraint (or instance count limit) of region r. The number varies per cloud, e.g. a private cloud with limited capacity. Even in Amazon, by default customer can launch up to 20 instances. If more servers are needed, customer must make an application for it.

The model has 2 variables:

-$N_{r,t}$: The number of new instances from instance type t running in region r that must be added to the system.

-$S_{r,t,tb}$: The number of instances of time bag tb from instance type t in region r that must be shut down.

The model will try to find out how many new instances of each type must be added and how many instances of each time bag from each instance type must be removed to minimize the resource provisioning cost in each of the regions. Therefore the objective function is as follows:

$$\begin{aligned} Min\ (&\sum_{i=1}^{n}\sum_{j=1}^{m}(N_{r_i,t_j} * C_{r_i,t_j} + N_{r_i,t_j} * (CT_{r_i,t_j} * CTB_{r_i,t_j})) \\ &+ \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{q} S_{r_i,t_j,tb_k} * KC_{r_i,t_j,tb_k} + \\ &\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{q}(X_{r_i,t_j,tb_k} - S_{r_i,t_j,tb_k}) * RC_{r_i,t_j,tb_k}) \end{aligned} \tag{2}$$

Following constraints are to be fulfilled by the model.

-The workload constraint $\forall$ regions r $\in$ R:

$$\sum_{j=1}^{m}(N_{r,t_j} + (\sum_{k=1}^{q} X_{r,t_j,tb_k} - S_{r,t_j,tb_k})) * P_{r,t_j} \geq W_r \tag{3}$$

-The cloud capacity constraint $\forall$ regions r $\in$ R:

$$\sum_{j=1}^{m}(N_{r,t_j} + (\sum_{k=1}^{q} X_{r,t_j,tb_k} - S_{r,t_j,tb_k})) \leq CC_r \tag{4}$$

-Instance type capacity constraint $\forall$ instance types t $\in$ T_r:

$$N_{t_r} + (\sum_{k=1}^{q} X_{t_r,tb_k} - S_{t_r,tb_k}) \leq CCT_{t_r} \tag{5}$$

-Shutdown constraint $\forall$ time bags tb $\in$ $TB_{r,t}$:

$$S_{tb_{r,t}} \leq X_{tb_{r,t}} \tag{6}$$

-And:

$$\begin{aligned} N_{r,t} &\geq 0 \\ S_{r,t} &\geq 0 \end{aligned} \tag{7}$$

The objective function comprises sum of all costs attached to changing the arrangement of resources at any point of time. The cost function, sums cost of new instances and their configuration, killing cost of each instance that must be shut down and retaining cost of each instance that will continue living. For each region the model outputs the number of new instances of each instance type that must be added and number of instances of each time bag from each instance

type that must be terminated so that the cost becomes minimal and all constraints are fulfilled.

Killing cost and retaining cost are the most valuable parameters of the model. The concept of time bags allows us to calculate these costs for each instance at any point of time. These two new parameters actually specify how valuable a running instance is still for us. The more the instance lives in its current time period, the retaining cost becomes higher and the killing cost become lower, and thus the instance becomes less valuable. This guarantees that during scale-down the instances from the last time bags will have higher chance to be terminated. However, adding a new instance is bound to a new configuration process that triggers redundant cost which might make the addition of the new instance unprofitable. This is avoided by adding configuration cost to objective function.

The constraint (3) is defined to ensure that the new setup will fulfill the incoming workload in each region. Furthermore, the total number of instances in each region must not exceed its capacity; this is fulfilled using the constraint (4). The constraint (5) checks that the number of instances of each instance type in each region does not surpass its limit. And finally using constraint (6) we make sure that from each time bag the model does not shut down more instances than it contains.

D. Method Implementation

The model is implemented in OptimJ [11] and using one of its free solvers, GLPK (GNU Linear Programming Kit) [12]. OptimJ is a Java-based modeling language for solving optimization problems including linear programming, mixed integer programming and nonlinear programming. This utility is an extension of Java programming language and is supported by the popular IDE, Eclipse. Due to Java-based nature of OptimJ, developers can have access to the whole Java library inside OptimJ modules. OptimJ has an easy interface to define decision variables, linear objective function and constraints, using straightforward keywords and structures. The engine of OptimJ translates the code, written by developer using the provided interface, to pure Java code at compile time, calling all required optimization functions. This tool has several LP solvers such as GLPK, lpsolve, CPLEX and MOSEC that can be used for solving LP problems, of them first two are free and open source. We also created a RESTful interface for the model implementation, so that it can be offered as a service.

IV. Application of the model for autoscaling enterprise applications

Once an enterprise application is provided, which is to be migrated to the cloud, we first have to identify the components which are scalable in the application. For example, a simple web/SOA application generally will have an application server handling business logic and a backend database (DB) server. When the load of the application raises, we can employ a load balancer (LB) and can add/remove servers (horizontal scaling) to/from the LB, dynamically. However, DB server can't be scaled horizontally, easily. Parallel databases are quite tricky and the recent developments in the domain include NoSQL, which are mainly non-relational, distributed data stores that often do not attempt to provide ACID guarantees, all the time. However, for simplicity the DB node can be considered as non-scalable component.

The scalable components are then to be load tested on the planned cloud, to extract the application specific parameters, discussed in the model. We later can provide the in-coming load information from different regions to the LP model and the model can produce the ideal deployment configuration for the complete enterprise application. Once the configuration is identified, it can be enforced with standard deployment scripts of starting the type of the instances and taking care of the application configuration to scale the system to the load. The process can be repeated at regular intervals, e.g. once per minute, and thus the application can be auto-scaled dynamically.

However, when we consider long running applications, the method would be a bit disruptive to the deployment configuration. The model suggests the most ideal configuration for the system for the coming load, at that particular time. But, this means some nodes will be removed/added from the setup, with each interval. To counter the problem, once the new deployment configuration is suggested, we do not have to terminate the instances immediately. They will be put in flagged state so that they can be terminated when they move to the bag where it has exactly enough time, so that it can be killed without having to pay for the next period. The flagged instances will be considered as the current setup for the next interval calculation and sometimes they may move back to the ideal deployment configuration. The scenario will be demonstrated further in the next section.

V. Analysis of the approach

In order to evaluate the model, we have designed a few test case scenarios. We have chosen two of the basic workflow control structures that are essence of many other complex structures and are used in any typical enterprise application/workflow, namely Parallel (AND) and Exclusive (XOR). The experiments are designed to measure cost-effectiveness, CPU utilization and time consumption of the model, and average response time and request loss rate of the load. Due to page limitations, only the results of the XOR control flow are discussed here in detail and are observed along with Amazon AutoScale mechanism to show the efficiency of our model.

Experiments were performed on Amazon EC2 USEast region. Amazon provides a wide variety of instances in multiple regions, fulfilling a broad range of customers with different requirements. In addition to various hardware

platforms, customer can choose among several operating systems such as Microsoft Windows, Red Hat Linux, SuSE Linux and Ubuntu, each in both 32 and 64 bit versions. In our experiments we used different instance types, *m1.small*, *m1.medium*, *m1.large*, *c3.large*, *c3.xlarge* and *r3.large* [13]. The first three were used as available instances for resource provisioning of the workflow, and the C3 types were chosen as the nodes assisting in experimental process, respectively, node with model implementation and LBs. The R3 instance type was used to generate load on the system.

ClarkNet was an Internet Service Provider (ISP) in United States between 1993 till 2003. The workload of this ISP between August 28, 1995 and September 4, 1995 is publicly available [6]. We cut a 24-hour load from 00:00:00 till 23:59:59 of August 29th for our experiments. We cut the load by minute, normalized and scaled it up, to reach a workload of 700 requests per second at peak time. Tsung [14] was used to generate the respective load for the system. Tsung has been widely used for stress testing of applications, and is capable of sending thousands of requests per second and is able to change the request rate according to the defined sessions by user. We changed the load according to the scaled load of ClarkNet on a minute basis.

We used Linux 64-bit as operating system of back-end instances running each task of the workflow and also instances running workflow control system. Ubuntu 64-bit was used on the instance running `tsung` (*r3.large*), and Microsoft Windows 64-bit was employed for running the simulation program, implementing our LP model.

A. Simulation Tool

We created a simulation tool that acts as a resource provisioning system, meaning that it fetches the load entering each task of the workflow, and feeds it to the model to calculate the amount of resources needed to handle the load. Based on the decision made by the resource provisioning policy, the application adds new instances to or removes the running ones from the system. At launch time, the instance receives the local time of the server as a timestamp with millisecond precision. This timestamp can be used at any moment for finding the number of periods an instance has lived and its time bag in current time period. Having the time bag number, we can easily calculate the killing and retaining cost of the instance.

Each task of the workflow has an Nginx load balancer which receives the load from the workflow management system (WMS) and passes it to its back-end servers. For fetching the load from the load balancers, the HttpStubStatusModule of Nginx is used, such that an HTTP request is sent to the address of each load balancer and the status of the load balancer, including total number of served requests is received. Subtracting previous registered requests number from this new number will give us the load change during the last interval.

We considered 60 time bags for each instance type, one per minute, and updated the instance setup of the workflow once per minute, each time based on the request rate of last minute. Configuration time for all instances are set to 3 minutes and each instance needs 3 minutes to finish the termination process, meaning that if an instance is set to be killed, it must be killed at 57th minute of its last life period. But, if an instance passes this minute in its current period, it will not be killed anymore and is considered as an extended instance which will live another time period, adding the cost of the new period to the total cost. We let each instance live till this minute even if it is set to be terminated before. At each setup update epoch, we set the status of all the instances such that they can have another chance to continue living, and then we run the resource provisioning policy. If based on the output of running the policy an instance must be terminated, we set its status accordingly, and if it is in its 57th minute, we will shut down the instance. In addition, according to configuration time, an instance is not considered as running until it passes 3 minutes of its current life period first time it is launched. So when calculating the current load capacity of each region, we just count the instances with running status. However, when running the policy in next updates we considered all launched instances even though they are not still in running status.

All tasks of the workflow can have three instance types of *m1.small*, *m1.medium* and *m1.large*. Based on the incoming load and processing power/price rate of these instance types, the resource provisioning policy searches for the most optimal combination of them which minimizes the cost and handles the workload. Each task of a workflow performs a specific operation, consuming specific amount of resources and taking specific amount of time. We considered a workflow with three tasks for our experiments, each of which does a different job. First task runs the Huffman coding, second one performs a Selection sort and task 3 conducts a Merge sort on each incoming request to the task. The codes are written in PHP language and are run on the respective servers. We stress tested *m1.small*, *m1.medium* and *m1.large* instances of Amazon using Tsung and measured their average power with the metric of requests per second (RPS). The results are shown in table I. Note that these numbers are specific to the application, and are to be calculated for each deployment component before migrating any enterprise application to the cloud, using our model.

In order to remove the effect of cloud capacity limit on results of our experiments, we set the capacity of cloud and each instance type to 100 instances, so that models can launch as many instances as needed for handling the load.

B. Test Case Scenario, Exclusive Structure

In this test case scenario, we considered a workflow management system consisting of an Exclusive OR gate (XOR). In an Exclusive gate, each time just one of the

Instance Type	Huffman Coding (RPS)	Selection Sort (RPS)	Merge Sort (RPS)	Price of instance ($)
m1.small	6	7	7	0.044
m1.medium	12	15	16	0.087
m1.large	19	25	25	0.175

Table I
PROCESSING POWER (RPS) AND COST OF INSTANCE TYPES

tasks connected to output of the gate is triggered. Each output branch of an Exclusive gate can have a weight which specifies how often each branch is activated, such that the branch with higher weight receives more load. In our experiments the branch connected to task 1 (in region 1) has the weight of 60 and the branch connected to task 2 (in region 2) has the weight of 40, meaning that 60% of the load is passed to the region 1 and 40% of it to the region 2. After receiving the successful response from the selected region, the request is redirected to the task 3 (in region 3). The structure of the experiment is shown in Figure 1.

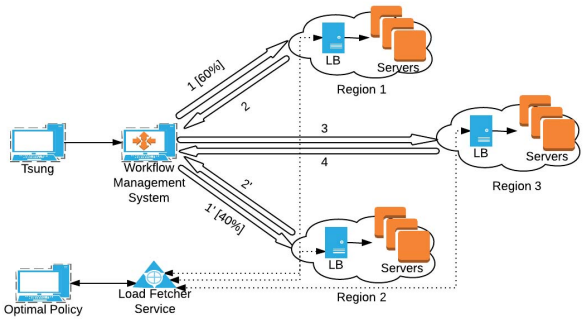


Figure 1. Workflow management system consisting of XOR gate

Table II shows the results of the experiments for regions 1, 2 and 3, after running for 24 hours in Amazon cloud. For observing the cost and efficiency of our optimization model, we ran the same experiments using Amazon AutoScale and Elastic Load Balancing. The results are also produced in Table II. For Amazon AutoScale mechanism, we considered only m1.medium instance, which is the most optimal one of the considered instances (Table I). In the launch configuration of AutoScale, we considered 45% and 65% as the average CPU utilization across the cluster for scaling down and scaling up 10% of the system configuration, respectively.

Figures 2, 3 and 4 represent the incoming load curve (a - red), scaling curve (a - blue), the instance type usage curves for the optimization model (b) and the scaling curve with Amazon AutoScale mechanism (c), of regions 1, 2 and 3, respectively.

From these figures we can see that, in all three regions, our optimization model followed the incoming load very

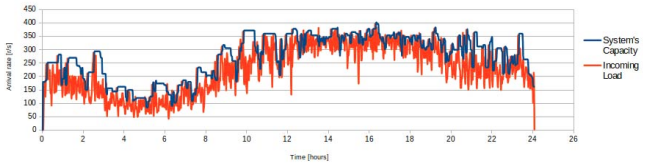
	Region 1	Region 2	Region 3	Total system
Optimal policy				
Total requests	19,771,558	13,180,299	32,897,804	32,951,857
Average response time (sec)	0.151	0.135	0.071	0.258
Request loss	22,657	31,396	396	54,449
Successful requests	99.885%	99.762%	99.998%	99.834%
Total cost of instances	50.846$	28.086$	64.278$	143.211$
Amazon Autoscale				
Total requests	19,725,901	13,152,246	32,873,371	32,878,147
Average response time (sec)	0.110	0.109	0.086	0.415
Request loss	2,221	2,555	3,288	8,064
Successful requests	99.988%	99.98%	99.99%	99.975%
Total cost of instances	49.068$	26.274$	61.509$	136.851$
Optimal policy - Normalized				
Total requests	20,029,944	13,342,840	33,355,671	33,372,784
Average response time (sec)	0.096	0.083	0.050	0.180
Request loss	16,080	1,033	30,594	47,707
Successful requests	99.919%	99.992%	99.908%	99.857%
Total cost of instances	46.938$	26.111$	59.519$	132.568$

Table II
RESOURCE PROVISIONING EXPERIMENTS' RESULTS IN DIFFERENT REGIONS FOR XOR CASE

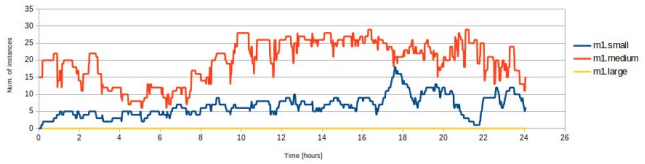
precisely. The registered incoming load curves and scaling curves display how well the model sticks to the load. The number of instances from each instance type launched in each region can also be observed in figures. The results suggest that, in all regions, optimal policy used more medium instances and less small instances on most of the occasions. However, small instances are still being added to the system to support slight variations in load, whenever it is cost efficient, thus trying to find the most optimal combination of different-types of instances at each setup update. The model did not launch large instances in any regions, which were not cost efficient in executing the designed codes.

From the results shown in Table II and the figures, we could observe that our optimization model performs at least as good as Amazon AutoScale, sometimes outperforming it in efficiency and mostly in total response times. Cost of Amazon AutoScale was slightly lower. However, one should not draw a direct comparison between the approaches. The results are produced to show that the model adjusts properly with the real-world load.

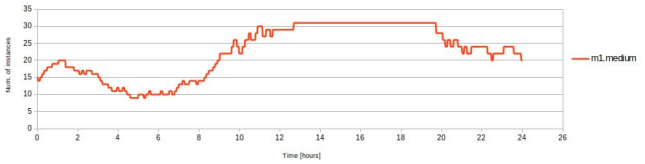
To be precise, the optimization model is generic and can be used with any number of instance types (even hundreds across different clouds), and the model can still find the ideal deployment configuration. With AutoScale one has to mention the instance type explicitly. In the experiments



(a) Incoming load curve and scaling curve

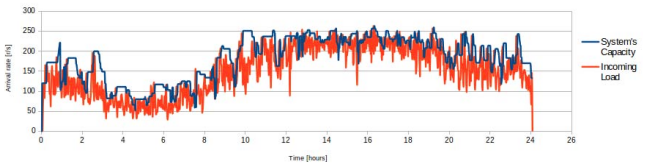


(b) Instance type usage curves for the optimization model

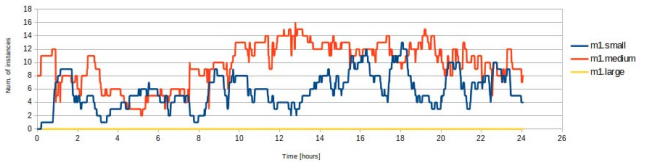


(c) Scaling curve with Amazon AutoScale

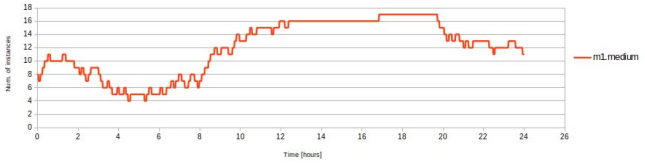
Figure 2. Results of region 1 for XOR case



(a) Incoming load curve and scaling curve



(b) Instance type usage curves for the optimization model

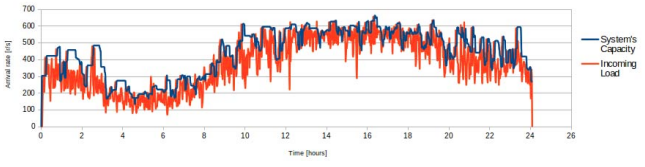


(c) Scaling curve with Amazon AutoScale

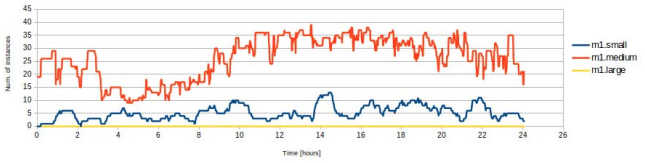
Figure 3. Results of region 2 for XOR case

presented here the *m1.medium* instance turned out to be a clear winner and we deliberately considered that instance for Amazon AutoScale mechanism. This made AutoScale save cost, however, this knowledge is not always obvious.

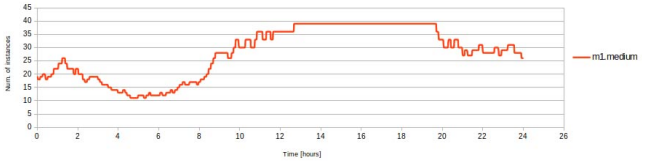
Moreover, the system to be migrated and auto-scaled, can span across multiple clouds. For example, one of the major challenges with cloud migration is the availability [1]. If



(a) Incoming load curve and scaling curve



(b) Instance type usage curves for the optimization model



(c) Scaling curve with Amazon AutoScale

Figure 4. Results of region 3 for XOR case

due to some reasons (maintenance issues, security attacks or bankruptcy of the cloud provider) the cloud is not available anymore the application also ceases to exist. In these cases, the best practice is to span the application across multiple cloud providers. The optimization model is applicable even under such circumstances.

Alternatively, further enhancements can be added to the simulation tool, to make the model more efficient. For example, instead of taking the load of the last minute, if we take the mean load from last five minutes, the costs were observed to be much lower, when compared to Amazon AutoScale (Table II - *Optimal policy - Normalized*).

We also have studied the effect of individual parameters on the optimal policy. For this, we developed a *mini-optimal* policy, where we could exclude certain parameters (such as KC and RC) and run the simulation on local infrastructure. Here we observed that KC and RC had maximum effect on optimal policy when the smaller/cheaper instances had relatively better performance than the medium instances.

Apart from the cost and efficiency, we are also interested in the performance of the execution of the model. Generally LP models are extremely resource-intensive. We used SIGAR [15] to measure CPU utilization of model execution. CPU utilization and durations are summarized in Table III. From these results, we can observe that the model could find the most cost-optimal setup, well within a second, with reasonable load on the node (*C3.large*).

VI. Related work

Over the past years, organizations have been moving their enterprise applications to the cloud with the aim of

	CPU Utilization	Execution Latency (milliseconds)
Min.	0.000%	0
1st Qu.	0.000%	0
Median	0.010%	18
Mean	5.475%	501.9
3rd Qu.	0.070%	648.5
Max.	54.820%	24,072

Table III
CPU UTILIZATION AND EXECUTION LATENCY OF THE MODEL

reducing infrastructure ownership and maintenance costs and to take advantage of the elasticity offered by the cloud. There are also several EU FP7 projects (REMICS [16], MODAClouds [17] and PaaSage [18]), which are actually targeted at migrating SOA based applications to the cloud. We were part of REMICS where we developed frameworks for monitoring and testing web/SOA application scalability on the cloud [3], and were involved in developing and standardizing automatic deployment strategies for cloud based applications [19].

Regarding Auto-scaling strategies, having been used by many auto-scaling services such as Amazon Auto-Scale [4], Scalr [20] or RightScale [5], threshold-based policies are very popular among users due to their simplicity. These policies generally observe performance metrics such as CPU usage, request rate, response time, network traffic and etc., which can be specified by user. At runtime when the conditions fulfill the defined requirements auto-scaling service alarm is triggered and automatic scaling is performed. However, proper setting of these parameters vary among applications according to their workload, and there is a need for expert knowledge of load and cloud computing to set up an optimal service. In addition, these methods generally do not consider parameters like costs other than the limits, current deployment configuration or addition of multiple types of instances to support load efficiently.

Regarding optimal resource provisioning policies, apart from LP several other technologies can also be used. Dutreilh et al. [21] used Reinforcement Learning (RL) [22], a machine learning algorithm, where current virtual instances can be defined as state, changing request rate as environment and the action is finding optimal number of instances fulfilling the load within a decent response time. However, RL has some disadvantages: since it is dependent on the experience and learning, it does not have good initial performance, and the time it takes to converge to an optimal policy can be quite long. Moreover, performance of this approach is suitable if the load incurs smooth changes, while if there are sudden bursts in load, it cannot react well. Dutreilh et al. tried to improve the approach with better initialization and faster convergence to optimal policy.

Similarly, Urgaonkar et al. [23] using Queuing theory, experimented a network of queues in a multi-tier application. He derived future workload from a load predictor and based on that found the adequate number of servers that can handle the load within the desired response time. Harold et al. [24] using Control Theory, propose a simple controller that produces the output based on average CPU usage. Ali-Eldin et al. [25] suggest to consolidate an adaptive and proactive controller for scaling-down process and a reactive model for scaling-up. Fuzzy controllers are also well-used, in which workload as input is mapped to the optimal amount of resources as output. Xu et al. [26] utilized a fuzzy model to estimate CPU capacity needed for handling the incoming workload. We also have tried other models and in [3], we combined heuristics and queuing models with a reactive model for auto-scaling of simulated MediaWiki application.

However, to our knowledge the model presented in this paper is the most comprehensive one considering all major factors involved in scaling including periodic cost, configuration cost and processing power of each instance type, instance count limit of clouds, and life duration of each instance with customizable level of precision. Additionally, the model takes lifetime of each running instance into account while trying to find the optimal setup.

VII. CONCLUSIONS AND FUTURE WORK

This paper introduced a novel resource provisioning policy that can find the most optimal setup of instances that fulfills incoming workload and minimizes the resource cost through taking full advantage of various available instance types of cloud. The presented LP model finds the optimal setup of each task/component in a workflow/SOA application at each run. Thus the model allows each component of the enterprise application to be hosted in a different cloud with different policies and instance types, suiting the task the most. All major factors involved in resource amount estimation such as processing power, periodic cost and configuration cost of each instance type and capacity of clouds are considered in the model. Additionally, the model takes lifetime of each running instance into account while trying to find the optimal setup, contributing to value determination of each instance at any time and discovering the optimal number of instances from each instance type in each region that must be added to or removed from the current setup. Using new concept of time bags and calculating two new costs bound to each running instance, namely killing and retaining cost, this method searches among all cost-effective configuration transformations, resulted from switching between various instance types having different processing power/price rates.

Benchmark experiments were conducted on the model using a real load trace and through two main control flow components of enterprise applications, AND and XOR. In these experiments our generic LP model could find the most cost-optimal setup for each task of the workflow at any point

of time within a decent amount of time. With the feasibility of the approach in basic models, we can conclude that the model is applicable for auto-scaling any web/SOA based enterprise workflow/application on the cloud.

Regarding future work, even though the presented model considers most major parameters related to the scaling of a system in the cloud, there are still some parameters such as network bandwidth that can be added to the model. In data-centric applications, network bandwidth of the system plays the main role in scaling decision rather than processing power of the servers. So in contrast to the service-based applications, in which we could mostly benefit from changing the number of instances, in data-centric applications we can optimize the system by changing the network usage.

Apart from this, we are also interested in adapting/remodeling enterprise applications for cloud migration. We propose remodeling and scheduling the applications, in a way that increases the intra-instance communication while reducing inter-instance communication, so that the applications will fit nicely to the cloud networks. The applications can be monitored for performance and partitioned with graph partitioning approaches [27]. Following the approach and joining the optimal resource provisioning policy we would like to achieve a framework to which any enterprise application can be provided, which would be studied, remodeled, migrated to, performance monitored and auto-scaled on the cloud, seamlessly.

ACKNOWLEDGMENT

This work is supported by European Regional Development Fund through EXCS, Estonian Science Foundation grant PUT360 and Target Funding theme SF0180008s12.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12*, 2012.

[3] M. Vasar, S. N. Srirama, and M. Dumas, "Framework for monitoring and testing web application scalability on the cloud," in *Nordic Symp. on Cloud Computing & Internet Technologies (NORDICLOUD)*. ACM, 2012, pp. 53–60.

[4] *Amazon Auto Scaling*. [Online]. Available: http://aws.amazon.com/autoscaling/

[5] *Rightscale*. [Online]. Available: http://support.rightscale.com/

[6] *ClarkNet-HTTP*. [Online]. Available: ftp://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html

[7] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, April 2004.

[8] S. N. Srirama, "Mobile hosts in enterprise service integration," Ph.D. dissertation, RWTH Aachen University, Germany, 2008.

[9] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and parallel databases*, vol. 14, no. 1, pp. 5–51, 2003.

[10] V. Cardellini, M. Colajanni, and S. Y. Philip, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, pp. 28–39, 1999.

[11] *OptimJ*. [Online]. Available: http://www.ateji.com/optimj/index.html

[12] *GLPK optimizer*. [Online]. Available: http://www.gnu.org/software/glpk/

[13] *Amazon EC2 Instances*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[14] *Tsung*. [Online]. Available: http://tsung.erlang-projects.org/

[15] *Sigar*. [Online]. Available: http://www.hyperic.com/products/sigar

[16] REMICS, "Reuse and migration of legacy applications to interoperable cloud services." [Online]. Available: http://www.remics.eu/

[17] MODAClouds, "MOdel-Driven Approach for design and execution of applications on multiple Clouds." [Online]. Available: http://www.modaclouds.eu/

[18] PaaSage, "Paasage: Model-based cloud platform upperware." [Online]. Available: http://www.paasage.eu/

[19] A. Sadovykh, A. Abhervé, S. Srirama, P. Jakovits, M. Smialek, W. Nowakowski, N. Ferry, and B. Morin, "Deliverable D4. 5 REMICS Migrate Principles and Methods," 2010.

[20] *Scalr*. [Online]. Available: https://scalr-wiki.atlassian.net/wiki/display/docs/Home

[21] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *7th Intl Conf. on Autonomic and Autonomous Systems (ICAS 2011)*, 2011, pp. 67–74.

[22] *Reinforcement Learning*. [Online]. Available: http://en.wikipedia.org/wiki/Reinforcement_Learning

[23] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2005, pp. 291–302.

[24] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009, pp. 13–18.

[25] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS 2012)*. IEEE, 2012, pp. 204–212.

[26] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the use of fuzzy modeling in virtualized data center management," in *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*. IEEE, 2007, pp. 25–25.

[27] S. N. Srirama and J. Viil, "Migrating Scientific Workflows to the Cloud: Through Graph-partitioning, Scheduling and Peer-to-Peer Data Sharing," in *Int. Conf. on High Performance and Communications (HPCC 2014) Workshops*. IEEE, 2014, pp. 1137–1144.